Panda3D developer manual

Table of Contents

Introduction

Hi, I'm frainfreeze.
This manual was made so I don't forget all the useful info other Panda3D developers told me.
I hope it will scale and eventually become big enough to be called official and to actually help someone.
I want to excuse for my poor English.
Huge, tera, extra , super 'thank you' to Panda community especially rdb who actively maintains Panda and of course others behind the scenes.

#note for manual writers
#info for .odt book source
-Made in LibreOffice
    Version: 4.3.2.2
    Build ID: edfb5295ba211bd31ad47d0bad0118690f76407d
-Added plug ins:
    1.
    Macro Formatter 3.0.4
    Copyright (c) Andrew Pitonyak
    2.
    Witer2LaTeX export filters 1.2.1

Preface

Book is organized in tree structure that follows panda 1.9 source tree.
It's bad, messy, confusing.
It also contains doc strings and comments from scripts.
Manual is built from several parts.

1. Source structured. It follows structure of panda source and notes are sorted in same way, under each item in tree.
2. File formats, specifications and similar
3. Miscellaneous and F.A.Q.
4. Appendix (Can contain code snippets or even whole scripts)

Note:
Maunal may be from 2014 but contents are much older. Some parts date from 2002 and loots of information might be deprecated. However most of Part 1 should be up to date at moment of writing.

Part 1

contrib

src

ai

contribbase

panda3dtoolsgui

sceneeditor

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.

Metalibs

direct

src

actor

autorestart

cluster

configfiles

controls

dcparse

dcparser

deadrec

directbase

directd

directdevices

directdServer

directnotify

directscripts

directtools

directutil

distributed

doc

extensions

extensions_native

ffi

filter

fsm

gui

heapq

http

interval

levelediter

motiontrail

p3d

particles

physics

plugin

plugin_activex

plugin_installer

plugin_npapi

plugin_standalone

pyinst

showbase

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

showutil

stdpy

task

test

tkpanels

tkwidgets

Wxwidgets

dmodels

src

Audio

sfx

fonts

gui

icons

level_editor

maps

misc

doc

dtool

Low-level libraries/tools for controlling CVS views, and interrogating C++ code.

metalibs

dtool

dtoolconfig

pptempl

src

attach

cppparser

dconfig

dtoolbase

dtoolutil

interrogate

interrogatedb

newheader

parser-inc

android

AR

libavcodec

libavformat

libavutil

libswscale

ode

ogg

vorbis

prc

prckeys

pystub

test_interrogate

makepanda

models

maps

plugin_images

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.

metalibs

panda

pandabullet

Panda has classes that represent underlying bullet objects, that basically wrap around it and integrate it with Panda classes and structures.
For instance, there's BulletRigidBodyNode, which is a class that extends a PandaNode and as such can be placed inside the panda scene graph.
However, it stores a btRigidBody object from bullet, and exposes methods that are wrappers around that underlying Bullet object.

pandadx8

pandadx9

pandaegg

pandaexpress

pandafx

pandagl

pandagles

pandagles2

pandaode

pandaphysics

pandaphysx

src

android

androiddisplay

audio

Audio API for panda

audiotraits

awesomium

bullet

cftalk

chan

Animation channels. This defines the various kinds of
AnimChannels that may be defined, as well as the MovingPart class
which binds to the channels and plays the animation. This is a
support library for char, as well as any other libraries that want
to define objects whose values change over time.

char

cocoadisplay

collada

collide

This package contains the classes that control and detect collisions

configfiles

This package contains the housekeeping and configuration files needed by things like attach, and emacs.

cull

This package contains the Cull Traverser. The cull traversal collects all state changes specified, and removes unneccesary state change requests. Also does all the depth sorting for proper alphaing.

device

Device drivers, such as mouse and keyboard, trackers, etc... The base class for using various device APIs is here.

dgraph

Defines and manages the data graph, which is the hierarchy of devices, tforms, and any other things which might have an input or an output and need to execute every frame.

display

Abstract display classes, including pipes, windows, channels, and display regions.

distort

doc

Documentation Panda3D developers considered that doesn't fit in any of the packages.
For contents please see the “part 2” of this manual.

downloader

Tool to allow automatic download of files in the background.

downloadertools

dxgsg8

Handles all communication with the DirectX backend, and manages state
to minimize redundant state changes.

dxgsg9

Handles all communication with the DirectX backend, and manages state
to minimize redundant state changes.

dxml

effects

Various graphics effects that aren't shaders. I.E Lens Flares

egg

A.k.a. the "egg library", this reads, writes, and manipulates egg files. It knows nothing about the scene graph structure in the rest of the player; it lives in its own little egg world.

Egg2pg

A.k.a. the "egg loader", this converts the egg structure read from
the egg library, above, to a scene graph structure, suitable for rendering.
egg2pg reads egg file and converts it to a Panda scene graph.
ie. in-memory structure of PandaNode etc.
I'm assuming "pg" stands for "panda graph".
Also, technically, the "egg" tree reads the .egg file into in-memory
EggData structures, and egg2pg converts those to scene graph structures.
When egg2pg converts that into scene graph structures egg files
from memory get deleted. If you want to keep them around, you can use
the lower-level interfaces yourself.

eglDisplay

event

Tools for throwing, handling and receiving events.

express

ffmpeg

framework

A simple, stupid framework around which to write a simple, stupid demo program. Handy for quickly writing programs that open a window and display the OmniTriangle.

gles2gsg

glesgsg

glgsg

Handles all communication with the GL backend, and manages state
to minimize redundant state changes.

glstuff

glxdisplay

X windows display classes that replace Glut functionality.

gobj

Graphical non-scene-graph objects, such as textures and geometry primitives.

grutil

gsgbase

Base GSG class defined to avoid cyclical dependency build.

gsgmisc

Some utility functions for gsg that could not live in the same directory for circular dependency reasons.

iphone

iphonedisplay

linmath

Linear algebra library.

mathutil

Math utility functions, such as frustum and plane

movies

nativenet

net

Net connection classes

ode

osxdisplay

pandabase

parametrics

particlesystem

Tool for doing particle systems. Contains various kinds of particles,
    emiters, factories and renderers.

pgraph

pgraphnodes

pgui

physics

Base classes for physical objects and forces. Also contains the
physics manager class

physx

pipeline

pnmimage

Reads and writes image files in various formats, by using the pnm
and tiff libraries.
PNMImage class manages reading and writing image files from disk
One of the properties of PNMImage is that all images are laid out (almost) the same way in memory, regardless of their properties. This makes it very easy to write a class like PNMPainter, which can paint equally well on grayscale, grayscale/alpha, 24-bit, 32-bit, or 64-bit images.

pnmimagetypes

pnmtext

ps2display (defunct)

Playstation 2 display classes. (defunct)

ps2gsg (defunct)

Play station 2 specific rendering backend. (defunct)

pstatclient

putil

recorder

rocket

skel

speedtree

testbed

C test programs, that primarily link with framework.

text

Package for generating renderable text using textured polygons.

tform

Data transforming objects that live in the data graph and convert raw data (as read from an input device, for instance) to something more useful.

tinydisplay

vision

vrpn

Defines the specific client code for interfacing to the VRPN API.

wgldisplay

Windows OpenGL specific display classes.

windisplay

x11display

pandatool

src

assimp

bam

configfiles

converter

cvscopy

daeegg

daeprogs

dxf

dxfegg

dxfprogs

egg-mkfont

egg-optchar

egg-palettize

egg-qtess

eggbase

eggcharbase

eggprogs

flt

fltegg

fltprogs

gtk-stats

imagebase

imageprogs

lwo

lwoegg

lwoprogs

maxegg

maxprogs

maya

mayaegg

mayaprogs

miscprogs

objegg

objprogs

palettizer

pandatoolbase

pfmprogs

progbase

pstatserver

ptloader

scripts

softegg

softprogs

text-stats

vrml

vrmlegg

vrmlprogs

win-stats

xfile

xfileegg

xfileprogs

ppremake

samples

Asteroids

Ball-in-Maze

Boxing-Robots

Bump-Mapping

Carousel

Cartoon-Shader

Chessboard

Disco-Lights

Distortion

Fireflies

Fractal-Plants

Glow-Filter

Infinite-Tunnel

Looking-and-Gripping

Media-Player

Motion-Trails

Music-Box

Particles

Procedural-Cube

Roaming-Ralph

Shadows

Solar-System

Teapot-on-TV

Texture-Swapping

thirdparty

Pmw

win-extras

JOD

phidgets

serial

smartroom

win-libs-vc10-x64

artoolkit

bullet

eigen

extras

fcollada

ffmpeg

fmodex

freetype

jpeg

npapi

nvidiacg

ode

openal

openssl

ovr

png

rocket

squish

tiff

vorbis

vrpn

zlib

win-nsis

win-python-x64

win-util

Part 2

The Graphics Engine

The GraphicsEngine is where it all begins. There is only one, global, GraphicsEngine in an application, and its job is to keep all of the pointers to your open windows and buffers, and also to manage the task of doing the rendering, for all of the open windows and buffers. Panda normally creates a GraphicsEngine for you at startup, which is available as base.graphicsEngine. There is usually no reason to create a second GraphicsEngine.

Note also that the following interfaces are strictly for the advanced user. Normally, if you want to create a new window or an offscreen buffer for rendering, you would just use the

```
base.openWindow()
```

or

```
window.makeTextureBuffer()
```

interfaces, which handle all of the details for you automatically.
However, please continue reading if you want to understand in detail how Panda manages windows and buffers, or if you have special needs that are not addressed by the above convenience methods.

GraphicsPipe

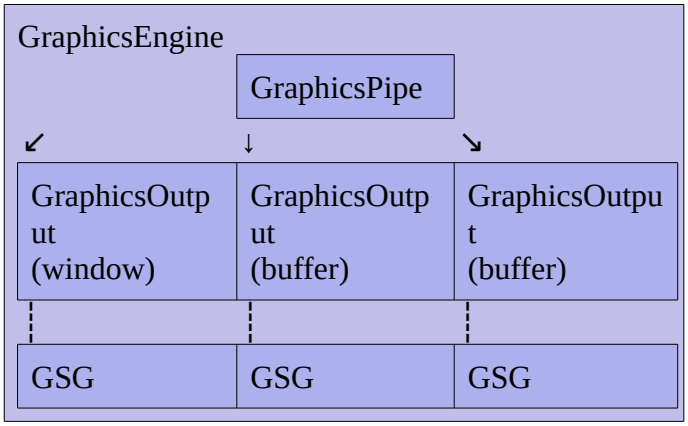
Each application will also need at least one GraphicsPipe. The GraphicsPipe encapsulates the particular API used to do rendering. For instance, there is one GraphicsPipe class for OpenGL rendering, and a different GraphicsPipe for DirectX. Although it is possible to create a GraphicsPipe of a specific type directly, normally Panda will create a default GraphicsPipe for you at startup, which is available in `base.pipe`.
The GraphicsPipe object isn't often used directly, except to create the individual GraphicsWindow and GraphicsBuffer objects.

GraphicsWindow and GraphicsBuffer

The GraphicsWindow class is the class that represents a single onscreen window for rendering. Panda normally opens a default window for you at startup, which is available in `base.win`. You can create as many additional windows as you like. (Note, however, that some graphics drivers incur a performance penalty when multiple windows are open simultaneously.)
Similarly, GraphicsBuffer is the class that represents a hidden, offscreen buffer for rendering special offscreen effects, such as render-to-texture. It is common for an application to have many offscreen buffers open at once.
Both classes inherit from the base class GraphicsOutput, which contains all of the code common to rendering to a window or offscreen buffer.

GraphicsStateGuardian

The GraphicsStateGuardian, or GSG for short, represents the actual graphics context. This class manages the actual nuts-and-bolts of drawing to a window; it manages the loading of textures and vertex buffers into graphics memory, and has the functions for actually drawing triangles to the screen. (During the process of rendering the frame, the "graphics state" changes several times; the GSG gets its name from the fact that most of its time is spent managing this graphics state.)
You would normally never call any methods on the GSG directly; Panda handles all of this for you, via the GraphicsEngine. This is important, because in some modes, the GSG may operate almost entirely in a separate thread from all of your application code, and it is important not to interrupt that thread while it might be in the middle of drawing.
Each GraphicsOutput object keeps a pointer to the GSG that will be used to render that window or buffer. It is possible for each GraphicsOutput to have its own GSG, or it is possible to share the same GSG between multiple different GraphicsOutputs. Normally, it is preferable to share GSG's, because this tends to be more efficient for managing graphics resources.
Consider the following diagram to illustrate the relationship between these classes. This shows a typical application with one window and two offscreen buffers:



The GraphicsPipe was used to create each of the three GraphicsOutputs, of which one is a GraphicsWindow, and the remaining two are GraphicsBuffers. Each GraphicsOutput has a pointer to the GSG that will be used for rendering. Finally, the GraphicsEngine is responsible for managing all of these objects.
In the above illustration, each window and buffer has its own GSG, which is legal, although it's usually better to share the same GSG across all open windows and buffers.

Rendering a frame

There is one key interface to rendering each frame of the graphics simulation:
`base.graphicsEngine.renderFrame()`
This method causes all open GraphicsWindows and GraphicsBuffers to render their contents for the current frame. In order for Panda3D to render anything, this method must be called once per frame. Normally, this is done automatically by the task "igloop", which is created when you start Panda.

Using a GraphicsEngine to create windows and buffers

In order to render in Panda3D, you need a GraphicsStateGuardian , and either a GraphicsWindow (for rendering into a window) or a GraphicsBuffer (for rendering offscreen). You cannot create or destroy these objects directly; instead, you must use interfaces on the GraphicsEngine to create them. Before you can create either of the above, you need to have a GraphicsPipe, which specifies the particular graphics API you want to use (e.g. OpenGL or DirectX). The default GraphicsPipe specified in your Config.prc file has already been created at startup, and can be accessed by base.pipe.

Now that you have a GraphicsPipe and a GraphicsEngine, you can create a GraphicsStateGuardian object. This object corresponds to a single graphics context on the graphics API, e.g. a single OpenGL context. (The context owns all of the OpenGL or DirectX objects like display lists, vertex buffers, and texture objects.) You need to have at least one GraphicsStateGuardian before you can create a GraphicsWindow:

```
myGsg=base.graphicsEngine.makeGsg(base.pipe)
```

Now that you have a GraphicsStateGuardian, you can use it to create an onscreen GraphicsWindow or an offscreen GraphicsBuffer:

```
base.graphicsEngine.makeWindow(gsg, name, sort)
base.graphicsEngine.makeBuffer(gsg, name, sort, xSize, ySize, wantTexture)
```

gsg is the GraphicsStateGuardian, name Is an arbitrary name you want to assign to the window/ buffer, and sort is an integer that determines the order in which the windows/buffers will be rendered. The buffer specific arguments xSize and ySize decide the dimensions of the buffer, and wantTexture should be set to True if you want to retrieve a texture from this buffer later on.
You can also use

```
graphicsEngine.makeParasite(host,name,sort,xSize,ySize)
```

where host is a GraphicsOutput object. It creates a buffer but it does not allocate room for itself. Instead it renders to the framebuffer of host. It effectively has wantTexture set to True so you can retrieve a texture from it later on. See The GraphicsOutput class and Graphics Buffers and Windows for more information.

```
myWindow=base.graphicsEngine.makeWindow(myGsg, "Hello World", 0)
myBuffer=base.graphicsEngine.makeBuffer(myGsg, "Hi World", 0, 800,600, True)
myParasite=base.graphicsEngine.makeBuffer(myBuffer,"Im a leech", 0, 800, 600)
```

Note: if you want the buffers to be visible add show-buffers true to your configuration file. This causes the buffers to be opened as windows instead, which is useful while debugging.

Sharing graphics contexts

It is possible to share the same GraphicsStateGuardian among multiple different GraphicsWindows and/or GraphicsBuffers; if you do this, then the graphics context will be used to render into each window one at a time. This is particularly useful if the different windows will be rendering many of the same objects, since then the same texture objects and vertex buffers can be shared between different windows.
It is also possible to use a different GraphicsStateGuardian for each different window. This means that if a particular texture is to be rendered in each window, it will have to be loaded into graphics memory twice, once in each context, which may be wasteful. However, there are times when this may be what you want to do, for instance if you have multiple graphics cards and you want to to render to both of them simultaneously. (Note that the actual support for simultaneously rendering to multiple graphics cards is currently unfinished in Panda at the time of this writing, but the API has been designed with this future path in mind.)

