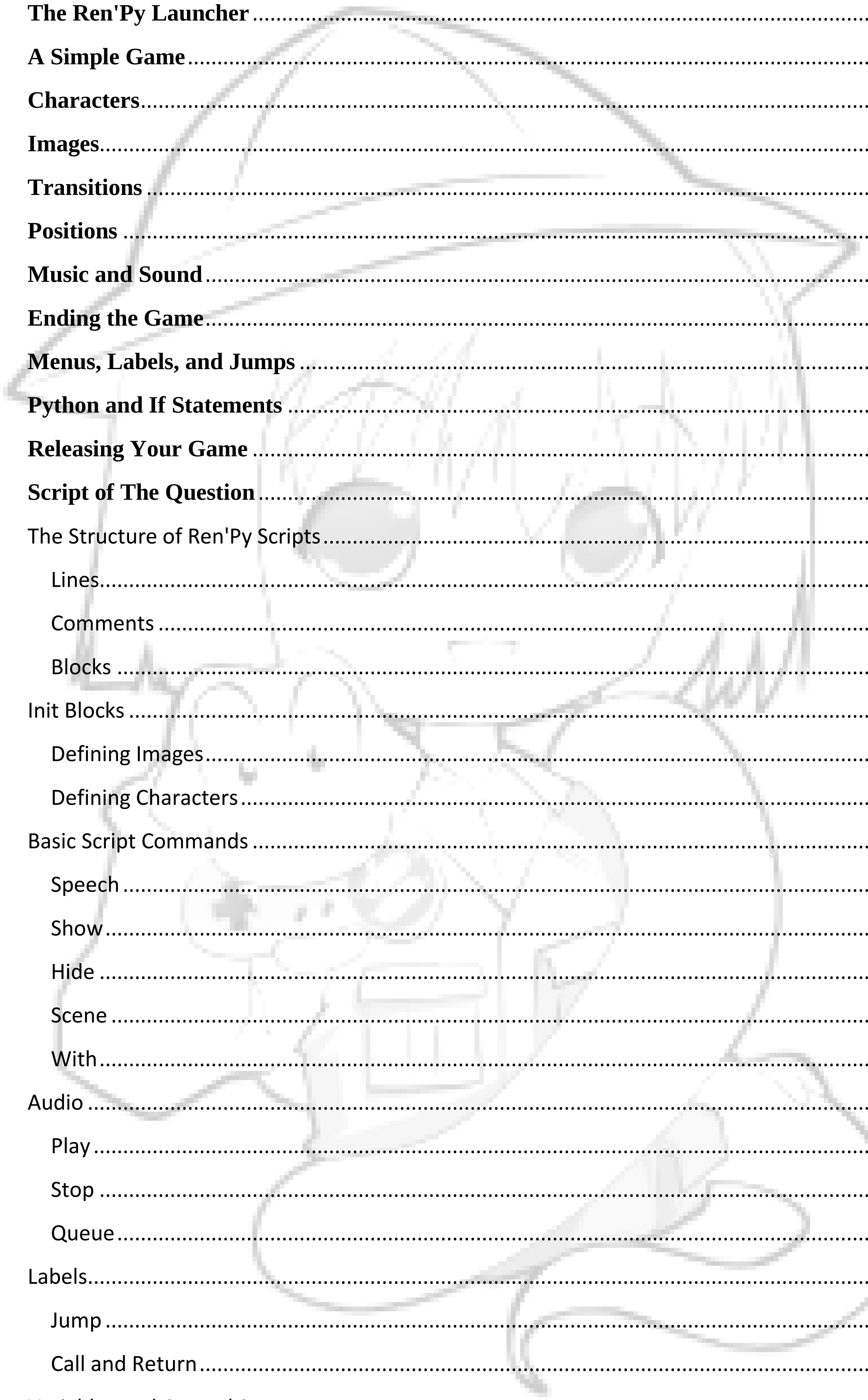
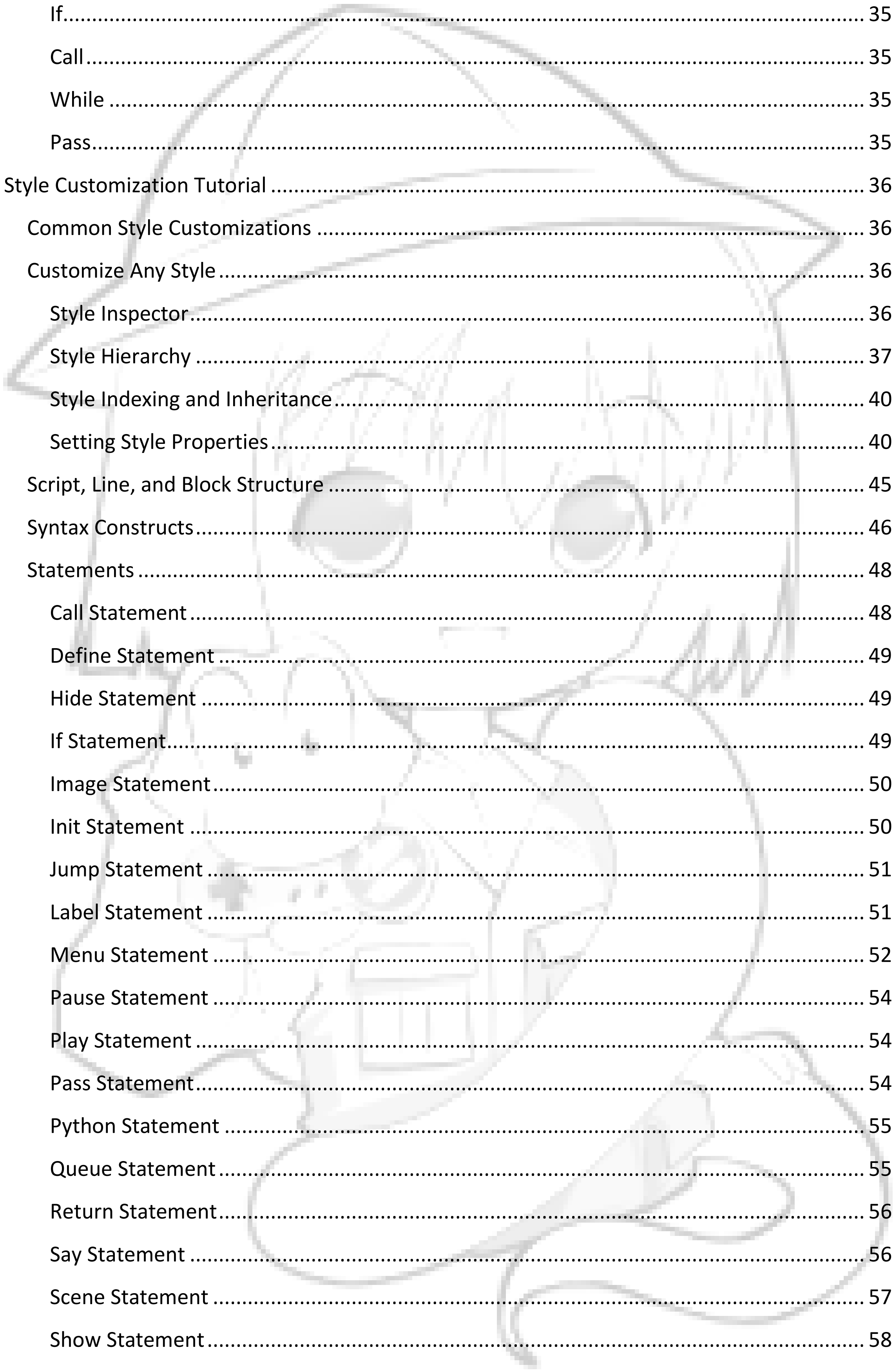