Closing windows

To close a specific window or buffer you use removeWindow(window). To close all windows

```
removeAllWindows()
base.graphicsEngine.removeWindow(myWindow)
base.graphicsEngine.removeAllWindows()
```

More about GraphicsEngine

Here is some other useful functionality of the GraphicsEngine class.
GetNumWindows() Returns the number of windows and buffers that this GraphicsEngine object is managing. IsEmpty() Returns True if this GraphicsEngine is not managing any windows or buffers. See API for advanced functionality of GraphicsEngine and GraphicsStateGuardian class.

ppython

ppython.exe is used for starting Panda3D. Basicly it is only duplicated copy of python.exe renamed so you don't mix Panda's python with other python on your PATH

Panda Audio Documenation

AudioManager and AudioSound

The AudioManager is a combination of a file cache and a category of sounds (e.g. sound effects, battle sounds, or music).

The first step is to decide which AudioManager to use and load it.

Once you have an AudioManager (e.g. effectsManager), a call to get_sound(<file>) on that manager should get you an AudioSound (e.g. mySound = effectsManager.getSound("bang")).

After getting a sound from an AudioManager, you can tell the sound change its volume, loop, start time, play, stop, etc. There is no need to involve the AudioManager explicitly in these operations.

Simply delete the sound when you're done with it. (The AudioSound knows which AudioManager it is associated with, and will do the right thing).

The audio system, provides an API for the rest of Panda; and leaves a lot of leaway to the low level sound system. This is good and bad. On the good side: it's easier to understand, and it allows for widely varrying low level systems. On the bad side: it may be harder to keep the behavior consistent accross implementations (please try to keep them consistent, when adding an implementation).

Example Usage

Python Example:

```
effects = AudioManager.createAudioManager()
music = AudioManager.createAudioManager()

bang = effects.load("bang")
background = music.load("background_music")

background.play()
bang.play()
```

C++ Example:

```
AudioManager effects = AudioManager::create_AudioManager();
AudioManager music = AudioManager::create_AudioManager();

bang = effects.get_sound("bang");
background = music.get_sound("background_music");

background.play();
bang.play();
```

Coding style

Almost any programming language gives a considerable amount of freedom to the programmer in style conventions. Most programmers eventually develop a personal style and use it as they develop code.

When multiple programmers are working together on one project, this can lead to multiple competing styles appearing throughout the code. This is not the end of the world, but it does tend to make the code more difficult to read and maintain if common style conventions are not followed throughout.

It is much better if all programmers can agree to use the same style when working together on the same body of work. It makes reading, understanding, and extending the existing code much easier and faster for everyone involved. This is akin to all of the animators on a feature film training themselves to draw in one consistent style throughout the film.

Often, there is no strong reason to prefer one style over another, except that at the end of the day just one must be chosen.

The following lays out the conventions that we have agreed to use within Panda. Most of these conventions originated from an amalgamation of the different styles of the first three programmers to do major development in Panda. The decisions were often arbitrary, and some may object to the particular choices that were made. Although discussions about the ideal style for future work are still welcome, considerable code has already been written using these existing conventions, and the most important goal of this effort is consistency. Thus, changing the style at this point would require changing all of the existing code as well.

Note that not all existing Panda code follows these conventions. This is unfortunate, but it in no way constitutes an argument in favor of abandoning the conventions. Rather, it means we should make an effort to bring the older code into compliance as we have the opportunity.

Naturally, these conventions only apply to C and C++ code; a completely different set of conventions has been established for Python code for the project, and those conventions will not be discussed here.

SPACING:

No tab characters should ever appear in a C++ file; we use only space characters to achieve the appropriate indentation. Most editors can be configured to use spaces instead of tabs.

We use two-character indentation. That is, each nested level of indentation is two characters further to the right than the enclosing level.

Spaces should generally surround operators, e.g. i + 1 instead of i+1. Spaces follow commas in a parameter list, and semicolons in a for statement. Spaces are not placed immediately within parentheses; e.g. foo(a, b) rather than foo(a,b).

Resist writing lines of code that extend beyond 80 columns; instead, fold a long line when possible. Occasionally a line cannot be easily folded and remain readable, so this should be taken as more of a suggestion than a fixed rule, but most lines can easily be made to fit within 80 columns.

Comments should never extend beyond 80 columns, especially sentence or paragraph comments that appear on a line or lines by themselves. These should generally be wordwrapped within 72 columns. Any smart editor can do this easily.

CURLY BRACES:

In general, the opening curly brace for a block of text trails the line that introduces it, and the matching curly brace is on a line by itself, lined up with the start of the introducing line, e.g.:

```
for (int i = 0; i < 10; i++) {
  ...
}
```

Commands like if, while, and for should always use curly braces, even if they only enclose one command. That is, do this:

```
if (foo) {
 bar();
}
```

instead of this:

```
if (foo)
 bar();
```

NAMING:

Class names are mixed case with an initial capital, e.g. MyNewClass. Each different class (except nested classes, of course) is defined in its own header file named the same as the class itself, but with the first letter lowercase, e.g. myNewClass.h.

Typedef names and other type names follow the same convention as class names: mixed case with an initial capital. These need not be defined in their own header file, but usually typedef names will be scoped within some enclosing class.

Local variable names are lowercase with an underscore delimiting words: my_value. Class data members, including static data members, are the same, but with a leading underscore: _my_data_member. We do not use Hungarian notation.

Class method names, as well as standalone function names, are lowercase with a delimiting underscore, just like local variable names: my_function().

LANGUAGE CONSTRUCTS:

Prefer C++ constructs over equivalent C constructs when writing C++ code. For instance, use:

```
static const int buffer_size = 1024;
```

instead of:

```
#define BUFFER_SIZE 1024
```

Resist using brand-new C++ features that are not broadly supported by compilers. One of our goals in Panda is ease of distribution to a wide range of platforms; this goal is thwarted if only a few compilers may be used.

More examples of the agreed coding style may be found in panda/src/doc/sampleClass.* file should be also in appendix of this manual.

COLLISION FLAGS

floor: for things that avatars can stand on
barrier: for things that avatars should collide against that are not floors
camera-collide: for things that the camera should avoid
trigger: for things (usually not barriers or floors) that should trigger an
    event when avatars intersect with them
sphere: for things that should have a collision sphere around them
tube: for things that should have a collision tube (cylinder) around them

NOTES

The barrier & camera-collide flags are typically used together.

Currently, the camera automatically pulls itself in front of anything marked with the camera-collide flag, so that the view of the avatar isn't blocked.

The trigger flag implies that avatars will not collide with the object; they can move freely through it.

The sphere & tube flags create a collision object that is as small as possible while completely containing the original flagged geometry.

egg palettize

HOW TO USE EGG_PALETTIZE

The program egg-palettize is used when building models to optimize texture usage on all models before loading them into the show. It is capable of collecting together several different small texture images from different models and assembling them together onto the same image file, potentially reducing the total number of different texture images that must be loaded and displayed at runtime from several thousand to several hundred or fewer.

It also can be used to group together related textures that will be rendered at the same time (for instance, textures related to one neighborhood), and if nothing else, it can resize textures at build time so that they may be painted at any arbitrary resolution according to the artist's convenience, and then reduced to a suitable size for texture memory management (and to meet hardware requirements of having dimensions that are always a power of two).

It is suggested that textures always be painted at high resolution and reduced using egg-palettize, since this allows the show designer the greatest flexibility; if a decision is later made to increase the resolution of a texture, this may be done by changing an option with egg-palettize, and does not require intervention of the artist.

The behavior of egg-palettize is largely controlled through a source file called textures.txa, which is usually found in the src/maps directory within the model tree. For a complete description of the syntax of the textures.txa file, invoke the command egg-palettize -H.

GROUPING EGG FILES

Much of the contents of textures.txa involves assigning egg files to various groups; assigning two egg files to the same group indicates that they are associated in some way, and their texture images may be copied together into the same palettes.

The groups are arbitrary and should be defined at the beginning of the egg file with the syntax:

```
:group groupname
```

Where groupname is the name of the group. It is also possible to assign a directory name to a group. This is optional, but if done, it indicates that all of the textures for this group should be installed within the named subdirectory. The syntax is:

```
:group groupname dir dirname
```

Where dirname is the name of the subdirectory. If you are generating a phased download, the dirname should be one of phase_1, phase_2, etc., corresponding to the PHASE variable in the install_egg rule (see ppremake-models.txt).

Finally, it is possible to relate the different groups to each other hierachically. Doing this allows egg-palettize to assign textures to the minimal common subset between egg files that share the textures. For instance, if group beta and group gamma both depend on group alpha, a texture that is assigned to both groups beta and gamma can actually be placed on group alpha, to maximize sharing and minimize duplication of palette space.

You relate two groups with the syntax:

```
:group groupname with basegroupname
```

Once all the groups are defined, you can assign egg files to the various groups with a syntax like this:

```
model.egg : groupname
```

where model.egg is the name of some egg model file built within the tree. You can explicitly group each egg file in this way, or you can use wildcards to group several at once, e.g.:

```
dog*.egg : dogs
```

Assigning an egg file to a group assigns all of the textures used by that egg file to that same group. If no other egg files reference the same textures, those textures will be placed in one or more palette images named after the group. If another egg file in a different group also references the textures, they will be assigned to the lowest group that both groups have in common (see relating the groups hierarchically, above), or copied into both palette images if the two groups having nothing in common.

CONTROLLING TEXTURE PARAMETERS

Most of the contents of the textures.txa is usually devoted to scaling the texture images appropriately. This is usually done with a line something like this:

```
texture.rgb : 64 64
```

where texture.rgb is the name of some texture image, and 64 64 is the size in pixels it should be scaled to. It is also possible to specify the target size as a factor of the source size, e.g.:

```
bigtexture.rgb : 50%
```

specifies that the indicated texture should be scaled to 50% in each dimension (for a total reduction to 0.5 * 0.5 = 25% of the original area).

As above, you may group together multiple textures on the same line using wildcards, e.g.:

```
wall*.rgb : 25%
```

Finally, you may include one or more optional keywords on the end of the texture scaling line that indicate additional properties to apply to the named textures. See egg-palettize -H for a complete list. Some of the more common keywords are:

```
mipmap - Enables mipmaps for the texture.
```

linear - Disables mipmaps for the texture.

omit - Omits the texture from any palettes. The texture will still be scaled and installed, but it will not be combined with other textures. Normally you need to do this only when the texture will be applied to some geometry at runtime. (Since palettizing a texture requires adjusting the UV's of all the geometry that references it, a texture that is applied to geometry at runtime cannot be palettized.)

RUNNING EGG-PALETTIZE

Normally, egg-palettize is run automatically just by typing:

make install

in the model tree. It automatically reads the textures.txa file and generates and installs the appropriate palette image files, as part of the whole build process, and requires no further intervention from the user. See ppremake-models.txt for more information on setting up the model tree.

When egg-palettize runs in the normal mode, it generates suboptimal palettes. Sometimes, for instance, a palette image is created with only one small texture in the corner, and the rest of it unused. This happens because egg-palettize is reserving space for future textures, and is ideal for development; but it is not suitable for shipping a finished product. When you are ready to repack all of the palettes as optimally as possible, run the command:

make opt-pal

This causes egg-palettize to reorganize all of the palette images to make the best usage of texture memory. It will force a regeneration of most of the egg files in the model tree, so it can be a fairly involved operation.

It is sometimes useful to analyze the results of egg-palettize. You can type:

```
make pi >pi.txt
```

to write a detailed report of every egg file, texture image, and generated palette image to the file pi.txt.

Finally, the command:

```
make pal-stats >stats.txt
```

will write a report to stats.txt of the estimated texture memory usage for all textures, broken down by group.

WHEN THINGS GO WRONG

The whole palettizing process is fairly complex; it's necessary for egg-palettize to keep a record of the complete state of all egg files and all textures ever built in a particular model tree. It generally does a good job of figuring out when things change and correctly regenerating the necessary egg files and textures when needed, but sometimes it gets confused.

This is particularly likely to happen when you have reassigned some egg files from one group to another, or redefined the relationship between different groups. Sometimes egg-palettize appears to run correctly, but does not generate correct palettes. Other times egg-palettize will fail with an assertion failure, or even a segment fault (general protection fault) when running egg-palettize, due to this kind of confusion. This behavior should not happen, but it does happen every once and a while.
When this sort of thing happens, often the best thing to do is to invoke the command:

```
make undo-pal
```

followed by:

```
make install
```

This removes all of the old palettization information, including the

state information cached from previous runs, and rebuilds a new set of palettes from scratch. It is a fairly heavy hammer, and may take some time to complete, depending on the size of your model tree, but it almost always clears up any problems related to egg-palettize.

THE PHILOSOPHY OF EGG FILES

THE PHILOSOPHY OF EGG FILES (vs. bam files)

Egg files are used by Panda3D to describe many properties of a scene: simple geometry, including special effects and collision surfaces, characters including skeletons, morphs, and multiple-joint assignments, and character animation tables.

Egg files are designed to be the lingua franca of model manipulation for Panda tools. A number of utilities are provided that read and write egg files, for instance to convert to or from some other modeling format, or to apply a transform or optimize vertices. The egg file philosophy is to describe objects in an abstract way that facilitates easy manipulation; thus, the format doesn't (usually) include information such as polygon connectivity or triangle meshes. Egg files are furthermore designed to be human-readable to help a developer diagnose (and sometimes repair) problems. Also, the egg syntax is always intended to be backward compatible with previous versions, so that as the egg syntax is extended, old egg files will continue to remain valid.

This is a different philosophy than Panda's bam file format, which is a binary representation of a model and/or animation that is designed to be loaded quickly and efficiently, and is strictly tied to a particular version of Panda. The data in a bam file closely mirrors the actual Panda structures that are used for rendering. Although an effort is made to keep bam files backward compatible, occasionally this is not possible and we must introduce a new bam file major version.

Where egg files are used for model conversion and manipulation of models, bam files are strictly used for loading models into Panda. Although you can load an egg file directly, a bam file will be loaded much more quickly.

Egg files might be generated by outside sources, and thus it makes sense to document its syntax here. Bam files, on the other hand,

should only be generated by Panda3D, usually by the program egg2bam. The exact specification of the bam file format, if you should need it, is documented within the Panda3D code itself.

GENERAL EGG SYNTAX

Egg files consist of a series of sequential and hierarchically-nested entries. In general, the syntax of each entry is:

```
<Entry-type> name { contents }
```

Where the name is optional (and in many cases, ignored anyway) and the syntax of the contents is determined by the entry-type. The name (and strings in general) may be either quoted with double quotes or unquoted. Newlines are treated like any other whitespace, and case is not significant. The angle brackets are literally a part of the entry keyword. (Square brackets and ellipses in this document are used to indicate optional pieces, and are not literally part of the syntax.)

The name field is always syntactically allowed between an entry keyword and its opening brace, even if it will be ignored. In the syntax lines given below, the name is not shown if it will be ignored.

Comments may be delimited using either the C++-style // ... or the C-style /* ... */. C comments do not nest. There is also a <Comment> entry type, of the form:

```
<Comment> { text }
```

<Comment> entries are slightly different, in that tools which read and write egg files will preserve the text within <Comment> entries, but they may not preserve comments delimited by // or /* */. Special characters and keywords within a <Comment> entry should be quoted; it's safest to quote the entire comment.

LOCAL INFORMATION ENTRIES

These nodes contain information relevant to the current level of nesting only.

```
<Scalar> name { value }
<Char*> name { value }
```

Scalars can appear in various contexts. They are always optional, and specify some attribute value relevant to the current context. The scalar name is the name of the attribute; different attribute names are meaningful in different contexts. The value is either a numeric or a (quoted or unquoted) string value; the interpretation as a number or as a string depends on the nature of the named attribute. Because of a syntactic accident with the way the egg syntax evolved, <Scalar> and <Char*> are lexically the same and both can represent either a string or a number. <Char*> is being phased out; it is suggested that new egg files use only <Scalar>.