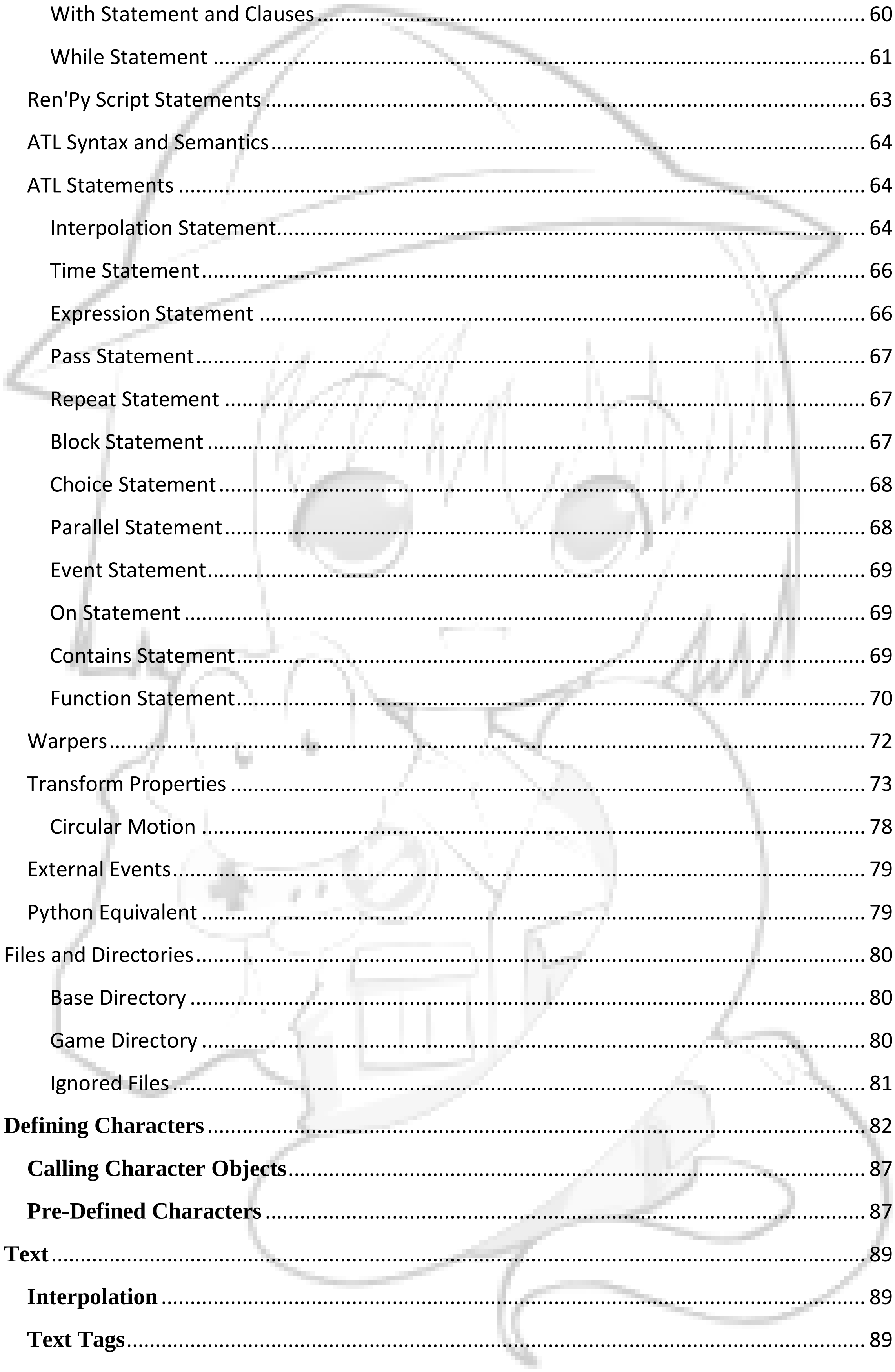