GLOBAL INFORMATION ENTRIES

These nodes contain information relevant to the file as a whole. They can be nested along with geometry nodes, but this nesting is irrelevant and the only significant placement rule is that they should appear before they are referenced.

```
<CoordinateSystem> { string }
```

This entry indicates the coordinate system used in the egg file; the egg loader will automatically make a conversion if necessary. The following strings are valid: Y-up, Z-up, Y-up-right, Z-up-right, Y-up-left, or Z-up-left. (Y-up is the same as Y-up-right, and Z-up is the same as Z-up-right.)

By convention, this entry should only appear at the beginning of the file, although it is technically allowed anywhere. It is an error

to include more than one coordinate system entry in the same file. If it is omitted, Y-up is assumed.

```
<Texture> name { filename [scalars] }
```

This describes a texture file that can be referenced later with <TRef> { name }. It is not necessary to make a <Texture> entry for each texture to be used; a texture may also be referenced directly by the geometry via an abbreviated inline <Texture> entry, but a separate <Texture> entry is the only way to specify anything other than the default texture attributes.

If the filename is a relative path, the current egg file's directory is searched first, and then the texture-path and model-path are searched.

The following attributes are presently implemented for textures:

```
<Scalar> alpha-file { alpha-filename }
```

If this scalar is present, the texture file's alpha channel is read in from the named image file (which should contain a grayscale image), and the two images are combined into a single two- or four-channel image internally. This is useful for loading alpha channels along with image file formats like JPEG that don't traditionally support alpha channels.

```
<Scalar> alpha-file-channel { channel }
```

This defines the channel that should be extracted from the file named by alpha-file to determine the alpha channel for the resulting channel. The default is 0, which means the grayscale combination of r, g, b. Otherwise, this should be the 1-based channel number, for instance 1, 2, or 3 for r, g, or b, respectively, or 4 for the alpha channel of a four-component image.

```
<Scalar> format { format-definition }
```

This defines the load format of the image file. The format-definition is one of:

RGBA, RGBM, RGBA12, RGBA8, RGBA4,
RGB, RGB12, RGB8, RGB5, RGB332,
LUMINANCE_ALPHA,
RED, GREEN, BLUE, ALPHA, LUMINANCE

The formats whose names end in digits specifically request a particular texel width. RGB12 and RGBA12 specify 48-bit texels with or without alpha; RGB8 and RGBA8 specify 32-bit texels, and RGB5 and RGBA4 specify 16-bit texels. RGB332 specifies 8-bit texels.

The remaining formats are generic and specify only the semantic meaning of the channels. The size of the texels is determined by the width of the components in the image file. RGBA is the most general; RGB is the same, but without any alpha channel. RGBM is like RGBA, except that it requests only one bit of alpha, if the graphics card can provide that, to leave more room for the RGB components, which is especially important for older 16-bit graphics cards (the "M" stands for "mask", as in a cutout).

The number of components of the image file should match the format specified; if it does not, the egg loader will attempt to provide the closest match that does.

`<Scalar> compression { compression-mode }`

Defines an explicit control over the real-time compression mode applied to the texture. The various options are:

DEFAULT OFF ON
FXT1 DXT1 DXT2 DXT3 DXT4 DXT5

This controls the compression of the texture when it is loaded into graphics memory, and has nothing to do with on-disk compression such as JPEG. If this option is omitted or "DEFAULT", then the texture compression is controlled by the compressed-textures config variable. If it is "OFF", texture compression is explicitly off for this texture regardless of the setting of the config variable; if it is "ON", texture compression is explicitly on, and a default compression algorithm supported by

the driver is selected. If any of the other options, it names the specific compression algorithm to be used.

```
<Scalar> wrap { repeat-definition }
<Scalar> wrapu { repeat-definition }
<Scalar> wrapv { repeat-definition }
<Scalar> wrapw { repeat-definition }
```

This defines the behavior of the texture image outside of the normal (u,v) range 0.0 - 1.0. It is "REPEAT" to repeat the texture to infinity, "CLAMP" not to. The wrapping behavior may be specified independently for each axis via "wrapu" and "wrapv", or it may be specified for both simultaneously via "wrap".

Although less often used, for 3-d textures wrapw may also be specified, and it behaves similarly to wrapu and wrapv.

There are other legal values in addtional to REPEAT and CLAMP. The full list is:

CLAMP
REPEAT
MIRROR
MIRROR_ONCE
BORDER_COLOR

```
<Scalar> borderr { red-value }
<Scalar> borderg { green-value }
<Scalar> borderb { blue-value }
<Scalar> bordera { alpha-value }
```

These define the "border color" of the texture, which is particularly important when one of the wrap modes, above, is BORDER_COLOR.

```
<Scalar> type { texture-type }
```

This may be one of the following attributes:

1D
2D

3D
CUBE_MAP

The default is "2D", which specifies a normal, 2-d texture. If any of the other types is specified instead, a texture image of the corresponding type is loaded.

If 3D or CUBE_MAP is specified, then a series of texture images must be loaded to make up the complete texture; in this case, the texture filename is expected to include a sequence of one or more hash mark ("#") characters, which will be filled in with the sequence number. The first image in the sequence must be numbered 0, and there must be no gaps in the sequence. In this case, a separate alpha-file designation is ignored; the alpha channel, if present, must be included in the same image with the color channel(s).

<Scalar> multiview { flag }

If this flag is nonzero, the texture is loaded as a multiview texture. In this case, the filename must contain a hash mark ("#") as in the 3D or CUBE_MAP case, above, and the different images are loaded into the different views of the multiview textures. If the texture is already a cube map texture, the same hash sequence is used for both purposes: the first six images define the first view, the next six images define the second view, and so on. If the texture is a 3-D texture, you must also specify num-views, below, to tell the loader how many images are loaded for views, and how many are loaded for levels.

A multiview texture is most often used to load stereo textures, where a different image is presented to each eye viewing the texture, but other uses are possible, such as for texture animation.

<Scalar> num-views { count }

This is used only when loading a 3-D multiview texture. It specifies how many different views the texture holds; the z height of the texture is then implicitly determined as (number of images) / (number of views).

<Scalar> read-mipmaps { flag }

If this flag is nonzero, then pre-generated mipmap levels will be loaded along with the texture. In this case, the filename should contain a sequence of one or more hash mark ("#") characters, which will be filled in with the mipmap level number; the texture filename thus determines a series of images, one for each mipmap level. The base texture image is mipmap level 0.

If this flag is specified in conjunction with a 3D or cube map texture (as specified above), then the filename should contain two hash mark sequences, separated by a character such as an underscore, hyphen, or dot. The first sequence will be filled in with the mipmap level index, and the second sequence will be filled in with the 3D sequence or cube map face.

<Scalar> minfilter { filter-type }
<Scalar> magfilter { filter-type }
<Scalar> magfilteralpha { filter-type }
<Scalar> magfiltercolor { filter-type }

This specifies the type of filter applied when minimizing or maximizing. Filter-type may be one of:

NEAREST
LINEAR
NEAREST_MIPMAP_NEAREST
LINEAR_MIPMAP_NEAREST
NEAREST_MIPMAP_LINEAR
LINEAR_MIPMAP_LINEAR

There are also some additional filter types that are supported for historical reasons, but each of those additional types maps to one of the above. New egg files should use only the above filter types.

<Scalar> anisotropic-degree { degree }

Enables anisotropic filtering for the texture, and specifies the degree of filtering. If the degree is 0 or 1, anisotropic filtering is disabled. The default is disabled.

<Scalar> envtype { environment-type }

This specifies the type of texture environment to create; i.e. it controls the way in which textures apply to models.
Environment-type may be one of:

MODULATE
DECAL
BLEND
REPLACE
ADD
BLEND_COLOR_SCALE
MODULATE_GLOW
MODULATE_GLOSS
*NORMAL
*NORMAL_HEIGHT
*GLOW
*GLOSS
*HEIGHT
*SELECTOR

The default environment type is MODULATE, which means the texture color is multiplied with the base polygon (or vertex) color. This is the most common texture environment by far. Other environment types are more esoteric and are especially useful in the presence of multitexture. In particular, the types prefixed by an asterisk (*) require enabling Panda's automatic ShaderGenerator.

```
<Scalar> combine-rgb { combine-mode }
<Scalar> combine-alpha { combine-mode }
<Scalar> combine-rgb-source0 { combine-source }
<Scalar> combine-rgb-operand0 { combine-operand }
<Scalar> combine-rgb-source1 { combine-source }
<Scalar> combine-rgb-operand1 { combine-operand }
<Scalar> combine-rgb-source2 { combine-source }
<Scalar> combine-rgb-operand2 { combine-operand }
<Scalar> combine-alpha-source0 { combine-source }
<Scalar> combine-alpha-operand0 { combine-operand }
<Scalar> combine-alpha-source1 { combine-source }
```

```
<Scalar> combine-alpha-operand1 { combine-operand }
<Scalar> combine-alpha-source2 { combine-source }
<Scalar> combine-alpha-operand2 { combine-operand }
```

These options replace the envtype and specify the texture combiner mode, which is usually used for multitexturing. This specifies how the texture combines with the base color and/or the other textures applied previously. You must specify both an rgb and an alpha combine mode. Some combine-modes use one source/operand pair, and some use all three; most use just two.

combine-mode may be one of:

REPLACE
MODULATE
ADD
ADD-SIGNED
INTERPOLATE
SUBTRACT
DOT3-RGB
DOT3-RGBA

combine-source may be one of:

TEXTURE
CONSTANT
PRIMARY-COLOR
PREVIOUS
CONSTANT_COLOR_SCALE
LAST_SAVED_RESULT

combine-operand may be one of:

SRC-COLOR
ONE-MINUS-SRC-COLOR
SRC-ALPHA
ONE-MINUS-SRC-ALPHA

The default values if any of these are omitted are:

```
<Scalar> combine-rgb { modulate }
<Scalar> combine-alpha { modulate }
<Scalar> combine-rgb-source0 { previous }
```

```
    <Scalar> combine-rgb-operand0 { src-color }
    <Scalar> combine-rgb-source1 { texture }
    <Scalar> combine-rgb-operand1 { src-color }
    <Scalar> combine-rgb-source2 { constant }
    <Scalar> combine-rgb-operand2 { src-alpha }
    <Scalar> combine-alpha-source0 { previous }
    <Scalar> combine-alpha-operand0 { src-alpha }
    <Scalar> combine-alpha-source1 { texture }
    <Scalar> combine-alpha-operand1 { src-alpha }
    <Scalar> combine-alpha-source2 { constant }
    <Scalar> combine-alpha-operand2 { src-alpha }

  <Scalar> saved-result { flag }
```

If flag is nonzero, then it indicates that this particular texture stage will be supplied as the "last_saved_result" source for any future texture stages.

```
  <Scalar> tex-gen { mode }
```

This specifies that texture coordinates for the primitives that reference this texture should be dynamically computed at runtime, for instance to apply a reflection map or some other effect. The valid values for mode are:

EYE_SPHERE_MAP (or SPHERE_MAP)
WORLD_CUBE_MAP
EYE_CUBE_MAP (or CUBE_MAP)
WORLD_NORMAL
EYE_NORMAL
WORLD_POSITION
EYE_POSITION
POINT_SPRITE

<Scalar> stage-name { name }

Specifies the name of the TextureStage object that is created to render this texture. If this is omitted, a custom TextureStage is created for this texture if it is required (e.g. because some other multitexturing parameter has been specified), or the system

default TextureStage is used if multitexturing is not required.

<Scalar> priority { priority-value }

Specifies an integer sort value to rank this texture in priority among other textures that are applied to the same geometry. This is only used to eliminate low-priority textures in case more textures are requested for a particular piece of geometry than the graphics hardware can render.

<Scalar> blendr { red-value }
<Scalar> blendg { green-value }
<Scalar> blendb { blue-value }
<Scalar> blenda { alpha-value }

Specifies a four-component color that is applied with the color in case the envtype, above, is "blend", or one of the combine-sources is "constant".

<Scalar> uv-name { name }

Specifies the name of the texture coordinates that are to be associated with this texture. If this is omitted, the default texture coordinates are used.

<Scalar> rgb-scale { scale }
<Scalar> alpha-scale { scale }

Specifies an additional scale factor that will scale the r, g, b (or a) components after the texture has been applied. This is only used when a combine mode is in effect. The only legal values are 1, 2, or 4.

<Scalar> alpha { alpha-type }

This specifies whether and what type of transparency will be performed. Alpha-type may be one of:

OFF
ON
BLEND

BLEND_NO_OCCLUDE
MS
MS_MASK
BINARY
DUAL

If alpha-type is OFF, it means not to enable transparency, even if the image contains an alpha channel or the format is RGBA. If alpha-type is ON, it means to enable the default transparency, even if the image filename does not contain an alpha channel. If alpha-type is any of the other options, it specifies the type of transparency to be enabled.

<Scalar> bin { bin-name }

This specifies the bin name order of all polygons with this texture applied, in the absence of a bin name specified on the polygon itself. See the description for bin under polygon attributes.

<Scalar> draw-order { number }

This specifies the fixed drawing order of all polygons with this texture applied, in the absence of a drawing order specified on the polygon itself. See the description for draw-order under polygon attributes.

<Scalar> depth-offset { number }
<Scalar> depth-write { mode }
<Scalar> depth-test { mode }

Specifies special depth buffer properties of all polygons with this texture applied. See the descriptions for the individual attributes under polygon attributes.

<Scalar> quality-level { quality }

Sets a hint to the renderer about the desired performance / quality tradeoff for this particular texture. This is most useful for the tinydisplay software renderer; for normal, hardware-accelerated renderers, this may have little or no effect.

This may be one of:

DEFAULT
FASTEST
NORMAL
BEST

"Default" means to use whatever quality level is specified by the global texture-quality-level config variable.

<Transform> { transform-definition }

This specifies a 2-d or 3-d transformation that is applied to the UV's of a surface to generate the texture coordinates.

The transform syntax is similar to that for groups, except it may define either a 2-d 3x3 matrix or a 3-d 4x4 matrix. (You should use the two-dimensional forms if the UV's are two-dimensional, and the three-dimensional forms if the UV's are three-dimensional.)

A two-dimensional transform may be any sequence of zero or more of the following. Transformations are post multiplied in the order they are encountered to produce a net transformation matrix. Rotations are counterclockwise about the origin in degrees. Matrices, when specified explicitly, are row-major.

<Translate> { x y }
<Rotate> { degrees }
<Scale> { x y }
<Scale> { s }

<Matrix3> {
00 01 02
10 11 12
20 21 22
}

A three-dimensional transform may be any sequence of zero or more of the following. See the description under <Group>, below, for more information.

```
<Translate> { x y z }
<RotX> { degrees }
<RotY> { degrees }
<RotZ> { degrees }
<Rotate> { degrees x y z }
<Scale> { x y z }
<Scale> { s }

<Matrix4> {
  00 01 02 03
  10 11 12 13
  20 21 22 23
  30 31 32 33
}
```

<Material> name { [scalars] }

This defines a set of material attributes that may later be referenced with <MRef> { name }.

The following attributes may appear within the material block:

<Scalar> diffr { number }
<Scalar> diffg { number }
<Scalar> diffb { number }
<Scalar> diffa { number }

<Scalar> ambr { number }
<Scalar> ambg { number }
<Scalar> ambb { number }
<Scalar> amba { number }

<Scalar> emitr { number }
<Scalar> emitg { number }
<Scalar> emitb { number }
<Scalar> emita { number }

<Scalar> specr { number }
<Scalar> specg { number }

<Scalar> specb { number }
<Scalar> speca { number }

<Scalar> shininess { number }
<Scalar> local { flag }

These properties collectively define a "material" that controls the lighting effects that are applied to a surface; a material is only in effect in the presence of lighting.

The four color groups, diff*, amb*, emit*, and spec* specify the diffuse, ambient, emission, and specular components of the lighting equation, respectively. Any of them may be omitted; the omitted component(s) take their color from the native color of the primitive, otherwise the primitive color is replaced with the material color.

The shininess property controls the size of the specular highlight, and the value ranges from 0 to 128. A larger value creates a smaller highlight (creating the appearance of a shinier surface).