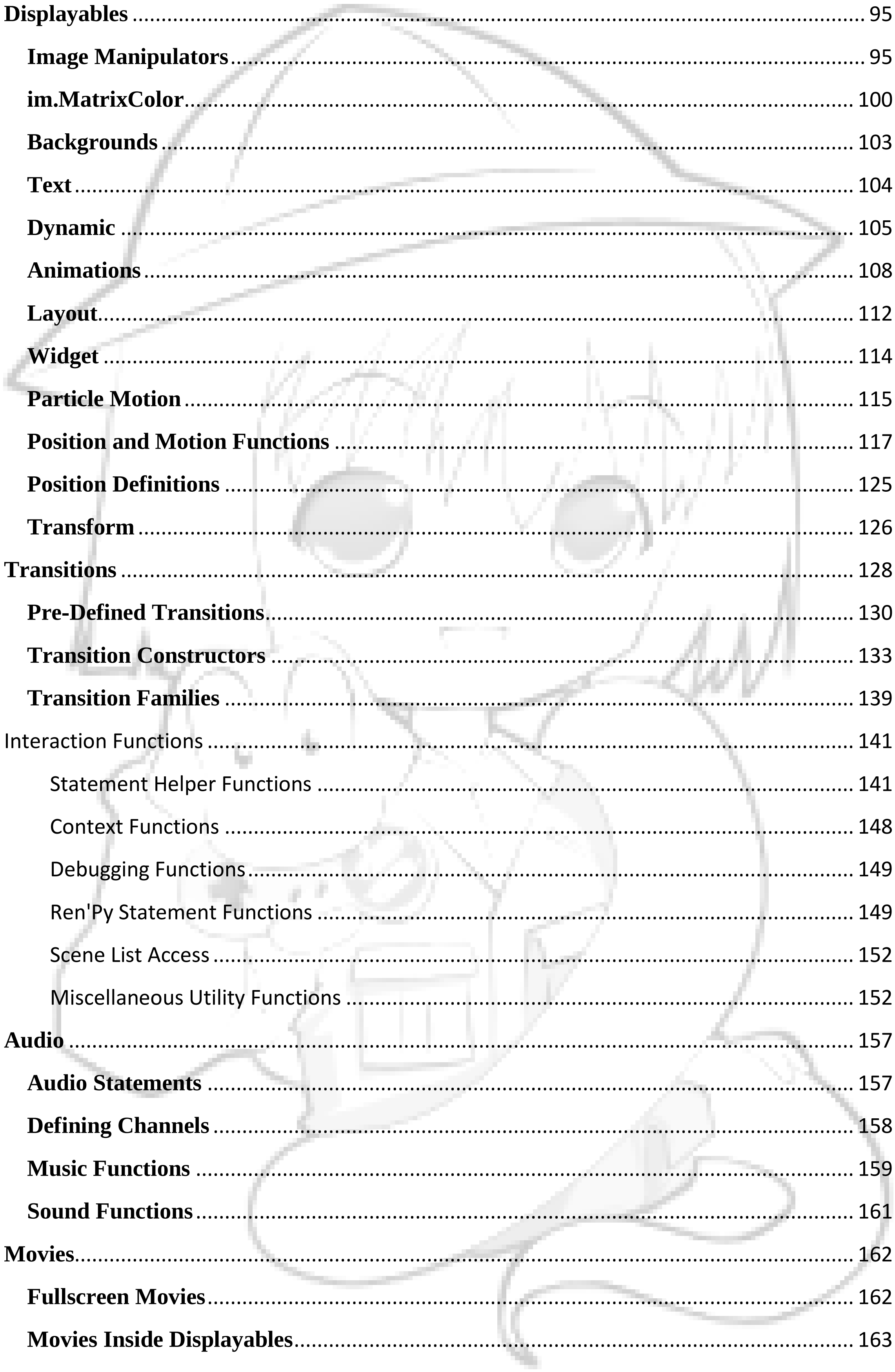
RENPY PROGRAMMING MANUAL

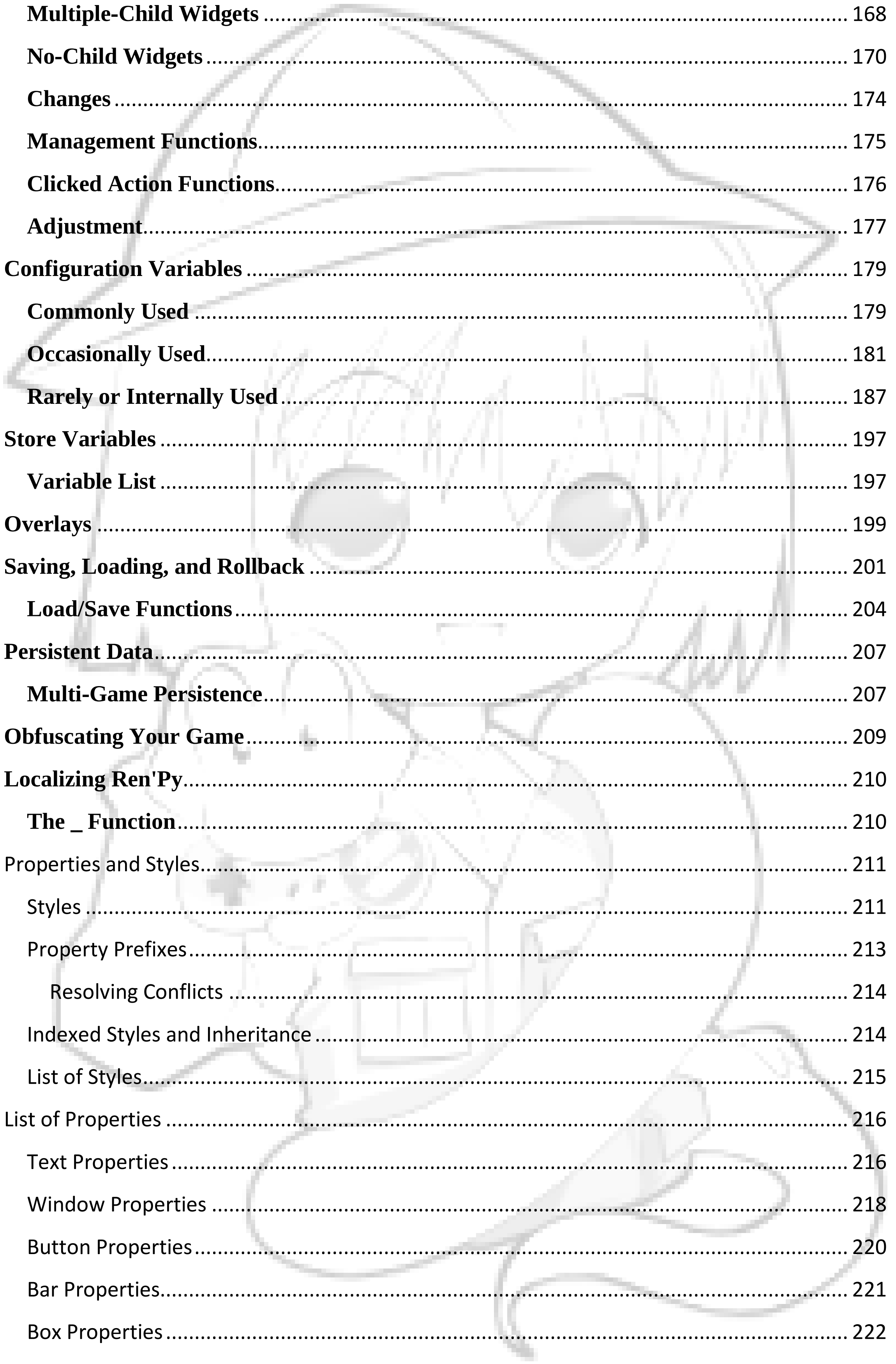
Contenido

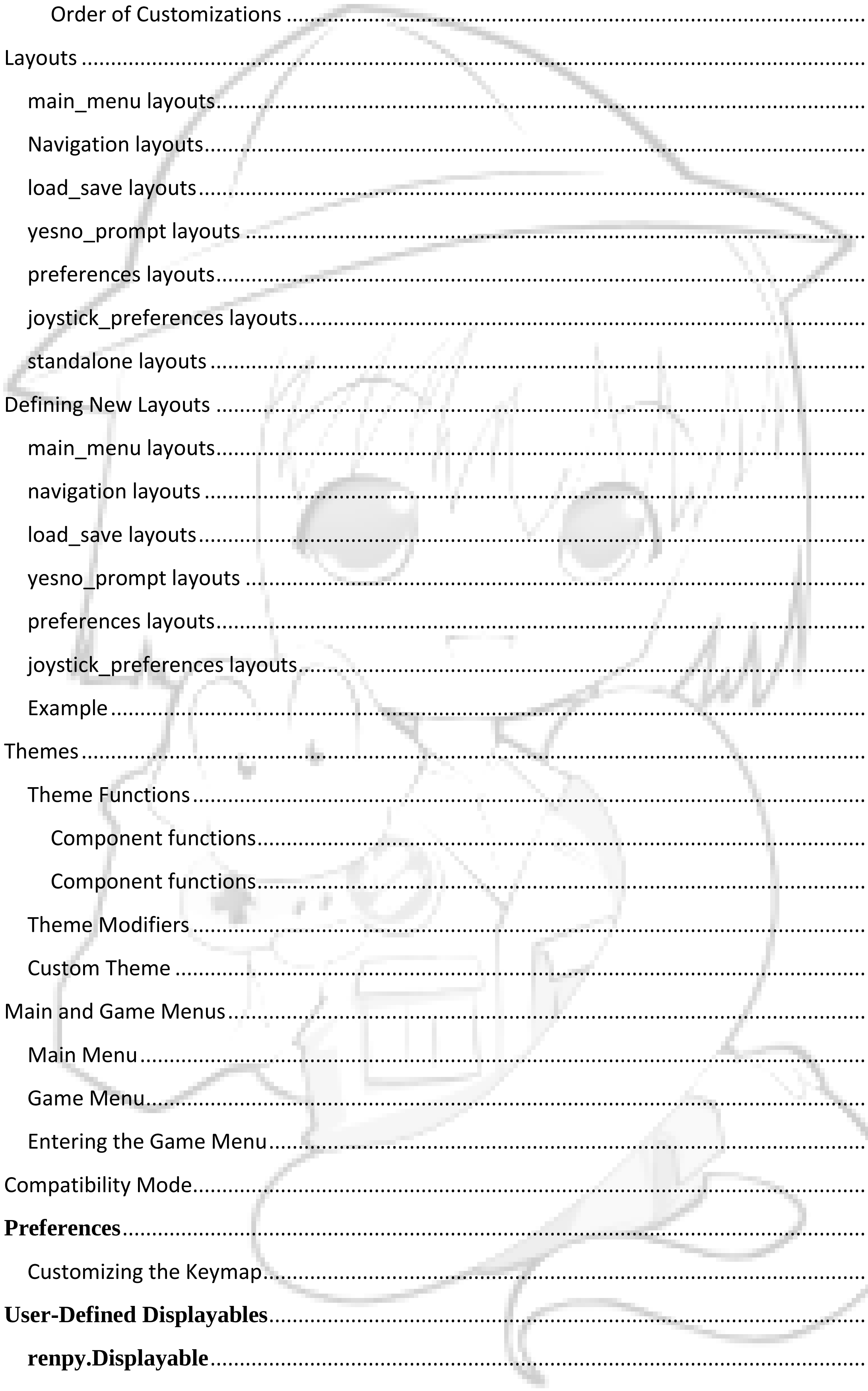


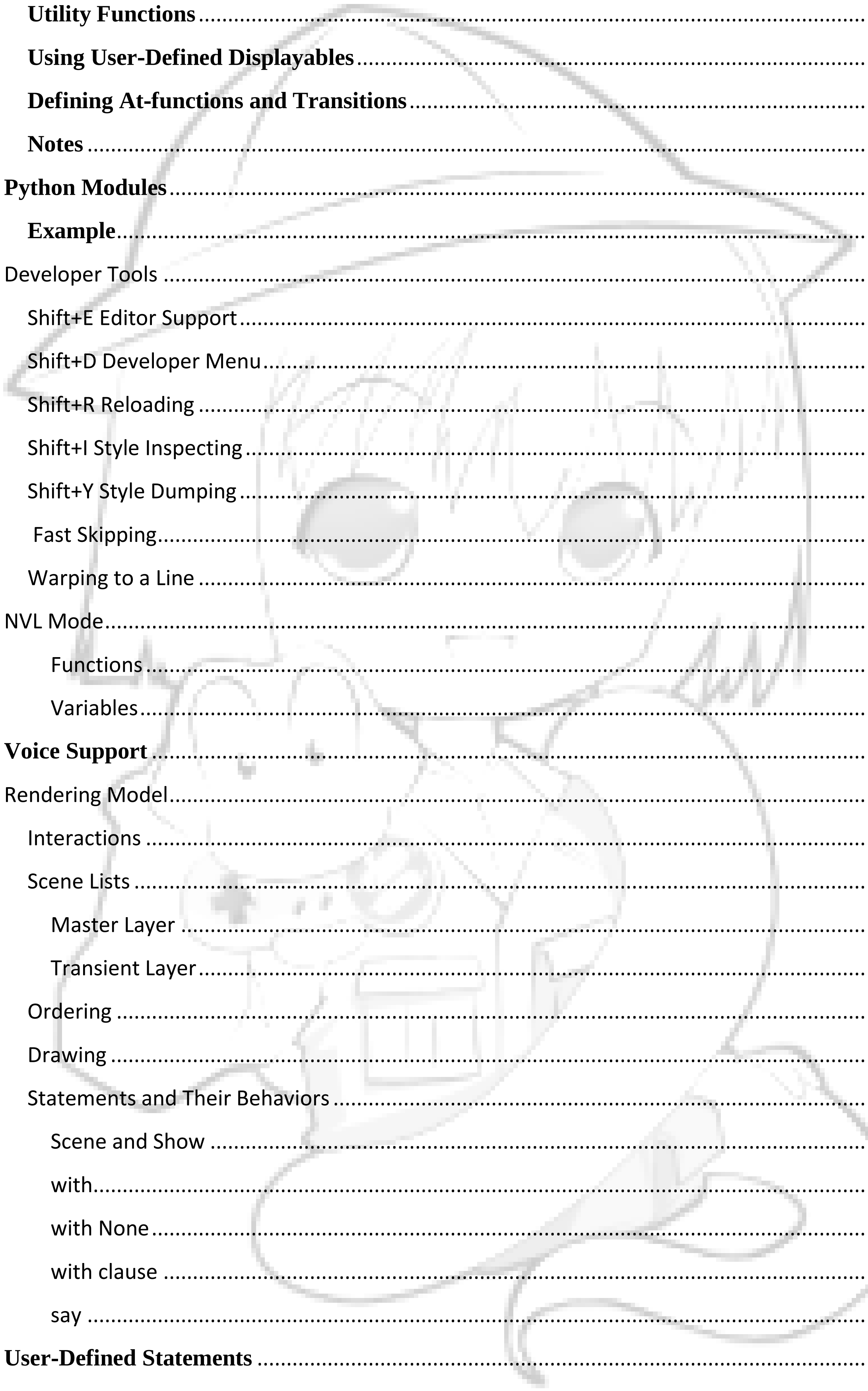












Welcome to the Ren'Py quickstart manual. The purpose of this manual is to demonstrate how you can make a Ren'Py game from scratch, in a few easy steps. We'll do this by showing how to make a simple game, *The Question*, from scratch. This manual contains a number of examples, which are included as part of the demo game.

The Ren'Py Launcher

Before you begin making a game, you should first take some time to learn how the Ren'Py launcher works. The launcher lets you create, manage, edit, and run Ren'Py projects.

Getting Started.

To get started you'll want to download Ren'Py and unzip it. You'll then want to start the launcher by running the renpy program.

Choosing a Project.

You should first see what the completed *The Question* game looks like. To do this, start the Ren'Py launcher, and choose "Select Project". A menu of projects will come up. Choose "the_question" from it. You'll be returned to the main menu, and you can now choose "Launch" to start *The Question*.

You can get back to the Ren'Py demo by doing the same thing, but choosing "demo" instead of "the_question".

Creating a new Project.

Create a new project by choosing "New Project" from the launcher. The launcher will ask you to choose a template. Choose "template". The launcher will then ask you for a project name. Since "the_question" is already taken, you should enter something different, like "my_question". The launcher will then ask you to choose a color theme for the project. It doesn't matter what you pick at this point, just choose something that appeals to you. You'll be returned to the top menu of the launcher with your new game chosen.

A Simple Game

```
label start:
    "I'll ask her..."

    "Me" "Um... will you..."
    "Me" "Will you be my artist for a visual novel?"
    "Silence."
    "She is shocked, and then..."

    "Sylvie" "Sure, but what is a \"visual novel?\""