<VertexPool> name { vertices }

A vertex pool is a set of vertices. All geometry is created by referring to vertices by number in a particular vertex pool. There may be one or several vertex pools in an egg file, but all vertices that make up a single polygon must come from the same vertex pool. The body of a <VertexPool> entry is simply a list of one or more <Vertex> entries, as follows:

<Vertex> number { x [y [z [w]]] [attributes] }

A <Vertex> entry is only valid within a vertex pool definition. The number is the index by which this vertex will be referenced. It is optional; if it is omitted, the vertices are implicitly numbered consecutively beginning at one. If the number is supplied, the vertices need not be consecutive.

Normally, vertices are three-dimensional (with coordinates x, y, and z); however, in certain cases vertices may have fewer or more

dimensions, up to four. This is particularly true of vertices used as control vertices of NURBS curves and surfaces. If more coordinates are supplied than needed, the extra coordinates are ignored; if fewer are supplied than needed, the missing coordinates are assumed to be 0.

The vertex's coordinates are always given in world space, regardless of any transforms before the vertex pool or before the referencing geometry. If the vertex is referenced by geometry under a transform, the egg loader will do an inverse transform to move the vertex into the proper coordinate space without changing its position in world space. One exception is geometry under an <Instance> node; in this case the vertex coordinates are given in the space of the <Instance> node. (Another exception is a <DynamicVertexPool>; see below.)

In neither case does it make a difference whether the vertex pool is itself declared under a transform or an <Instance> node. The only deciding factor is whether the geometry that *uses* the vertex pool appears under an <Instance> node. It is possible for a single vertex to be interpreted in different coordinate spaces by different polygons.

While each vertex must at least have a position, it may also have a color, normal, pair of UV coordinates, and/or a set of morph offsets. Furthermore, the color, normal, and UV coordinates may themselves have morph offsets. Thus, the [attributes] in the syntax line above may be replaced with zero or more of the following entries:

<Dxyz> target { x y z }

This specifies the offset of this vertex for the named morph target. See the "MORPH DESCRIPTION ENTRIES" header, below.

<Normal> { x y z [morph-list] }

This specifies the surface normal of the vertex. If omitted, the

vertex will have no normal. Normals may also be morphed; morph-list here is thus an optional list of <DNormal> entries, similar to the above.

<RGBA> { r g b a [morph-list] }

This specifies the four-valued color of the vertex. Each component is in the range 0.0 to 1.0. A vertex color, if specified for all vertices of the polygon, overrides the polygon's color. If neither color is given, the default is white (1 1 1 1). The morph-list is an optional list of <DRGBA> entries.

<UV> [name] { u v [w] [tangent] [binormal] [morph-list] }

This gives the texture coordinates of the vertex. This must be specified if a texture is to be mapped onto this geometry.

The texture coordinates are usually two-dimensional, with two component values (u v), but they may also be three-dimensional, with three component values (u v w). (Arguably, it should be called <UVW> instead of <UV> in the three-dimensional case, but it's not.)

As before, morph-list is an optional list of <DUV> entries.

Unlike the other kinds of attributes, there may be multiple sets of UV's on each vertex, each with a unique name; this provides support for multitexturing. The name may be omitted to specify the default UV's.

The UV's also support an optional tangent and binormal. These values are based on the vertex normal and the UV coordinates of connected vertices, and are used to render normal maps and similar lighting effects. They are defined within the <UV> entry because there may be a different set of tangents and binormals for each different UV coordinate set. If present, they have the expected syntax:

<UV> [name] { u v [w] <Tangent> { x y z } <Binormal> { x y z } }

<AUX> name { x y z w }

This specifies some named per-vertex auxiliary data which is imported from the egg file without further interpretation by Panda. The auxiliary data is copied to the vertex data under a column with the specified name. Presumably the data will have meaning to custom code or a custom shader. Like named UV's, there may be multiple Aux entries for a given vertex, each with a different name.

<DynamicVertexPool> name { vertices }

A dynamic vertex pool is similar to a vertex pool in most respects, except that each vertex might be animated by substituting in values from a <VertexAnim> table. Also, the vertices defined within a dynamic vertex pool are always given in local coordinates, instead of world coordinates.

The presence of a dynamic vertex pool makes sense only within a character model, and a single dynamic vertex pool may not span multiple characters. Each dynamic vertex pool creates a DynVerts object within the character by the same name; this name is used later when matching up the corresponding <VertexAnim>.

At the present time, the DynamicVertexPool is not implemented in Panda3D.

GEOMETRY ENTRIES

```
<Polygon> name {
   [attributes]
   <VertexRef> {
      indices
      <Ref> { pool-name }
   }
}
```

A polygon consists of a sequence of vertices from a single vertex pool. Vertices are identified by pool-name and index number within the pool; indices is a list of vertex numbers within the given vertex pool. Vertices are listed in counterclockwise order. Although the vertices must all come from the same vertex pool, they may have been assigned to arbitrarily many different joints regardless of joint connectivity (there is no "straddle-polygon" limitation). See Joints, below.

The polygon syntax is quite verbose, and there isn't any way to specify a set of attributes that applies to a group of polygons--the attributes list must be repeated for each polygon. This is why egg files tend to be very large.

The following attributes may be specified for polygons:

<TRef> { texture-name }

This refers to a named <Texture> entry given earlier. It applies the given texture to the polygon. This requires that all the polygon's vertices have been assigned texture coordinates.

This attribute may be repeated multiple times to specify multitexture. In this case, each named texture is applied to the polygon, in the order specified.

<Texture> { filename }

This is another way to apply a texture to a polygon. The <Texture> entry is defined "inline" to the polygon, instead of referring to a <Texture> entry given earlier. There is no way to specify texture attributes given this form.

There's no advantage to this syntax for texture mapping. It's supported only because it's required by some older egg files.

<MRef> { material-name }

This applies the material properties defined in the earlier

<Material> entry to the polygon.

<Normal> { x y z [morph-list] }

This defines a polygon surface normal. The polygon normal will be used unless all vertices also have a normal. If no normal is defined, none will be supplied. The polygon normal, like the vertex normal, may be morphed by specifying a series of <DNormal> entries.

The polygon normal is used only for lighting and environment mapping calculations, and is not related to the implicit normal calculated for CollisionPolygons.

<RGBA> { r g b a [morph-list] }

This defines the polygon's color, which will be used unless all vertices also have a color. If no color is defined, the default is white (1 1 1 1). The color may be morphed with a series of <DRGBA> entries.

<BFace> { boolean-value }

This defines whether the polygon will be rendered double-sided (i.e. its back face will be visible). By default, this option is disabled, and polygons are one-sided; specifying a nonzero value disables backface culling for this particular polygon and allows it to be viewed from either side.

<Scalar> bin { bin-name }

It is sometimes important to control the order in which objects are rendered, particularly when transparency is in use. In Panda, this is achieved via the use of named bins and, within certain kinds of bins, sometimes an explicit draw-order is also used (see below).

In the normal (state-sorting) mode, Panda renders its geometry by first grouping into one or more named bins, and then rendering the bins in a specified order. The programmer is free to define any number of bins, named whatever he/she desires.

This scalar specifies which bin this particular polygon is to be rendered within. If no bin scalar is given, or if the name given does not match any of the known bins, the polygon will be assigned to the default bin, which renders all opaque geometry sorted by state, followed by all transparent geometry sorted back-to-front.

See also draw-order, below.

<Scalar> draw-order { number }

This works in conjunction with bin, above, to further refine the order in which this polygon is drawn, relative to other geometry in the same bin. If (and only if) the bin type named in the bin scalar is a CullBinFixed, this draw-order is used to define the fixed order that all geometry in the same will be rendered, from smaller numbers to larger numbers.

If the draw-order scalar is specified but no bin scalar is specified, the default is a bin named "fixed", which is a CullBinFixed object that always exists by default.

<Scalar> depth-offset { number }

Specifies a special depth offset to be applied to the polygon. This must be an integer value between 0 and 16 or so. The default value is 0; values larger than 0 will cause the polygon to appear closer to the camera for purposes of evaluating the depth buffer. This can be a simple way to resolve Z-fighting between coplanar polygons: with two or more coplanar polygons, the polygon with the highest depth-offset value will appear to be visible on top. Note that this effect doesn't necessarily work well when the polygons are viewed from a steep angle.

<Scalar> depth-write { mode }

Specifies the mode for writing to the depth buffer. This may be ON or OFF. The default is ON.

<Scalar> depth-test { mode }

Specifies the mode for testing against the depth buffer. This may be ON or OFF. The default is ON.

<Scalar> visibility { hidden | normal }

If the visibility of a primitive is set to "hidden", the primitive is not generated as a normally visible primitive. If the Config.prc variable egg-suppress-hidden is set to true, the primitive is not converted at all; otherwise, it is converted as a "stashed" node.

This, like the other rendering flags alpha, draw-order, and bin, may be specified at the group level, within the primitive level, or even within a texture.

```
<Patch> name {
  [attributes]
  <VertexRef> {
    indices
    <Ref> { pool-name }
  }
}
```

A patch is similar to a polygon, but it is a special primitive that can only be rendered with the use of a tessellation shader. Each patch consists of an arbitrary number of vertices; all patches with the same number of vertices are collected together into the same GeomPatches object to be delivered to the shader in a single batch. It is then up to the shader to create the correct set of triangles from the patch data.

All of the attributes that are valid for Polygon, above, may also be specified for Patch.

```
<PointLight> name {
   [attributes]
   <VertexRef> {
      indices
      <Ref> { pool-name }
   }
}
```

A PointLight is a set of single points. One point is drawn for each vertex listed in the <VertexRef>. Normals, textures, and colors may be specified for PointLights, as well as draw-order, plus one additional attribute valid only for PointLights and Lines:

<Scalar> thick { number }

This specifies the size of the PointLight (or the width of a line), in pixels, when it is rendered. This may be a floating-point number, but the fractional part is meaningful only when antialiasing is in effect. The default is 1.0.

<Scalar> perspective { boolean-value }

If this is specified, then the thickness, above, is to interpreted as a size in 3-d spatial units, rather than a size in pixels, and the point should be scaled according to its distance from the viewer normally.

```
<Line> name {
   [attributes]
   <VertexRef> {
      indices
      <Ref> { pool-name }
   }
   [component attributes]
}
```

A Line is a connected set of line segments. The listed N vertices define a series of N-1 line segments, drawn between vertex 0 and

vertex 1, vertex 1 and vertex 2, etc. The line is not implicitly closed; if you wish to represent a loop, you must repeat vertex 0 at the end. As with a PointLight, normals, textures, colors, draw-order, and the "thick" attribute are all valid (but not "perspective"). Also, since a Line (with more than two vertices) is made up of multiple line segments, it may contain a number of <Component> entries, to set a different color and/or normal for each line segment, as in TriangleStrip, below.

```
<TriangleStrip> name {
  [attributes]
  <VertexRef> {
    indices
    <Ref> { pool-name }
  }
  [component attributes]
}
```

A triangle strip is only rarely encountered in an egg file; it is normally generated automatically only during load time, when connected triangles are automatically meshed for loading, and even then it exists only momentarily. Since a triangle strip is a rendering optimization only and adds no useful scene information over a loose collection of triangles, its usage is contrary to the general egg philosophy of representing a scene in the abstract. Nevertheless, the syntax exists, primarily to allow inspection of the meshing results when needed. You can also add custom TriangleStrip entries to force a particular mesh arrangement.

A triangle strip is defined as a series of connected triangles. After the first three vertices, which define the first triangle, each new vertex defines one additional triangle, by alternating up and down.

It is possible for the individual triangles of a triangle strip to have a separate normal and/or color. If so, a <Component> entry should be given for each so-modified triangle:

```
<Component> index {
  <RGBA> { r g b a [morph-list] }
```

```
  <Normal> { x y z [morph-list] }
 }
```

Where index ranges from 0 to the number of components defined by the triangle strip (less 1). Note that the component attribute list must always follow the vertex list.

```
<TriangleFan> name {
  [attributes]
  <VertexRef> {
    indices
    <Ref> { pool-name }
  }
  [component attributes]
}
```

A triangle fan is similar to a triangle strip, except all of the connected triangles share the same vertex, which is the first vertex. See <TriangleStrip>, above.

PARAMETRIC DESCRIPTION ENTRIES

The following entries define parametric curves and surfaces. Generally, Panda supports these only in the abstract; they're not geometry in the true sense but do exist in the scene graph and may have specific meaning to the application. However, Panda can create visible representations of these parametrics to aid visualization.

These entries might also have meaning to external tools outside of an interactive Panda session, such as egg-qtess, which can be used to convert NURBS surfaces to polygons at different levels of resolution.

In general, dynamic attributes such as morphs and joint assignment are legal for the control vertices of the following parametrics, but Panda itself doesn't support them and will always create static curves and surfaces. External tools like egg-qtess, however, may respect them.

```
<NURBSCurve> {
  [attributes]

  <Order> { order }
  <Knots> { knot-list }
  <VertexRef> { indices <Ref> { pool-name } }
}
```

A NURBS curve is a general parametric curve. It is often used to represent a motion path, e.g. for a camera or an object.

The order is equal to the degree of the polynomial basis plus 1. It must be an integer in the range [1,4].

The number of vertices must be equal to the number of knots minus the order.

Each control vertex of a NURBS is defined in homogeneous space with four coordinates x y z w (to convert to 3-space, divide x, y, and z by w). The last coordinate is always the homogeneous coordinate; if only three coordinates are given, it specifies a curve in two dimensions plus a homogeneous coordinate (x y w).

The following attributes may be defined:

```
<Scalar> type { curve-type }
```

This defines the semanting meaning of this curve, either XYZ, HPR, or T. If the type is XYZ, the curve will automatically be transformed between Y-up and Z-up if necessary; otherwise, it will be left alone.

```
<Scalar> subdiv { num-segments }
```

If this scalar is given and nonzero, Panda will create a visible representation of the curve when the scene is loaded. The number represents the number of line segments to draw to approximate the curve.

```
<RGBA> { r g b a [morph-list] }
```

This specifies the color of the overall curve.

NURBS control vertices may also be given color and/or morph attributes, but <Normal> and <UV> entries do not apply to NURBS vertices.

```
<NURBSSurface> name {
  [attributes]

  <Order> { u-order v-order }
  <U-knots> { u-knot-list }
  <V-knots> { v-knot-list }

  <VertexRef> {
    indices
    <Ref> { pool-name }
  }
}
```

A NURBS surface is an extension of a NURBS curve into two parametric dimensions, u and v. NURBS surfaces may be given the same set of attributes assigned to polygons, except for normals: <TRef>,

<Texture>, <MRef>, <RGBA>, and draw-order are all valid attributes for NURBS. NURBS vertices, similarly, may be colored or morphed, but <Normal> and <UV> entries do not apply to NURBS vertices. The attributes may also include <NURBSCurve> and <Trim> entries; see below.

To have Panda create a visualization of a NURBS surface, the following two attributes should be defined as well:

```
<Scalar> U-subdiv { u-num-segments }
<Scalar> V-subdiv { v-num-segments }
```

These define the number of subdivisions to make in the U and V directions to represent the surface. A uniform subdivision is always made, and trim curves are not respected (though they will be drawn in if the trim curves themselves also have a subiv parameter). This is only intended as a cheesy visualization.

The same sort of restrictions on order and knots applies to NURBS surfaces as do to NURBS curves. The order and knot description may be different in each dimension.

The surface must have u-num * v-num vertices, where u-num is the number of u-knots minus the u-order, and v-num is the number of v-knots minus the v-order. All vertices must come from the same vertex pool. The nth (zero-based) index number defines control vertex (u, v) of the surface, where n = (v * u-num) + u. Thus, it is the u coordinate which changes faster.

As with the NURBS curve, each control vertex is defined in homogeneous space with four coordinates x y z w.

A NURBS may also contain curves on its surface. These are one or more nested <NURBSCurve> entries included with the attributes; these curves are defined in the two-dimensional parametric space of the surface. Thus, these curve vertices should have only two dimensions plus the homogeneous coordinate: u v w. A curve-on-surface has no intrinsic meaning to the surface, unless it is defined within a <Trim> entry, below.