```

This is perhaps one of the simplest Ren'Py games. It doesn't include any pictures or anything like that, but it does show a conversation between the two characters.

To try this out, go into the launcher, change to the "My Question" project, and pick "Edit Script". This will open the script files in a text editor. Choose the script.rpy file, and erase everything in it. We're starting from scratch, so you don't need what's there. Copy the example above into script.rpy, and save it.

You're now ready to run this example. Go back to the launcher, and click Run. Ren'Py will start up. Notice how, without any extra work, Ren'Py has given you menus that let you load and save the game, and change various preferences. When ready, click "Start Game", and play through this example game.

This example shows some of the commonly-used Ren'Py statements.

The first line is a label statement. The label statement is used to give a name to a place in the program. In this case, we create a label named "start". The start label is special, as it's where Ren'Py scripts begin running when the user clicks "Start Game" on the main menu.

The other lines are say statements. There are two forms of the say statement. The first is a string (beginning with a double-quote, containing characters, and ending with a double-quote) on a line by itself, which is used for narration, and the thoughts of the main character. The second form consists of two strings. It's used for dialogue, with the first string being a character name and the second being what that character is saying.

Note that all the say statements are indented by four spaces. This is because they are a block underneath the label statement. In Ren'Py, blocks must be indented relative to the prior statement, and all of the statements in a block must be indented by the same amount.

When strings contain double-quote characters, those characters need to be preceded by a backslash. This is done in the last line of our example.

While this simple game isn't much to look at, it's an example of how easy it is to get something working in Ren'Py. We'll add the pictures in a little bit, but first, let's see how to declare characters.

Characters

One problem with the first example is that it requires you to repeatedly type the name of a character each time they speak. In a dialogue-heavy game, this might be a lot of typing. Also, both character names are displayed in the same way, in fairly boring white text. To fix this, Ren'Py lets you define characters in advance. This lets you associate a short name with a character, and to change the color of the character's name.

```
define s = Character('Sylvie', color="#c8ffc8")
define m = Character('Me', color="#c8c8ff")

label start:
    "I'll ask her..."

    m "Um... will you..."
    m "Will you be my artist for a visual novel?"

    "Silence."
    "She is shocked, and then..."

    s "Sure, but what is a \"visual novel?\""
```

The first and and second lines define characters. The first line defines a character with the short name of "s", the long name "Sylvie", with a name that is shown in a greenish color. (The colors are red-green-blue hex triples, as used in web pages.)

The second line creates a character with a short name "m", a long name "Me", with the name shown in a reddish color. Other characters can be defined by copying one of the character lines, and changing the short name, long name, and color.

We've also changed the say statements to use character objects instead of a character name string. This tells Ren'Py to use the characters we defined in the init block.

Images

A visual novel isn't much of a visual novel without pictures. Let's add some pictures to our game.

```
image bg meadow = "meadow.jpg"
image bg uni = "uni.jpg"

image sylvie smile = "sylvie_smile.png"
image sylvie surprised = "sylvie_surprised.png"

define s = Character('Sylvie', color="#c8ffc8")
define m = Character('Me', color="#c8c8ff")

label start:
    scene bg meadow
    show sylvie smile

    "I'll ask her..."

    m "Um... will you..."
    m "Will you be my artist for a visual novel?"

    show sylvie surprised

    "Silence."
    "She is shocked, and then..."

    show sylvie smile

    s "Sure, but what is a \"visual novel?\""
```

The first new thing we needed to do was to declare the images, using image statements on lines 2, 3, 5, and 6, inside the init block. These image statements give an image name, and the filename the image is found in.

For example, line 5 declares an image named "sylvie smile", found in the filename "sylvie_smile.png", with the tag "sylvie".

We have a scene statement on line 12. This statement clears out the screen, and shows the "bg meadow" image. The next line is a show statement, which shows the "sylvie smile" image on the screen.

The first part of an image name is the image tag. If an image is being shown, and another image with the same tag is on the screen, then the image that's on the screen is replaced with the one being shown. This happens on line 19, the second show statement. Before line 19 is run, the image "sylvie smile" is on the screen. When line 19 is run, that image is replaces with "sylvie surprised", since they share the "sylvie" tag.

For Ren'Py to find the image files, they need to be placed in the game directory of the current project. The game directory can be found at "<Project-Name>/game/", or by clicking the "Game Directory" button in the launcher. You'll probably want to copy the image files from the "the_question/game/" directory into the "my_question/game/" directory, so you can run this example.

Ren'Py does not make any distinction between character and background art, as they're both treated as images. In general, character art needs to be transparent, which means it should be a PNG file. Background art can be JPEG or PNG files. By convention, background images start with the "bg" tag.

Hide Statement.

Ren'Py also supports a hide statement, which hides the given image.

```
    s "I'll get right on it!"

    hide sylvie

    "..."

    m "That wasn't what I meant!"
```

It's actually pretty rare that you'll need to use hide. Show can be used when a character is changing emotions, while scene is used when everyone leaves. You only need to use hide when a character leaves and the scene stays the same.

Transitions

Simply having pictures pop in and out is boring, so Ren'Py implements transitions that can make changes to the screen more interesting. Transitions change the screen from what it looked like at the end of the last interaction (dialogue, menu, or transition), to what it looks like after any scene, show, and hide statements.

```
label start:
    scene bg uni
    show sylvie smile

    s "Oh, hi, do we walk home together?"
    m "Yes..."
    "I said and my voice was already shaking."

    scene bg meadow
    with fade

    "We reached the meadows just outside our hometown."
    "Autumn was so beautiful here."
    "When we were children, we often played here."
    m "Hey... ummm..."

    show sylvie smile
    with dissolve

    "She turned to me and smiled."
    "I'll ask her..."
    m "Ummm... will you..."
    m "Will you be my artist for a visual novel?"
```

The with statement takes the name of a transition to use. The most common one is `dissolve` which dissolves from one screen to the next. Another useful transition is `fade` which fades the screen to black, and then fades in the new screen.

When a transition is placed after multiple scene, show, or hide statements, it applies to them all at once. If you were to write:

```
    scene bg meadow
    show sylvie smile
    with dissolve
```

Both the "bg meadow" and "sylvie smiles" would be dissolved in at the same time. To dissolve them in one at a time, you need to write two with statements:

```
    scene bg meadow
    with dissolve
    show sylvie smile
    with dissolve
```

This first dissolves in the meadow, and then dissolves in sylvie. If you wanted to instantly show the meadow, and then show sylvie, you could write:

```
    scene bg meadow
    with None
    show sylvie smile
    with dissolve
```

Here, None is used to indicate a special transition that updates Ren'Py's idea of what the prior screen was, without actually showing anything to the user.

Positions

By default, images are shown centered horizontally, and with their bottom edge touching the bottom of the screen. This is usually okay for backgrounds and single characters, but when showing more than one character on the screen it probably makes sense to do it at another position. It also might make sense to reposition a character for story purposes.

```
    show sylvie smile at right
```

To do this repositioning, add an at-clause to a show statement. The at clause takes a position, and shows the image at that position. Ren'Py includes several pre-defined positions: *left* for the right side of the screen, *right* for the right side, *center* for centered horizontally (the default), and *truecenter* for centered horizontally and vertically.

A user can define their own positions, but that's outside of the scope of this quickstart.