Finally, a NURBS may be trimmed by one or more trim curves. These are special curves on the surface which exclude certain areas from the NURBS surface definition. The inside is specified using two rules: an odd winding rule that states that the inside consists of all regions for which an infinite ray from any point in the region will intersect the trim curve an odd number of times, and a curve orientation rule that states that the inside consists of the regions to the left as the curve is traced.

Each trim curve contains one or more loops, and each loop contains one or more NURBS curves. The curves of a loop connect in a head-to-tail fashion and must be explicitly closed.

The trim curve syntax is as follows:

```
<Trim> {
 <Loop> {
  <NURBSCurve> {
   <Order> { order }
     <Knots> { knot-list }

     <VertexRef> { indices <Ref> { pool-name } }
  }
  [ <NURBSCurve> { ... } ... ]
 }
 [ <Loop> { ... } ... ]
}
```

Although the egg syntax supports trim curves, there are at present no egg processing tools that respect them. For instance, egg-qtess ignores trim curves and always tesselates the entire NURBS surface.

MORPH DESCRIPTION ENTRIES

Morphs are linear interpolations of attribute values at run time, according to values read from an animation table. In general, vertex positions, surface normals, texture coordinates, and colors may be morphed.

A morph target is defined by giving a net morph offset for a series of

vertex or polygon attributes; this offset is the value that will be added to the attribute when the morph target has the value 1.0. At run time, the morph target's value may be animated to any scalar value (but generally between 0.0 and 1.0); the corresponding fraction of the offset is added to the attribute each frame.

There is no explicit morph target definition; a morph target exists solely as the set of all offsets that share the same target name. The target name may be any arbitrary string; like any name in an egg file, it should be quoted if it contains special characters.

The following types of morph offsets may be defined, within their corresponding attribute entries:

`<Dxyz> target { x y z }`

A position delta, valid within a <Vertex> entry or a <CV> entry. The given offset vector, scaled by the morph target's value, is added to the vertex or CV position each frame.

`<DNormal> target { x y z }`

A normal delta, similar to the position delta, valid within a <Normal> entry (for vertex or polygon normals). The given offset vector, scaled by the morph target's value, is added to the normal vector each frame. The resulting vector may not be automatically normalized to unit length.

`<DUV> target { u v [w] }`

A texture-coordinate delta, valid within a <UV> entry (within a <Vertex> entry). The offset vector should be 2-valued if the enclosing UV is 2-valued, or 3-valued if the enclosing UV is 3-valued. The given offset vector, scaled by the morph target's value, is added to the vertex's texture coordinates each frame.

`<DRGBA> target { r g b a }`

A color delta, valid within an <RGBA> entry (for vertex or polygon colors). The given 4-valued offset vector, scaled by the morph target's value, is added to the color value each frame.

GROUPING ENTRIES

```
<Group> name { group-body }
```

A <Group> node is the primary means of providing structure to the egg file. Groups can contain vertex pools and polygons, as well as other groups. The egg loader translates <Group> nodes directly into PandaNodes in the scene graph (although the egg loader reserves the right to arbitrarily remove nodes that it deems unimportant--see the <Model> flag, below to avoid this). In addition, the following entries can be given specifically within a <Group> node to specify attributes of the group:

GROUP BINARY ATTRIBUTES

These attributes may be either on or off; they are off by default. They are turned on by specifying a non-zero "boolean-value".

```
<DCS> { boolean-value }
```

DCS stands for Dynamic Coordinate System. This indicates that show code will expect to be able to read the transform set on this node at run time, and may need to modify the transform further. This is a special case of <Model>, below.

```
<DCS> { dcs-type }
```

This is another syntax for the <DCS> flag. The dcs-type string should be one of either "local" or "net", which specifies the kind of preserve_transform flag that will be set on the corresponding ModelNode. If the string is "local", it indicates that the local transform on this node (as well as the net transform) will not be affected by any flattening operation and will be preserved through the entire model loading process. If the string is "net", then only the net transform will be preserved; the local transform may be adjusted in the event of a flatten operation.

```
<Model> { boolean-value }
```

This indicates that the show code might need a pointer to this particular group. This creates a ModelNode at the corresponding

level, which is guaranteed not to be removed by any flatten operation. However, its transform might still be changed, but see also the <DCS> flag, above.

```
<Dart> { boolean-value }
```

This indicates that this group begins an animated character. A Character node, which is the fundamental animatable object of Panda's high-level Actor class, will be created for this group.

This flag should always be present within the <Group> entry at the top of any hierarchy of <Joint>'s and/or geometry with morphed vertices; joints and morphs appearing outside of a hierarchy identified with a <Dart> flag are undefined.

```
<Switch> { boolean-value }
```

This attribute indicates that the child nodes of this group represent a series of animation frames that should be consecutively displayed. In the absence of an "fps" scalar for the group (see below), the egg loader creates a SwitchNode, and it the responsibility of the show code to perform the switching. If an fps scalar is defined and is nonzero, the egg loader creates a SequenceNode instead, which automatically cycles through its children.

GROUP SCALARS

```
<Scalar> fps { frame-rate }
```

This specifies the rate of animation for a SequenceNode (created when the Switch flag is specified, see above). A value of zero indicates a SwitchNode should be created instead.

```
<Scalar> bin { bin-name }
```

This specifies the bin name for all polygons at or below this node that do not explicitly set their own bin. See the description of bin for geometry attributes, above.

```
<Scalar> draw-order { number }
```

This specifies the drawing order for all polygons at or below this node that do not explicitly set their own drawing order. See the description of draw-order for geometry attributes, above.

```
<Scalar> depth-offset { number }
<Scalar> depth-write { mode }
<Scalar> depth-test { mode }
```

Specifies special depth buffer properties of all polygons at or below this node that do not override this. See the descriptions for the individual attributes under polygon attributes.

```
<Scalar> visibility { hidden | normal }
```

If the visibility of a group is set to "hidden", the primitives nested within that group are not generated as a normally visible primitive. If the Config.prc variable egg-suppress-hidden is set to true, the primitives are not converted at all; otherwise, they are converted as a "stashed" node.

```
<Scalar> decal { boolean-value }
```

If this is present and boolean-value is non-zero, it indicates that the geometry *below* this level is coplanar with the geometry *at* this level, and the geometry below is to be drawn as a decal onto the geometry at this level. This means the geometry below this level will be rendered "on top of" this geometry, but without the Z-fighting artifacts one might expect without the use of the decal flag.

```
<Scalar> decalbase { boolean-value }
```

This can optionally be used with the "decal" scalar, above. If present, it should be applied to a sibling of one or more nodes with the "decal" scalar on. It indicates which of the sibling nodes should be treated as the base of the decal. In the absence of this scalar, the parent of all decal nodes is used as the decal base. This scalar is useful when the modeling package is unable to parent geometry nodes to other geometry nodes.

```
<Scalar> collide-mask { value }
<Scalar> from-collide-mask { value }
<Scalar> into-collide-mask { value }
```

Sets the CollideMasks on the collision nodes and geometry nodes created at or below this group to the indicated values. These are bits that indicate which objects can collide with which other objects. Setting "collide-mask" is equivalent to setting both "from-collide-mask" and "into-collide-mask" to the same value.

The value may be an ordinary decimal integer, or a hex number in the form 0x000, or a binary number in the form 0b000.

```
<Scalar> blend { mode }
```

Specifies that a special blend mode should be applied geometry at this level and below. The available options are none, add, subtract, inv-subtract, min, and max. See ColorBlendAttrib.

```
<Scalar> blendop-a { mode }
<Scalar> blendop-b { mode }
```

If blend mode, above, is not none, this specifies the A and B operands to the blend equation. Common options are zero, one, incoming-color, one-minus-incoming-color. See ColorBlendAttrib for the complete list of available options. The default is "one".

```
<Scalar> blendr { red-value }
<Scalar> blendg { green-value }
<Scalar> blendb { blue-value }
<Scalar> blenda { alpha-value }
```

If blend mode, above, is not none, and one of the blend operands is constant-color or a related option, this defines the constant color that will be used.

```
<Scalar> occluder { boolean-value }
```

This makes the first (or only) polygon within this group node into

an occluder. The polygon must have exactly four vertices. An occluder polygon is invisible. When the occluder is activated with model.set_occluder(occluder), objects that are behind the occluder will not be drawn. This can be a useful rendering optimization for complex scenes, but should not be overused or performance can suffer.

OTHER GROUP ATTRIBUTES

```
<Billboard> { type }
```

This entry indicates that all geometry defined at or below this group level is part of a billboard that will rotate to face the camera. Type is either "axis" or "point", describing the type of rotation.

Billboards rotate about their local axis. In the case of a Y-up file, the billboards rotate about the Y axis; in a Z-up file, they rotate about the Z axis. Point-rotation billboards rotate about the origin.

There is an implicit <Instance> around billboard geometry. This means that the geometry within a billboard is not specified in world coordinates, but in the local billboard space. Thus, a vertex drawn at point 0,0,0 will appear to be at the pivot point of the billboard, not at the origin of the scene.

```
<SwitchCondition> {
  <Distance> {
    in out [fade] <Vertex> { x y z }
  }
}
```

The subtree beginning at this node and below represents a single level of detail for a particular model. Sibling nodes represent the additional levels of detail. The geometry at this node will be visible when the point (x, y, z) is closer than "in" units, but further than "out" units, from the camera. "fade" is presently ignored.

```
<Tag> key { value }
```

This attribute defines the indicated tag (as a key/value pair), retrievable via NodePath::get_tag() and related interfaces, on this node.

<Collide> name { type [flags] }

This entry indicates that geometry defined at this group level is actually an invisible collision surface, and is not true geometry. The geometry is used to define the extents of the collision surface. If there is no geometry defined at this level, then a child is searched for with the same collision type specified, and its geometry is used to define the extent of the collision surface (unless the "descend" flag is given; see below).

Valid types so far are:

Plane

The geometry represents an infinite plane. The first polygon found in the group will define the plane.

Polygon

The geometry represents a single polygon. The first polygon is used.

Polyset

The geometry represents a complex shape made up of several polygons. This collision type should not be overused, as it provides the least optimization benefit.

Sphere

The geometry represents a sphere. The vertices in the group are averaged together to determine the sphere's center and radius.

Box

The geometry represents a box. The smalles axis-alligned box that will fit around the vertices is used.

InvSphere

The geometry represents an inverse sphere. This is the same as Sphere, with the normal inverted, so that the solid part of an inverse sphere is the entire world outside of it. Note that an inverse sphere is in infinitely large solid with a finite hole cut into it.

Tube

The geometry represents a tube. This is a cylinder-like shape with hemispherical endcaps; it is sometimes called a capsule or a lozenge in other packages. The smallest tube shape that will fit around the vertices is used.

The flags may be any zero or more of:

event

Throws the name of the <Collide> entry, or the name of the surface if the <Collide> entry has no name, as an event whenever an avatar strikes the solid. This is the default if the <Collide> entry has a name.

intangible

Rather than being a solid collision surface, the defined surface represents a boundary. The name of the surface will be thrown as an event when an avatar crosses into the interior, and name-out will be thrown when an avater exits.

descend

Instead of creating only one collision object of the given type,

each group descended from this node that contains geometry will define a new collision object of the given type. The event name, if any, will also be inherited from the top node and shared among all the collision objects.

keep

Don't discard the visible geometry after using it to define a collision surface; create both an invisible collision surface and the visible geometry.

level

Stores a special effective normal with the collision solid that points up, regardless of the actual shape or orientation of the solid. This can be used to allow an avatar to stand on a sloping surface without having a tendency to slide downward.

<ObjectType> { type }

This is a short form to indicate one of several pre-canned sets of attributes. Type may be any word, and a Config definition will be searched for by the name "egg-object-type-word", where "word" is the type word. This definition may contain any arbitrary egg syntax to be parsed in at this group level.

A number of predefined ObjectType definitions are provided:

barrier

This is equivalent to <Collide> { Polyset descend }. The geometry defined at this root and below defines an invisible collision solid.

trigger

This is equivalent to <Collide> { Polyset descend intangible }. The geometry defined at this root and below defines an invisible trigger surface.

sphere

Equivalent to <Collide> { Sphere descend }. The geometry is replaced with the smallest collision sphere that will enclose it. Typically you model a sphere in polygons and put this flag on it to create a collision sphere of the same size.

tube

Equivalent to <Collide> { Tube descend }. As in sphere, above, but the geometry is replaced with a collision tube (a capsule). Typically you will model a capsule or a cylinder in polygons.

bubble

Equivalent to <Collide> { Sphere keep descend }. A collision bubble is placed around the geometry, which is otherwise unchanged.

ghost

Equivalent to <Scalar> collide-mask { 0 }. It means that the geometry beginning at this node and below should never be collided with--characters will pass through it.

backstage

This has no equivalent; it is treated as a special case. It means that the geometry at this node and below should not be translated. This will normally be used on scale references and other modeling tools.

There may also be additional predefined egg object types not listed here; see the *.pp files that are installed into the etc directory for a complete list.

```
<Transform> { transform-definition }
```

This specifies a matrix transform at this group level. This defines a local coordinate space for this group and its

descendents. Vertices are still specified in world coordinates (in a vertex pool), but any geometry assigned to this group will be inverse transformed to move its vertices to the local space.

The transform definition may be any sequence of zero or more of the following. Transformations are post multiplied in the order they are encountered to produce a net transformation matrix. Rotations are defined as a counterclockwise angle in degrees about a particular axis, either implicit (about the x, y, or z axis), or arbitrary. Matrices, when specified explicitly, are row-major.

```
<Translate> { x y z }
<RotX> { degrees }
<RotY> { degrees }
<RotZ> { degrees }
<Rotate> { degrees x y z }
<Scale> { x y z }
<Scale> { s }

<Matrix4> {
 00 01 02 03
 10 11 12 13
 20 21 22 23
 30 31 32 33
}
```

Note that the <Transform> block should always define a 3-d transform when it appears within the body of a <Group>, while it may define either a 2-d or a 3-d transform when it appears within the body of a <Texture>. See <Texture>, above.

```
<DefaultPose> { transform-definition }
```

This defines an optional default pose transform, which might be a different transform from that defined by the <Transform> entry, above. This makes sense only for a <Joint>. See the <Joint> description, below.

The default pose transform defines the transform the joint will maintain in the absence of any animation being applied. This is

different from the <Transform> entry, which defines the coordinate space the joint must have in order to keep its vertices in their (global space) position as given in the egg file. If this is different from the <Transform> entry, the joint's vertices will *not* be in their egg file position at initial load. If there is no <DefaultPose> entry for a particular joint, the implicit default-pose transform is the same as the <Transform> entry.

Normally, the <DefaultPose> entry, if any, is created by the egg-optchar -defpose option. Most other software has little reason to specify an explicit <DefaultPose>.

```
<VertexRef> { indices <Ref> { pool-name } }
```

This moves geometry created from the named vertices into the current group, regardless of the group in which the geometry is actually defined. See the <Joint> description, below.

```
<AnimPreload> {
 <Scalar> fps { float-value }
 <Scalar> num-frames { integer-value }
}
```

One or more AnimPreload entries may appear within the <Group> that contains a <Dart> entry, indicating an animated character (see above). These AnimPreload entries record the minimal preloaded animation data required in order to support asynchronous animation binding. These entries are typically generated by the egg-optchar program with the -preload option, and are used by the Actor code when allow-async-bind is True (the default).

```
<Instance> name { group-body }
```

An <Instance> node is exactly like a <Group> node, except that vertices referenced by geometry created under the <Instance> node are not assumed to be given in world coordinates, but are instead given in the local space of the <Instance> node itself (including any transforms given to the node).

In other words, geometry under an <Instance> node is defined in local coordinates. In principle, similar geometry can be created

under several different <Instance> nodes, and thus can be positioned in a different place in the scene each instance. This doesn't necessarily imply the use of shared geometry in the Panda3D scene graph, but see the <Ref> syntax, below.

This is particularly useful in conjunction with a <File> entry, to load external file references at places other than the origin.