Music and Sound

Most games play music in the background. In Ren'Py, music files automatically loop until they are stopped by the user. Music is played with the play music statement.

```
    play music "illurock.ogg"
```

When changing music, one can supply a fadeout clause, which is used to fade out the old music when new music is played.

```
    play music "illurock.ogg" fadeout 1.0
```

Music can be stopped with the stop music statement, which can also optionally take a fadeout clause.

```
    stop music
```

Sound effects can be played with the play sound statement:

```
    play sound "effect.ogg"
```

Ren'Py support many formats for sound and music, but OGG Vorbis is preferred. Like image files, sound and music files must be placed in the game directory.

Ending the Game

You can end the game by running the return statement, without having called anything. Before doing this, it's best to put something in the game that indicates that the game is ending, and perhaps giving the user an ending number or ending name.

```
    ".:. Good Ending."

    return
```

That's all you need to make a kinetic novel, a game without any choices in it. Now, we'll look at what it takes to make a game that presents menus to the user.

Menus, Labels, and Jumps

The menu statement lets you present a choice to the user.

```
    s "Sure, but what's a \"visual novel?\""

menu:
    "It's a story with pictures.":
         jump vn

    "It's a hentai game.":
         jump hentai

label vn:
    m "It's a story with pictures and music."
    jump marry

label hentai:
    m "Why it's a game with lots of sex."
    jump marry

label marry:
    scene black
    with dissolve

    "--- years later ---"
```

This example shows how menus are used with Ren'Py. The menu statement introduces an in-game-menu. The menu statement takes a block of lines, each consisting of a string followed by a colon. These are the menu choices which are presented to the user. Each menu choice should be followed by a block of one or more Ren'Py statements. When a choice is chosen, the statements following it are run.

In our example, each menu choice runs a jump statement. The jump statement transfers control to a label defined using the label statement. The code following that label is run.

In our example above, after Sylvie asks her question, the user is presented with a menu containing two choices. If the user picks "It's a story with pictures.", the first jump statement is run, and control is transferred to the vn label. This will cause the pov character to say "It's a story with pictures and music.", after which control is transferred to the marry label.

Labels may be defined in any file that is in the game directory, and ends with .rpy. The filename doesn't matter to Ren'Py, only the labels contained within it. A label may only appear in a single file.

Python and If Statements

While simple (and even fairly complex) games can be made using only using menus and jump statements, after a point it becomes necessary to store the user's choices in variables, and access them again later. This is what Ren'Py's python support is for.

Python support can be accessed in two ways. A line beginning with a dollar-sign is a single-line python statement, while the keyword "python:" is used to introduce a block of python statements.

Python makes it easy to store flags in response to user input. Just initialize the flag at the start of the game:

```
label start:
    $ bl_game = False
```

You can then change the flag in code that is chosen by menus:

```
label hentai:

    $ bl_game = True
    m "Why it's a game with lots of sex."
    s "You mean, like a boy's love game?"
    s "I've always wanted to make one of those."
    s "I'll get right on it!"

    jump marry
```

And check it later:

```
    "And so, we became a visual novel creating team."
    "We made games and had a lot of fun making them."

    if bl_game:
        "Well, apart from that boy's love game she insisted on
making."

    "And one day..."
```

Of course, python variables need not be simple True/False values. They can be arbitrary python values. They can be used to store the player's name, to store a points score, or for any other purpose. Since Ren'Py includes the ability to use the full Python programming language, many things are possible.

Releasing Your Game

Once you've made a game, there are a number of things you should do before releasing it:

Edit options.rpy. The options.rpy file, created when you create a new game, contains a number of settings that you may want to customize. Some of them, like the screen height and screen width, should probably be set before making the game. Others, like the window title, can be set any time.

Add a plug for Ren'Py. This step is completely optional, but we do ask that if you have credits in your game, you mention Ren'Py in them. We suggest using something like "Made with the Ren'Py visual novel engine.", but that's just a suggestion, and what you write is up to you.

We think that the games people make are the best advertising for Ren'Py, and we hope that by including this, you'll help more people learn how to make visual novels in Ren'Py.

Create a README File. You'll probably want to include a README.txt file in the base directory of your project (that is, the directory above the game directory). We include a sample file with Ren'Py, which you can customize as necessary. You should also consider choosing a license for your game, one that determines how people may distribute it.

Check for a new version of Ren'Py. New versions of Ren'Py are released on a regular basis, to fix bugs and add new features. You should check the download page to see if a new version has come out. You may also want to see if any bug-fixes are available on that page.

Check the Script. In the Launcher, you should go to the Tools page, and pick "Check Script (Lint)". This will check your games for errors that may affect some users. These errors can affect users on the Mac and Linux platforms, so it's important to fix them all, even if you don't see them on your computer.

Build Distributions. From the Tools page, click distribute. The launcher will check your script again, ask you a few questions, and then build the distribution of your game.

Test. Lint is not a substitute for thorough testing. It's your responsibility to check your game before it is released.

Release. You should post the generated files (for Windows, Mac, and Linux) up on the web somewhere, and tell people where to download them from. Congratulations, you've released a game!

If you want, you can tell us about it at the Lemma Soft forums.

Script of The Question

You can view the full script of *The Question* here.

This page provides a simple overview of the Ren'Py language intended for non-programmers. There is also a more technical version of this page for more advanced users.

This simple version may not contain the same amount of detail, and may not be strictly accurate in a technical sense, but instead aims to give the reader an understanding of the common usage of the language features described, skipping exact detail for utility.

The Structure of Ren'Py Scripts

Lines

Ren'Py scripts consist of one or more .rpy files. These script files may be read in any order, and all of them together make up a Ren'Py script. When the game is started, Ren'Py will find the 'start' label and step through the lines one by one in order, doing whatever each line tells it to, until it runs out of script.