A special syntax of <Instance> entries does actually create shared geometry in the scene graph. The syntax is:

```
<Instance> name {
 <Ref> { group-name }
 [ <Ref> { group-name } ... ]
}
```

In this case, the referenced group name will appear as a duplicate instance in this part of the tree. Local transforms can be applied and are relative to the referencing group's transform. The referenced group must appear preceding this point in the egg file, and it will also be a part of the scene in the point at which it first appears. The referenced group may be either a <Group> or an <Instance> of its own; usually, it is a <Group> nested within an earlier <Instance> entry.

```
<Joint> name { [transform] [ref-list] [joint-list] }
```

A joint is a highly specialized kind of grouping node. A tree of joints is used to specify the skeletal structure of an animated character.

A joint may only contain one of three things. It may contain a <Transform> entry, as above, which defines the joint's unanimated (rest) position; it may contain lists of assigned vertices or CV's; and it may contain other joints.

A tree of <Joint> nodes only makes sense within a character definition, which is created by applying the <DART> flag to a group. See <DART>, above.

The vertex assignment is crucial. This is how the geometry of a character is made to move with the joints. The character's geometry is actually defined outside the joint tree, and each vertex must be assigned to one or more joints within the tree.
This is done with zero or more <VertexRef> entries per joint, as the following:

```
<VertexRef> { indices [<Scalar> membership { m }] <Ref> { pool-name } }
```

This is syntactically similar to the way vertices are assigned to polygons. Each <VertexRef> entry can assign vertices from only one vertex pool (but there may be many <VertexRef> entries per joint).
Indices is a list of vertex numbers from the specied vertex pool, in an arbitrary order.
The membership scalar is optional. If specified, it is a value between 0.0 and 1.0 that indicates the fraction of dominance this joint has over the vertices. This is used to implement soft-skinning, so that each vertex may have partial ownership in several joints.
The <VertexRef> entry may also be given to ordinary <Group> nodes. In this case, it treats the geometry as if it was parented under the group in the first place. Non-total membership assignments are meaningless.

```
<Bundle> name { table-list }
<Table> name { table-body }
```

A table is a set of animated values for joints. A tree of tables with the same structure as the corresponding tree of joints must be defined for each character to be animated. Such a tree is placed under a <Bundle> node, which provides a handle within Panda to the tree as a whole.
Bundles may only contain tables; tables may contain more tables, bundles, or any one of the following (<Scalar> entries are optional, and default as shown):

```
<S$Anim> name {
  <Scalar> fps { 24 }
  <V> { values }
}
```

This is a table of scalar values, one per frame. This may be applied to a morph slider, for instance.

```
<Xfm$Anim> name {
  <Scalar> fps { 24 }
  <Scalar> order { srpht }
  <Scalar> contents { ijkabcrphxyz }
  <V> { values }
}
```

This is a table of matrix transforms, one per frame, such as may be applied to a joint. The "contents" string consists of a subset of the letters "ijkabcrphxyz", where each letter corresponds to a column of the table; <V> is a list of numbers of length(contents) * num_frames. Each letter of the contents string corresponds to a type of transformation:

i, j, k - scale in x, y, z directions, respectively
a, b, c - shear in xy, xz, and yz planes, respectively
r, p, h - rotate by roll, pitch, heading
x, y, z - translate in x, y, z directions

The net transformation matrix specified by each row of the table is defined as the net effect of each of the individual columns' transform, according to the corresponding letter in the contents string. The order the transforms are applied is defined by the order string:

s - all scale and shear transforms
r, p, h - individual rotate transforms
t - all translation transforms

```
<Xfm$Anim_S$> name {
  <Scalar> fps { 24 }
  <Scalar> order { srpht }
  <S$Anim> i { ... }
  <S$Anim> j { ... }
  ...
}
```

This is a variant on the <Xfm$Anim> entry, where each column of the table is entered as a separate <S$Anim> table. This syntax reflects an attempt to simplify the description by not requiring repetition of values for columns that did not change value during an animation sequence.

```
<VertexAnim> name {
  <Scalar> width { table-width }
  <Scalar> fps { 24 }
  <V> { values }
}
```

This is a table of vertex positions, normals, texture coordinates, or colors. These values will be subsituted at runtime for the corresponding values in a <DynamicVertexPool>. The name of the table should be "coords", "norms", "texCoords", or "colors", according to the type of values defined. The number table-width is the number of floats in each row of the table. In the case of a coords or norms table, this must be 3 times the number of vertices in the corresponding dynamic vertex pool. (For texCoords and colors, this number must be 2 times and 4 times, respectively.)

MISCELLANEOUS

```
<File> { filename }
```

This includes a copy of the referenced egg file at the current point. This is usually placed under an <Instance> node, so that the current transform will apply to the geometry in the external file. The extension ".egg" is implied if it is omitted.

As with texture filenames, the filename may be a relative path, in which case the current egg file's directory is searched first, and then the model-path is searched.

ANIMATION STRUCTURE

Unanimated models may be defined in egg files without much regard to any particular structure, so long as named entries like VertexPools and Textures appear before they are referenced.

However, a certain rigid structural convention must be followed in order to properly define an animated skeleton-morph model and its associated animation data.

The structure for an animated model should resemble the following:

```
<Group> CHARACTER_NAME {
 <Dart> { 1 }
 <Joint> JOINT_A {
  <Transform> { ... }
  <VertexRef> { ... }
  <Group> { <Polygon> ... }
  <Joint> JOINT_B {
   <Transform> { ... }
   <VertexRef> { ... }
   <Group> { <Polygon> ... }
  }
  <Joint> JOINT_C {
   <Transform> { ... }
   <VertexRef> { ... }
   <Group> { <Polygon> ... }
  }
  ...
 }
}
```

The <Dart> flag is necessary to indicate that this group begins an animated model description. Without the <Dart> flag, joints will be treated as ordinary groups, and morphs will be ignored.

In the above, UPPERCASE NAMES represent an arbitrary name that you may choose. The name of the enclosing group, CHARACTER_NAME, is taken as the name of the animated model. It should generally match the bundle name in the associated animation tables.

Within the <Dart> group, you may define an arbitrary hierarchy of <Joint> entries. There may be as many <Joint> entries as you like, and they may have any nesting complexity you like. There may be either one root <Joint>, or multiple roots. However, you must always include at least one <Joint>, even if your animation consists entirely of morphs.

Polygons may be directly attached to joints by enclosing them within the <Joint> group, perhaps with additional nesting <Group> entries, as illustrated above. This will result in the polygon's vertices being hard-assigned to the joint it appears within. Alternatively, you declare the polygons elsewhere in the egg file, and use <VertexRef> entries within the <Joint> group to associate the vertices with the joints. This is the more common approach, since it allows for soft-assignment of vertices to multiple joints.

It is not necessary for every joint to have vertices at all. Every joint should include a transform entry, however, which defines the initial, resting transform of the joint (but see also <DefaultPose>, above). If a transform is omitted, the identity transform is assumed.

Some of the vertex definitions may include morph entries, as described in MORPH DESCRIPTION ENTRIES, above. These are meaningful only for vertices that are assigned, either implicitly or explicitly, to at least one joint.

You may have multiple versions of a particular animated model--for instance, multiple different LOD's, or multiple different clothing options. Normally each different version is stored in a different egg file, but it is also possible to include multiple versions within the same egg file. If the different versions are intended to play the same animations, they should all have the same CHARACTER_NAME, and their joint hierarchies should exactly match in structure and names.

The structure for an animation table should resemble the following:

```
<Table> {
 <Bundle> CHARACTER_NAME {
  <Table> "<skeleton>" {
   <Table> JOINT_A {
```

```
  <Xfm$Anim_S$> xform {
   <Char*> order { sphrt }
   <Scalar> fps { 24 }
   <S$Anim> x { 0 0 10 10 20 ... }
   <S$Anim> y { 0 0 0 0 0 ... }
   <S$Anim> z { 20 20 20 20 20 ... }
  }
  <Table> JOINT_B {
   <Xfm$Anim_S$> xform {
    <Char*> order { sphrt }
    <Scalar> fps { 24 }
    <S$Anim> x { ... }
    <S$Anim> y { ... }
    <S$Anim> z { ... }
   }
  }
  <Table> JOINT_C {
   <Xfm$Anim_S$> xform {
    <Char*> order { sphrt }
    <Scalar> fps { 24 }
    <S$Anim> x { ... }
    <S$Anim> y { ... }
    <S$Anim> z { ... }
   }
  }
 }
}
<Table> morph {
 <S$Anim> MORPH_A {
  <Scalar> fps { 24 }
  <V> { 0 0 0 0.1 0.2 0.3 1 ... }
 }
 <S$Anim> MORPH_B {
  <Scalar> fps { 24 }
  <V> { ... }
 }
 <S$Anim> MORPH_C {
  <Scalar> fps { 24 }
  <V> { ... }
```

```
    }
   }
  }
 }
```

The <Bundle> entry begins an animation table description. This entry must have at least one child: a <Table> named "<skeleton>" (this name is a literal keyword and must be present). The children of this <Table> entry should be a hierarchy of additional <Table> entries, one for each joint in the model. The joint structure and names defined by the <Table> hierarchy should exactly match the joint structure and names defined by the <Joint> hierarchy in the corresponding model.

Each <Table> that corresponds to a joint should have one child, an <Xfm$Anim_S$> entry named "xform" (this name is a literal keyword and must be present). Within this entry, there is a series of up to twelve <S$Anim> entries, each with a one-letter name like "x", "y", or "z", which define the per-frame x, y, z position of the corresponding joint. There is one numeric entry for each frame, and all frames represent the same length of time. You can also define rotation, scale, and shear. See the full description of <Xfm$Anim_S$>, above.

Within a particular animation bundle, all of the various components throughout the various <Tables> should define the same number of frames, with the exception that if any of them define exactly one frame value, that value is understood to be replicated the appropriate number of times to match the number of frames defined by other components.

(Note that you may alternatively define an animation table with an <Xfm$Anim> entry, which defines all of the individual components in one big matrix instead of individually. See the full description above.)

Each joint defines its frame rate independently, with an "fps" scalar. This determines the number of frames per second for the frame data within this table. Typically, all joints have the same frame rate, but it is possible for different joints to animate at different speeds.

Each joint also defines the order in which its components should be composed to determine the complete transform matrix, with an "order" scalar. This is described in more detail above.

If any of the vertices in the model have morphs, the top-level <Table> should also include a <Table> named "morph" (this name is also a literal keyword). This table in turn contains a list of <S$Anim> entries, one for each named morph description. Each table contains a list of numeric values, one per frame; as with the joint data, there should be the same number of numeric values in all tables, with the exception that just one value is understood to mean hold that value through the entire animation.

The "morph" table may be omitted if there are no morphs defined in the model.

There should be a separate <Bundle> definition for each different animation. The <Bundle> name should match the CHARACTER_NAME used for the model, above. Typically each bundle is stored in a separate egg file, but it is also possible to store multiple different animation bundles within the same egg file. If you do this, you may violate the CHARACTER_NAME rule, and give each bundle a different name; this will become the name of the animation in the Actor interface.

Although animations and models are typically stored in separate egg files, it is possible to store them together in one large egg file. The Actor interface will then make available all of the animations it finds within the egg file, by bundle name.

HOW TO CONTROL RENDER ORDER

In most simple scenes, you can naively attach geometry to the scene graph and let Panda decide the order in which objects should be rendered. Generally, it will do a good enough job, but there are occasions in which it is necessary to step in and take control of the process.

To do this well, you need to understand the implications of render order. In a typical OpenGL- or DirectX-style Z-buffered system, the order in which primitives are sent to the graphics hardware is theoretically unimportant, but in practice there are many important reasons for rendering one object before another.

Firstly, state sorting is one important optimization. This means choosing to render things that have similar state (texture, color, etc.) all at the same time, to minimize the number of times the graphics hardware has to be told to change state in a particular frame. This sort of optimization is particularly important for very high-end graphics hardware, which achieves its advertised theoretical polygon throughput only in the absence of any state changes; for many such advanced cards, each state change request will completely flush the register cache and force a restart of the pipeline.

Secondly, some hardware has a different optimization requirement, and may benefit from drawing nearer things before farther things, so that the Z-buffer algorithm can effectively short-circuit some of the advanced shading features in the graphics card for pixels that would be obscured anyway. This sort of hardware will draw things fastest when the scene is sorted in order from the nearest object to the farthest object, or "front-to-back" ordering.

Finally, regardless of the rendering optimizations described above, a particular sorting order is required to render transparency properly (in the absence of the specialized transparency support that only a few graphics cards provide). Transparent and semitransparent objects are normally rendered by blending their semitransparent parts with

what has already been drawn to the framebuffer, which means that it is important that everything that will appear behind a semitransparent object must have already been drawn before the semitransparent parts of the occluding object is drawn. This implies that all semitransparent objects must be drawn in order from farthest away to nearest, or in "back-to-front" ordering, and furthermore that the opaque objects should all be drawn before any of the semitransparent objects.
Panda achieves these sometimes conflicting sorting requirements through the use of bins.

CULL BINS

The CullBinManager is a global object that maintains a list of all of the cull bins in the world, and their properties. Initially, there are five default bins, and they will be rendered in the following order:

Bin Name	Sort	Type
"background"	10	BT_fixed
"opaque"	20	BT_state_sorted
"transparent"	30	BT_back_to_front
"fixed"	40	BT_fixed
"unsorted"	50	BT_unsorted

When Panda traverses the scene graph each frame for rendering, it assigns each Geom it encounters into one of the bins defined in the CullBinManager. (The above lists only the default bins. Additional bins may be created as needed, using either the CullBinManager::add_bin() method, or the Config.prc "cull-bin" variable.)

You may assign a node or nodes to an explicit bin using the NodePath::set_bin() interface. set_bin() requires two parameters, the bin name and an integer sort parameter; the sort parameter is only meaningful if the bin type is BT_fixed (more on this below), but it must always be specified regardless.

If a node is not explicitly assigned to a particular bin, then Panda will assign it into either the "opaque" or the "transparent" bin, according to whether it has transparency enabled or not. (Note that

the reverse is not true: explicitly assigning an object into the "transparent" bin does not automatically enable transparency for the object.)

When the entire scene has been traversed and all objects have been assigned to bins, then the bins are rendered in order according to their sort parameter. Within each bin, the contents are sorted according to the bin type.

The following bin types may be specified:

BT_fixed

Render all of the objects in the bin in a fixed order specified by the user. This is according to the second parameter of the NodePath::set_bin() method; objects with a lower value are drawn first.

BT_state_sorted

Collects together objects that share similar state and renders them together, in an attempt to minimize state transitions in the scene.

BT_back_to_front

Sorts each Geom according to the center of its bounding volume, in linear distance from the camera plane, so that farther objects are drawn first. That is, in Panda's default right-handed Z-up coordinate system, objects with large positive Y are drawn before objects with smaller positive Y.

BT_front_to_back

The reverse of back_to_front, this sorts so that nearer objects are drawn first.

BT_unsorted

Objects are drawn in the order in which they appear in the scene graph, in a depth-first traversal from top to bottom and then from left to right.

How to make multipart actor

MULTIPART ACTORS vs. HALF-BODY ANIMATION

Sometimes you want to be able to play two different animations on the same Actor at once. Panda does have support for blending two animations on the whole Actor simultaneously, but what if you want to play one animation (say, a walk cycle) on the legs while a completely different animation (say, a shoot animation) is playing on the torso?

Although Panda doesn't currently have support for playing two different animations on different parts of the same actor at once (half-body animation), it does support loading up two completely different models into one actor (multipart actors), which can be used to achieve the same effect, albeit with a bit more setup effort.

Multipart actors are more powerful than half-body animations, since you can completely mix-and-match the pieces with parts from other characters: for instance, you can swap out short legs for long legs to make your character taller. On the other hand, multipart actors are also more limited in that there cannot be any polygons that straddle the connecting joint between the two parts.

BROAD OVERVIEW

What you have to do is split your character into two completely different models: the legs and the torso. You don't have to do this in the modeling package; you should be able to do it in the conversion process. The converter needs to be told to get out the entire skeleton, but just a subset of the geometry. Maya2egg, for instance, will do this with the -subset command-line parameter.

Then, in a nutshell, you load up a multipart actor with the legs and the torso as separate parts, and you can play the same animation on both parts, or you can use the per-part interface to play a different animation on each part.

MORE DETAILS

That nutshell oversimplifies things only a little bit. Unless your different animations are very similar to each other, you will have issues keeping the different parts from animating in different directions. To solve this, you need to parent them together properly, so that the torso is parented to the hips. This means exposing the hip joint in the legs model, and subtracting the hip joint animation from the torso model using egg-topstrip (because it will pick it up again when it gets stacked up on the hips). Also, you should strongly consider egg-optchar to remove the unused joints from each part's skeleton, although this step is just an optimization.

Unfortunately, all this only works if your character has no polygons that straddle the connecting joint between the hips and the torso. If it does, you may have to find a clever place to draw the line between them (under a shirt?) so that the pieces can animate in different directions without visible artifacts. If that can't be done, then the only solution is to add true half-body animation support to Panda. :)

NUTS AND BOLTS

You need to parent the two parts together in Panda. The complete process is this (of course, you'll need to flesh out the details of the maya2egg command line according to the needs of your model, and insert your own filenames and joint names where appropriate):

(1) Extract out the model into two separate files, legs and torso.
Extract the animation out twice too, even though both copies will be the same, just so it can conveniently exist in two different egg files, one for the legs and one for the torso.

```
maya2egg -subset legs_group -a model -cn legs -o legs-model.egg myFile.mb
maya2egg -a chan -cn legs -o legs-walk.egg myFile.mb
maya2egg -subset torso_group -a model -cn torso -o torso-model.egg myFile.mb
maya2egg -a chan -cn torso -o torso-walk.egg myFile.mb
```

Note that I use the -cn option to give the legs and torso pieces different character names. It helps out Panda to know which animations are intended to be played with which models, and the character name serves this purpose--this way I can now just type:

```
pview legs-model.egg legs-walk.egg torso-model.egg torso-walk.eggPanda will bind up the
```

appropriate animations to their associated
models automatically, and I should see my character walking normally. We could skip straight to step (5) now, but the character isn't stacked up yet, and he's only sticking together now because we're playing the walk animation on both parts at the same time--if we want to play different animations on different parts, we have to stack them.

(2) Expose the hip joint on the legs:

egg-optchar -d opt -expose hip_joint legs-model.egg legs-walk.egg

(3) Strip out the hip joint animation from the torso and egg-optchar it to remove the leg joints:

egg-topstrip -d strip -t hip_joint torso-model.egg torso-walk.egg
egg-optchar -d opt strip/torso-model.egg strip/torso-walk.egg

(4) Bamify everything.

egg2bam -o legs-model.bam opt/legs-model.egg
egg2bam -o legs-walk.bam opt/legs-walk.egg
egg2bam -o torso-model.bam opt/torso-model.egg
egg2bam -o torso-walk.bam opt/torso-walk.egg

(5) Create a multipart character in Panda. This means loading up the torso model and parenting it, in toto, to the hip joint of the legs. But the Actor interface handles this for you:

```
from direct.actor import Actor

a = Actor.Actor(
 # part dictionary
 { 'torso' : 'torso-model.bam',
  'legs' : 'legs-model.bam',
 },

 # anim dictionary
 { 'torso' : {'walk' : 'torso-walk.bam'},
  'legs' : {'walk' : 'legs-walk.bam'},
 })
```

```
# Tell the Actor how to stack the pieces.
a.attach('torso', 'legs', 'hip_joint')
```

(6) You can now play animations on the whole actor, or on only part of it:

```
a.loop('walk')

a.stop()
a.loop('walk', partName = 'legs')
```

MULTIGEN MODEL FLAGS

This document describes the different kinds of model flags one can place in the comment field of MultiGen group beads. The general format for a model flag is:

<egg> { <FLAGNAME> {value} }

The most up-to-date version of this document can be found in:

$PANDA/src/doc/howto.MultiGenModelFlags

****

QUICKREF

****

FLAG	DESCRIPTION
<egg> { <Model> {1} }	Handle to show/hide, color, etc. a chunk
<egg> { <DCS> {1} }	Handle to move, rotate, scale a chunk
<egg> { <ObjectType> {barrier} }	Invisible collision surface
<egg> { <ObjectType> {trigger} }	Invisible trigger polygon
<egg> { <ObjectType> {floor} }	Collides with vertical ray (used to specify avatar height and zone)
<egg> { <ObjectType> {sphere} }	Invisible sphere collision surface
<egg> { <ObjectType> {trigger-sphere} }	Invisible sphere collision surface
<egg> { <ObjectType> {camera-collide} }	Invisible collision surface for camera
<egg> { <ObjectType> {camera-collide-sphere} }	Invisible collision surface for camera
<egg> { <ObjectType> {camera-barrier} }	Invisible collision surface for camera and colliders
<egg> { <ObjectType> {camera-barrier-sphere} }	Invisible sphere collision surface for camera and colliders
<egg> { <ObjectType> {backstage} }	Modeling reference object
<egg> { <Decal> {1} }	Decal the node below to me (like a window on a wall)
<egg> { <Scalar> fps { # } }	Set rate of animation for a pfSequence

****

DETAILS

****

The player uses several different types of model flags: HANDLES, BEHAVIORS, and PROPERTIES. The following sections give examples of some of the most common flag/value pairs and describes what they are used for.

HANDLES

These flags give the programmers handles which they can use to show/hide, move around, control the texture, etc. of selected segments (chunks) of the model. The handle is the name of the object bead in which one places the flag (so names like red-hut are more useful than names like o34).

<egg> { <Model> {1} }

Used to show/hide, change the color, or change the collision properties of a chunk.

<egg> { <DCS> {1} }

Used to move, rotate, or scale a chunk of the model. Also can be used (like the <Model> flag) to show/hide, change the color, and change the collision properties of a chunk.

BEHAVIORS

These flags are used to control collision properties, visibility and behavior of selected chunks. An "X" in the associated column means:

VISIBLE the object can be seen (see NOTE below for invisible objects)
SOLID avatars can not pass through the object
EVENT an event is thrown whenever an avatar collides with the object

	VISIBLE	SOLID	EVENT
<egg> { <ObjectType> {barrier} }		X	X
<egg> { <ObjectType> {trigger} }			X
<egg> { <ObjectType> {backstage} }			

Descriptions

- BARRIERS are invisible objects that block the avatars. Use these to funnel avatars through doorways, keep them from falling off bridges, and so on.
- TRIGGERS can be used to signal when avatars have entered a certain area of the model. One could place a trigger polygon in front of a door, for example, so the player can tell when the avatar has moved through the door.
- BACKSTAGE objects are not translated over to the player. Modelers should use this flag on reference objects that they include to help in the modeling task (such as scale references)

IMPORTANT NOTE

It is not necessary, and in fact some cases it will actually cause problems if you set the transparency value for the invisible objects above (barrier, trigger, eye-trigger) to 0.0. These objects will automatically be invisible in the player if they have been flagged as one of these three invisible types. If you wish to make it clear in MultiGen that these objects are invisible objects, set the transparency value to some intermediate level (0.5). Again, do not set the transparency value to 0.0.

PROPERTIES

These are used to control properties of selected chunks.

<egg> { <Scalar> fps { frame-rate } }

This specifies the rate of animation for a pfSequence node

****

NOTES

****

1) Combinations

Multiple Flag/value pairs can be combined within an single <egg> field.
For example:

<egg> { <Model> {1}
        <ObjectType> {barrier} }

Generally, the <Model> flag can be combined with most other flags
(except DCS). Each bead, however, can only have *one* <ObjectType> flag.

2) Newlines, spaces, and case (usually) do not matter. This above entry
could also be written as:

<egg>{<model>{1}<objecttype>{barrier}}

3) Where to place the flags

All model flags except <Normal> flags are generally placed in the
topmost group bead of the geometry to which the flag applies.

```
GROUP  <- place flags here, except <Normal>
  |
  --------------------------
  |          |          |
OBJECT1     OBJECT2     OBJECT3 .....
  |          |          |
polygons    polygons    polygons  <- place <Normal> flag here
```

Flags can also be placed in object beads, though for consistency sake
its better to place them in the group beads.

4) Flags at different levels in the model

Flags in lower level beads generally override flags in upper level
beads.

5) For more detailed information see THE PHILOSOPHY OF EGG FILES in Part 2 of this manual or refer to source in $PANDA/src/doc/eggSyntax.txt.

Multi-Texturing in Maya

A good rule of thumb is to create your Multi-Layered shader first to get an idea of what kind of blendmode you want. You can do that by using Maya's kLayeredShader.

Following blendmode from Maya is supported directly in Panda.

"Multiply" => "Modulate"
"Over" => "Decal"
"Add" => "Add"

More blendmodes will be supported very soon. You should be able to pview this change if you restart Maya from the "runmaya.bat" (or however you restart maya).

Once the shader is setup, you should create the texture coordinates or uvsets for your multitexture. Make sure, the uvset name matches the shader names that you made
in the kLayeredShader. For Example, if the two shaders (not the texure file name) in your kLayeredShader are called "base" and "top", then your geometry (that will have
the layeresShader) will have two uvsets called "base" and "top".

After this you will link the uvsets to the appropriate shaders.

A reminder note: by default the alpha channel of the texture on the bottom is dropped in the conversion. If you want to retain the alpha channel of your texture,
please make a connection to the alpha channel in Maya when setting up the shader (alpha on the layerShader will be highlighted in yellow).

Config

This document describes the use of the Panda's Config.prc configuration files and the runtime subsystem that extracts values from these files, defined in dtool/src/prc.

The Config.prc files are used for runtime configuration only, and are not related to the Config.pp files, which control compile-time configuration. If you are looking for documentation on the Config.pp files, see howto.use_ppremake.txt, and ppremake-*.txt, in this directory.

USING THE PRC FILES

In its default mode, when Panda starts up it will search in the install/etc directory (or in the directory named by the environment variable PRC_DIR if it is set) for all files named *.prc (that is, any files with an extension of "prc") and read each of them for runtime configuration. (It is possible to change this default behavior; see COMPILE-TIME OPTIONS FOR FINDING PRC FILES, below.)

All of the prc files are loaded in alphabetical order, so that the files that have alphabetically later names are loaded last. Since variables defined in an later file may shadow variables defined in an earlier file, this means that filenames towards the end of the alphabet have the most precedence.

Panda by default installs a handful of system prc files into the install/etc directory. These files have names beginning with digits, like 20_panda.prc and 40_direct.prc, so that they will be loaded in a particular order. If you create your own prc file in this directory, we recommend that you begin its filename with letters, so that it will sort to the bottom of the list and will therefore override any of the default variables defined in the system prc files.

Within a particular prc file, you may define any number of configuration variables and their associated value. Each definition must appear one per line, with at least one space separating the

variable and its definition, e.g.:

load-display pandagl

This specifies that the variable "load-display" should have the value "pandagl".

Comments may also appear in the file; they are introduced by a leading hash mark (#). A comment may be on a line by itself, or it may be on the same line following a variable definition; if it is on the same line as a variable definition, the hash mark must be preceded by at least one space to separate it from the definition.

The legal values that you may specify for any particular variable depends on the variable. The complete list of available variables and the valid values for each is not documented here (a list of the most commonly modified variables appears in another document, but also see cvMgr.listVariables(), below).

Many variables accept any string value (such as load-display, above); many others, such as aspect-ratio, expect a numeric value.

A large number of variables expect a simple boolean true/false value. You may observe the Python convention of using 0 vs. 1 to represent false vs. true; or you may literally type "false" or "true", or just "f" and "t". For historical reasons, Panda also recognizes the Scheme convention of "#f" and "#t".

Most variables only accept one value at a time. If there are two different definitions for a given variable in the same file, the topmost definition applies. If there are two different definitions in two different files, the definition given in the file loaded later applies.

However, some variables accept multiple values. This is particularly common for variables that name search directories, like model-path. In the case of this kind of variable, all definitions given for the variable are taken together; it is possible to extend the definition by adding another prc file, but you cannot remove any value defined in a previously-loaded prc file.

DEFINING CONFIG VARIABLES

New config variables may be defined on-the-fly in either C++ or Python code. To do this, create an instance of one of the following classes:

ConfigVariableString
ConfigVariableBool
ConfigVariableInt
ConfigVariableDouble
ConfigVariableFilename
ConfigVariableEnum (C++ only)
ConfigVariableList
ConfigVariableSearchPath

These each define a config variable of the corresponding type. For instance, a ConfigVariableInt defines a variable whose value must always be an integer value. The most common variable types are the top four, which are self-explanatory; the remaining four are special types:

ConfigVariableFilename -

This is a convenience class which behaves very much like a ConfigVariableString, except that it automatically converts from OS-specific filenames that may be given in the prc file to Panda-specific filenames, and it also automatically expands environment variable references, so that the user may name a file based on the value of an environment variable (e.g. $PANDAMODELS/file.egg).

ConfigVariableEnum -

This is a special template class available in C++ only. It provides a convenient way to define a variable that may accept any of a handful of different values, each of which is defined by a keyword. For instance, the text-encoding variable may be set to any of "iso8859", "utf8", or "unicode", which correspond to TextEncoder::E_iso8859, E_utf8, and E_unicode, respectively.

The ConfigVariableEnum class relies on a having sensible pair of

functions defined for operator << (ostream) and operator >>(istream)
for the enumerated type. These two functions should
reverse each other, so that the output operator generates a keyword
for each value of the enumerated type, and the input operator
recognizes each of the keywords generated by the output operator.

This is a template class. It is templated on its enumerated type,
e.g. ConfigVariableEnum<TextEncoder::Encoding>.

ConfigVariableList -

This class defines a special config variable that records all of its
definitions appearing in all prc files and retrieves them as a list,
instead of a standard config variable that returns only the topmost
definition. (See "some variables accept multiple values", above.)

Unlike the other kinds of config variables, a ConfigVariableList is
read-only; it can be modified only by loading additional prc files,
rather than directly setting its value. Also, its constructor lacks
a default_value parameter, since there is no default value (if the
variable is not defined in any prc file, it simply returns an empty
list).

ConfigVariableSearchPath -

This class is very similar to a ConfigVariableList, above, except
that it is intended specifically to represent the multiple
directories of a search path. In general, a
ConfigVariableSearchPath variable can be used in place of a
DSearchPath variable.

Unlike ConfigVariableList, instances of this variable can be locally
modified by appending or prepending additional directory names.

In general, each of the constructors to the above classes accepts the
following parameters:

(name, default_value, description = "", flags = 0)

The default_value parameter should be of the same type as the variable
itself; for instance, the default_value for a ConfigVariableBool must

be either true or false. The ConfigVariableList and ConfigVariableSearchPath constructors do not have a default_value
parameter.

The description should be a sentence or two describing the purpose of the variable and the effects of setting it. It will be reported with variable.getDescription() or ConfigVariableManager.listVariables(); see QUERYING CONFIG VARIABLES, below.

The flags variable is usually set to 0, but it may be an integer trust level and/or the union of any of the values in the enumerated type ConfigFlags::VariableFlags. For the most part, this is used to restrict the variable from being set by unsigned prc files. See SIGNED PRC FILES, below.

Once you have created a config variable of the appropriate type, you may generally treat it directly as a simple variable of that type. This works in both C++ and in Python. For instance, you may write code such as this:

```
ConfigVariableInt foo_level("foo-level", -1, "The level of foo");

if (foo_level < 0) {
  cerr << "You didn't specify a valid foo_level!\n";

} else {
  // Four snarfs for every foo.
  int snarf_level = 4 * foo_level;
}
```

In rare cases, you may find that the implicit typecast operators aren't resolved properly by the compiler; if this happens, you can use variable.get_value() to retrieve the variable's value explicitly.

DIRECTLY ASSIGNING CONFIG VARIABLES

In general, config variables can be directly assigned values appropriate to their type, as if they were ordinary variables. In C++, the assignment operator is overloaded to perform this function,

e.g.:
 foo_level = 5;
In Python, this syntax is not possible--the assignment operator in Python completely replaces the value of the assigned symbol and cannot be overloaded. So the above statement in Python would replace foo_level with an actual integer of the value 5. In many cases, this is close enough to what you intended anyway, but if you want to keep the original functionality of the config variable (e.g. so you can restore it to its original value later), you need to use the set_value() method instead, like this:

```
fooLevel.setValue(5)
```

When you assign a variable locally, the new definition shadows all prc files that have been read or will ever be read, until you clear your definition. To restore a variable to its original value as defined by the topmost prc file, use clear_local_value():

```
fooLevel.clearLocalValue()
```

This interface for assigning config variables is primarily intended for the convenience of developing an application interactively; it is sometimes useful to change the value of a variable on the fly.

QUERYING CONFIG VARIABLES

There are several mechanisms for finding out the values of individual config variables, as well as for finding the complete list of available config variables.

In particular, one easy way to query an existing config variable's value is simply to create a new instance of that variable, e.g.:

```
print ConfigVariableInt("foo-level")
```

The default value and comment are optional if another instance of the same config variable has previously been created, supplying these parameters. However, it is an error if no instance of a particular config variable specifies a default value. It is also an error (but

it is treated as a warning) if two different instances of a variable specify different default values.
(Note that, although it is convenient to create a new instance of the variable in order to query or modify its value interactively, we recommend that all the references to a particular variable in code should use the same instance wherever possible. This minimizes the potential confusion about which instance should define the variable's default value and/or description, and reduces chance of conflicts should two such instances differ.)

If you don't know the type of the variable, you can also simply create an instance of the generic ConfigVariable class, for the purpose of querying an existing variable only (you should not define a new variable using the generic class).

To find out more detail about a variable and its value, use the ls() method in Python (or the write() method in C++), e.g.:

```
ConfigVariable("foo-level").ls()
```

In addition to the variable's current and default values, this also prints a list of all of the prc files that contributed to the value of the variable, as well as the description provided for the variable.

To get a list of all known config variables, use the methods on ConfigVariableManager. In C++, you can get a pointer this object via ConfigVariableManager::get_global_ptr(); in Python, use the cvMgr builtin, created by ShowBase.py.

```
print cvMgr
```

Lists all of the variables in active use: all of the variables whose value has been set by one or more prc files, along with the name of the prc file that defines that value.

```
cvMgr.listVariables()
```

Lists all of the variables currently known to the config system; that is, all variables for which a ConfigVariable instance has been created at runtime, whether or not its value has been changed from the default. This may omit variables defined in some unused

subsystem (like pandaegg, for instance), and it will omit
variables defined by Python code which hasn't yet been executed
(e.g. variables within defined with a function that hasn't yet been called).
This will also omit variables deemed to be "dynamic" variables,
for instance all of the notify-level-* variables, and variables
such as pstats-active-*. These are omitted simply to keep the
list of variable names manageable, since the list of dynamic
variable names tends to be very large. Use
cvMgr.listDynamicVariables() if you want to see these variable
names.

cvMgr.listUnusedVariables()

Lists all of the variables that have been defined by some prc
file, but which are not known to the config system (no
ConfigVariable instance has yet been created for this variable).
These variables may represent misspellings or typos in your prc
file, or they may be old variables which are no longer used in the
system. However, they may also be legitimate variables for some
subsystem or application which simply has not been loaded; there
is no way for Panda to make this distinction.

RE-READING PRC FILES

If you modify a prc file at some point after Panda has started, Panda
will not automatically know that it needs to reload its config files
and will not therefore automatically recognize your change. However,
you can force this to happen by making the following call:

ConfigPageManager::get_global_ptr()->reload_implicit_pages()

Or, in Python:

cpMgr.reloadImplicitPages()

This will tell Panda to re-read all of the prc files it found
automatically at startup and update the variables' values accordingly.

RUNTIME PRC FILE MANAGEMENT

In addition to the prc files that are found and loaded automatically by Panda at startup, you can load files up at runtime as needed. The
functions to manage this are defined in load_prc_file.h:

```
ConfigPage *page = load_prc_file("myPage.prc")

...

unload_prc_file(page);
```

(The above shows the C++ syntax; the corresponding Python code is similar, but of course the functions are named loadPrcFile() and unloadPrcFile().)

That is to say, you can call load_prc_file() to load up a new prc file at any time. Each file you load is added to a LIFO stack of prc files. If a variable is defined in more than one prc file, the topmost file on the stack (i.e. the one most recently loaded) is the one that defines the variable's value.

You can call unload_prc_file() at any time to unload a file that you have previously loaded. This removes the file from the stack and allows any variables it modified to return to their previous value. The single parameter to unload_prc_file() should be the pointer that was returned from the corresponding call to load_prc_file(). Once you have called unload_prc_file(), the pointer is invalid and should no longer be used. It is an error to call unload_prc_file() twice on the same pointer.

The filename passed to load_prc_file() may refer to any file that is on the standard prc file search path (e.g. $PRC_DIR), as well as on the model-path. It may be a physical file on disk, or a subfile of a multifile (and mounted via Panda's virtual file system).

If your prc file is stored as an in-memory string instead of as a disk file (for instance, maybe you just built it up), you can use the load_prc_file_data() method to load the prc file from the string data. The first parameter is an arbitrary name to assign to your in-memory

prc file; supply a filename if you have one, or use some other name
that is meaningful to you.

You can see the complete list of prc files that have been loaded into
the config system at any given time, including files loaded explicitly
via load_prc_file(), as well as files found in the standard prc file
search path and loaded implicitly at startup. Simply use
ConfigPageManager::write(), e.g. in Python:

```
print cpMgr
```

COMPILE-TIME OPTIONS FOR FINDING PRC FILES

As described above in USING THE PRC FILES, Panda's default startup
behavior is to load all files named *.prc in the directory named by
the environment variable PRC_DIR. This is actually a bit of an
oversimplification. The complete default behavior is as follows:

(1) If PRC_PATH is set, separate it into a list of directories and
    make a search path out of it.

(2) If PRC_DIR is set, prepend it onto the search path defined by
    PRC_PATH, above.

(3) If neither was set, put the compiled-in value for DEFAULT_PRC_DIR,
    which is usually the install/etc directory, alone on the search
    path.

    Steps (1), (2), and (3) define what is referred to in this
    document as "the standard prc search path". You can query this
    search path via cpMgr.getSearchPath().

(4) Look for all files named *.prc on each directory of the resulting
    search path, and load them up in reverse search path order, and
    within each directory, in forward alphabetical order. This means
    that directories listed first on the search path override
    directories listed later, and within a directory, files
    alphabetically later override files alphabetically earlier.

This describes the default behavior, without any modifications to Config.pp. If you wish, you can further fine-tune each of the above steps by defining various Config.pp variables at compile time. The following Config.pp variables may be defined:

#define PRC_PATH_ENVVARS PRC_PATH
#define PRC_DIR_ENVVARS PRC_DIR

These name the environment variable(s) to use instead of PRC_PATH and PRC_DIR. In either case, you may name multiple environment variables separated by a space; each variable is consulted one at a time, in the order named, and the results are concatenated.

For instance, if you put the following line in your Config.pp file:

#define PRC_PATH_ENVVARS CFG_PATH ETC_PATH

Then instead of checking $PRC_PATH in step (1), above, Panda will first check $CFG_PATH, and then $ETC_PATH, and the final search path will be the concatenation of both.

You can also define either or both of PRC_PATH_ENVVARS or PRC_DIR_ENVVARS to the empty string; this will disable runtime checking of environment variables, and force all prc files to be loaded from the directory named by DEFAULT_PRC_DIR.

#define PRC_PATTERNS *.prc

This describes the filename patterns that are used to identify prc files in each directory in step(4), above. The default is *.prc, but you can change this if you have any reason to. You can specify multiple filename patterns separated by a space. For instance, if you still have some config files named "Configrc", following an older Panda convention, you can define the following in your Config.pp file:

#define PRC_PATTERNS *.prc Configrc

This will cause Panda to recognize files named "Configrc", as well as any file ending in the extension prc, as a legitimate prc file.

#define DEFAULT_PRC_DIR $[INSTALL_DIR]/etc

This is the directory from which to load prc files if all of the variables named by PRC_PATH_ENVVARS and PRC_DIR_ENVVARS are undefined or empty.

#define DEFAULT_PATHSEP

This doesn't strictly apply to the config system, since it globally affects search paths throughout Panda. This specifies the character or characters used to separate the different directory names of a search path, for instance $PRC_PATH. The default character is ':' on Unix, and ';' on Windows. If you specify multiple characters, any of them may be used as a separator.

EXECUTABLE PRC FILES

One esoteric feature of Panda's config system is the ability to automatically execute a standalone program which generates a prc file as output.

This feature is not enabled by default. To enable it, you must define the Config.pp variable PRC_EXECUTABLE_PATTERNS before you build Panda. This variable is similar to PRC_PATTERNS, described above, except it names file names which, when found along the standard prc search path, should be taken to be the name of an executable program. Panda will execute each of these programs, in the appropriate order according to alphabetical sorting with the regular prc files, and whatever the program writes to standard output is taken to be the contents of a prc file.

By default the contents of the environment variable $PRC_EXECUTABLE_ARGS are passed as arguments to the executable program. You can change this to a different environment variable by redefining PRC_EXECUTABLE_ARGS_ENVVAR in your Config.pp (or prevent the passing of arguments by defining this to the empty string).

SIGNED PRC FILES

Another esoteric feature of Panda's config system is the ability to restrict certain config variables to modification only by a prc file that has been provided by an authorized source. This is primarily useful when Panda is to be used for deployment of applications (games, etc.) to a client; it has little utility in a fully trusted environment.

When this feature is enabled, you can specify an optional trust level to each ConfigVariable constructor. The trust level is an integer value, greater than 0 (and <= ConfigFlags::F_trust_level_mask), which should be or'ed in with the flags parameter.

A number of random keys must be generated ahead of time and compiled into Panda; there must be a different key for each different trust level. Each prc file can then optionally be signed by exactly one of the available keys. When a prc file has been signed by a recognized key, Panda assigns the corresponding trust level to that prc file. An unsigned prc file has an implicit trust level of 0.

If a signed prc file is modified in any way after it has been signed, its signature will no longer match the contents of the file and its trust level drops to 0. The newly-modified file must be signed again to restore its trust level.

When a ConfigVariable is constructed with a nonzero trust level, that variable's value may then not be set by any prc file with a trust level lower that the variable's trust level. If a prc file with an insufficient trust level attempts to modify the variable, the new value is ignored, and the value from the previous trusted prc file (or the variable's default value) is retained.

The default trust level for a ConfigVariable is 0, which means the variable can be set by any prc file, signed or unsigned. To set any nonzero trust level, pass the integer trust level value as the flags parameter to the ConfigVariable constructor. To explicitly specify a trust level of 0, pass ConfigFlags::F_open.

To specify a ConfigVariable that cannot be set by any prc files at
all, regardless of trust level, use ConfigFlags::F_closed.

This feature is not enabled by default. It is somewhat complicated to
enable this feature, because doing so requres generating one or more
private/public key pairs, and compiling the public keys into the
low-level Panda system so that it can recognize signed prc files when
they are provided, and compiling the private keys into standalone
executables, one for each private key, that can be used to officially
sign approved prc files. This initial setup therefore requires a bit
of back-and-forth building and rebuilding in the dtool directory.

To enable this feature, follow the following procedure.
(1) Decide how many different trust levels you require. You can have
    as many as you want, but most applications will require only one
    trust level, or possibly two. The rare application will require
    three or more. If you decide to use multiple trust levels, you
    can make a distinction between config variables that are somewhat
    sensitive and those that are highly sensitive.

(2) Obtain and install the OpenSSL library, if it is not already
    installed (http://www.openssl.org). Adjust your Config.pp file as
    necessary to point to the installed OpenSSL headers and libraries
    (in particular, define SSL_IPATH and SSL_LIBS), and then ppremake
    and make install your dtool tree. It is not necessary to build
    the panda tree or any subsequent trees yet.

(3) Set up a directory to hold the generated public keys. The
    contents of this directory must be accessible to anyone building
    Panda for your application; it also must have a lifetime at least
    as long as the lifetime of your application. It probably makes
    sense to make this directory part of your application's source
    tree. The contents of this directory will not be particularly
    sensitive and need not be kept any more secret than the rest of
    your application's source code.

(4) Set up a directory in a secure place to hold the generated private
    keys. The contents of this directory should be regarded as
    somewhat sensitive, and should not be available to more than a

manageable number of developers. It need not be accessible to people building Panda. However, this directory should have a lifetime as long as the lifetime of your application. Depending on your environment, it may or may not make sense to make this directory a part of your application's source tree; it can be the same directory as that chosen for (3), above.

(5) Run the program make-prc-key. This program generates the public and private key pairs for each of your trust levels. The following is an example:

make-prc-key -a <pubdir>/keys.cxx -b <privdir>/sign#.cxx 1 2

The output of make-prc-key will be compilable C++ source code. The first parameter, -a, specifies the name of the public key output file. This file will contain all of the public keys for the different trust levels, and will become part of the libdtool library. It is not particularly sensitive, and must be accessible to anyone who will be compiling dtool.

The second parameter, -b, specifies a collection of output files, one for each trust level. Each file can be compiled as a standalone program (that links with libdtool); the resulting program can then be used to sign any prc files with the corresponding trust level. The hash character '#' appearing in the filename will be filled in with the numeric trust level.

The remaining arguments to make-prc-key are the list of trust levels to generate key pairs for. In the example above, we are generating two key pairs, for trust level 1 and for trust level 2.

The program will prompt you to enter a pass phrase for each private key. This pass phrase is used to encrypt the private key as written into the output file, to reduce the sensitivity of the prc signing program (and its source code). The user of the signing program must re-enter this pass phrase in order to sign a prc file. You may specify a different pass phrase for each trust level, or you may use the -p "pass phrase" command-line option to provide the same pass phrase for all trust levels. If you do not want to use the pass phrase feature at all, use -p "", and keep the generated programs in a safe place.

(6) Modify your Config.pp file (for yourself, and for anyone else who will be building dtool for your application) to add the following line:

#define PRC_PUBLIC_KEYS_FILENAME <pubdir>/keys.cxx

Where <pubdir>/keys.cxx is the file named by -a, above.

Consider whether you want to enforce the trust level in the development environment. The default is to respect the trust level only when Panda is compiled for a release build, i.e. when OPTIMIZE is set to 4. You can redefine PRC_RESPECT_TRUST_LEVEL if you want to change this default behavior.

Re-run ppremake and then make install in dtool.

(7) Set up a Sources.pp file in your private key directory to compile the files named by -b against dtool. It should contain an entry something like these for each trust level:

```
#begin bin_target
  #define OTHER_LIBS dtool
  #define USE_PACKAGES ssl
  #define TARGET sign1
  #define SOURCES sign1.cxx
#end bin_target

#begin bin_target
  #define OTHER_LIBS dtool
  #define USE_PACKAGES ssl
  #define TARGET sign2
  #define SOURCES sign2.cxx
#end bin_target
```

(8) If your private key directory is not a part of your application source hierarchy (or your application does not use ppremake), create a Package.pp in the same directory to mark the root of a ppremake source tree. You can simply copy the Package.pp file from panda/Package.pp. You do not need to do this if your private key directory is already part of a ppremake-controlled source hierarchy.

(9) Run ppremake and then make install in the private key directory.
This will generate the programs sign1 and sign2 (or whatever you have named them). Distribute these programs to the appropriate people who have need to sign prc files, and tell them the pass phrases that you used to generate them.

(10) Build the rest of the Panda trees normally.

Advanced tip: if you follow the directions above, your sign1 and sign2 programs will require libdtool.dll at runtime, and may need to be recompiled from time to time if you get a new version of dtool. To avoid this, you can link these programs statically, so that they are completely standalone. This requires one more back-and-forth rebuilding of dtool:
(a) Put the following line in your Config.pp file:

```
#define LINK_ALL_STATIC 1
```

(b) Run ppremake and make clean install in dtool. Note that you must make clean. This will generate a static version of libdtool.lib.

(c) Run ppremake and make clean install in your private key directory, to recompile the sign programs against the new static libdtool.lib.

(d) Remove (or comment out) the LINK_ALL_STATIC line in your Config.pp file.

(e) Run ppremake and make clean install in dtool to restore the normal dynamic library, so that future builds of panda and the rest of your application will use the dynamic libdtool.dll properly.