As Ren'Py runs through the script, it generally treats every line in the script file as a separate instruction; however, there are some exceptions:

- If you've opened a parenthesis and not closed it, then Ren'Py will keep reading on to the next line to find the closed parenthesis. Similarly, Ren'Py will match quotes enclosing text across lines. For example, the following are considered all one instruction:

```
    $ a = Character(
        'Abigail',
        color="#454580"
        )
    "I was disoriented,
    feeling like my thoughts
    were split up, and
    scattered somehow."
```

- If you want to split something across multiple lines, you can end one line with a backslash (\) and continue on the next line, e.g.:

```
    $ e = m * \
    c * c
```

is equivalent to:

```
    $ e = m * c * c
```

Comments

If you want to make notes within your script, then you can use comments. It's generally considered a good idea to write comments describing anything you think you'll need to remember - perhaps why a particular condition is checked, or what a variable is used for. To write a comment, simply type a hash (#). Ren'Py will ignore everything from that point until the end of the line, so you can write anything you like.

```
    # This line is a comment and will be ignored.
    # This line will be ignored too.
    "This line is going to be picked up by Ren'Py and used as
dialogue"
    "This is dialogue" # but this bit on the end is a comment.
```

Note that because Ren'Py's ignoring the comments, it's not checking for parentheses or quotes or backslashes on the end of the line - so comments can never be extended over more than one line in your script file without putting another hash in.

Blocks

Sometimes, a collection of lines need to be grouped together. This happens most frequently for labels, but also for conditional statements and loops and so on. Lines which are collected together are called 'blocks', and are written with an extra indentation in the script file:

```
line 1
    line a
    line b
line 2
    line c
    line d
```

In this example, here are three blocks. One block contains everything, the second block lines a and b, and the third contains lines c and d.

Note that lines a and b are said to be associated with line 1 - typically, line 1 will describe *why* lines a and b are in a block, e.g.:

```
    if (x < 10):
        # The lines in this block are only run if x is less than ten
        "Sam" "Oh no!"
        "Sam" "Ecks is less than ten!"
    # But these lines - outside of the if's block - are run regardless
of the value of x.
    "We wandered down the road a bit more, looking for anything else
as interesting as that ecks."
```

Note that lines which start a new block end with a colon (:). If a line ends with a colon, Ren'Py will expect to find a new block beneath it; if a line doesn't end in a colon, a new block shouldn't be started after that line.

The entire script will be divided up into label blocks and init blocks - these control the overall flow of the game; Ren'Py will always start at the 'start' label.

Init Blocks

Init blocks are used to define things which are going to be used again and again throughout your script. Typically, this includes images and Characters. It's generally a bad idea to define anything which has a changing state in an init block; it's best to stick to things which remain the same throught the game, like background images, character sprites and so on.

When the game first starts, all of the init blocks in all of the game scripts are run; if any init blocks are later encountered in the middle of the script they're ignored and skipped entirely.

An init block is started by the word 'init', followed by a colon, e.g.:

```
init:
    # code for init block goes here
    # more init code
```

The most common things to find within an init block are definitions of Images and Characters:

Defining Images

Images must be defined in an init block.

When we define an image we give a name to an image file by which we can refer to it later in the script - so when we want to show a character sprite or start a new scene in a different background, we can use the images that we defined earlier.

An image is defined by the command 'image', followed by a name for that image, then an equals sign, then the filename for the image file we wish to use. Place the image file in the game directory alongside the script, and Ren'Py will be able to find it.

```
init:
    image eileen = "eileen.png"
```

Note that you can give images names with spaces in - if you do this, Ren'Py pays special attention to the first word of the name (see the show and hide commands for more details) so it's wise to group your images by that first part, e.g.:

```
init:
    image eileen happy = "eileen-happy.png"
    image eileen sad = "eileen-sad.png"
```

Defining Characters

Also commonly defined in init blocks are Characters - a Character allows a quick and easy way to attach a name to dialogue, format text and so on, rather than having to type out the Character's full name and assign colours every time that Character speaks.

In its simplest form, one defines a character as follows:

```
init:
    $ l = Character("Lucy")
```

The dollar sign at the beginning of the line tells Ren'Py that this is a Python line. This is necessary for defining characters because there's no Ren'Py command to do it for us - Characters have too many options! The next part - 'l' in the example above - is the name by which we refer to that character in the rest of the script. The last part creates the Character itself; here we're just supplying the name which we want to be displayed when that character is used.

Creating a Character for each of the characters in your VN is not by any means *essential*, but at the very least it can help to save you typing the character name out each time they speak.

For further details on Characters and the many other options they have, see the article on Defining Characters.

Basic Script Commands

The following items are a list of common commands Ren'Py understands, along with a usage explanation and example for each. While some effort has been made to order the list sensibly, it's unavoidable that sometimes references to later in the list will be necessary.

Speech

Speech displays text onto the screen, then waits for the user to click the mouse button or hit space or enter before proceeding to the next instruction. Speech falls into two categories - narration or dialogue.

Narration is simplest - it consists of the text to be narrated, enclosed between quotes, e.g.:

```
    "It was a dark and stormy night"
```

Note that you can use either single or double quotes, but you must start and end the text with the same kind of quote. Of course, this means that you can include single-quotes in double-quoted text and vice-versa:

```
    "'Twas the night before Christmas"

    'I could tell it was going to be one of "those" days.'
```

Dialogue is used to have particular characters speak. Because the reader can't tell the difference between one character's voice and another's, when all their dialogue is just text, the name of the character speaking is displayed alongside the dialogue. Of course, this means that you have to specify the name of the character in the script, which is done by writing the character's name (in quotes) first, then the dialogue for that character immediately afterward, e.g.:

```
    "Hamlet" "Alas, poor Yorick! I knew him, Horatio, a fellow of
infinite jest, of most excellent fancy."
```

If you have defined a Character (in an init block), you can use that Character instead of the name when writing dialogue, e.g.:

```
init:
    $ e = Character("Eileen")

label start:
    e "Hello!"
```

Show

The show command is used for adding a sprite to a scene. The sprite must previously have been defined as an image in an init block, as the show command uses the image name.

In its most basic form, the show command consists of the word 'show', followed by the name of the image to show, e.g.:

```
init:
    image eileen = "eileen.png"

label start:
    show eileen
```

This will show the image named 'eileen' - the one from the file 'eileen.png' - in the bottom-centre of the screen.

When you show an image, Ren'Py will look for an image which is already in the scene which shares the same first part of the name - if such an image exists, Ren'Py will remove it before showing the new image. This means that - for example - you don't have to hide old sprites of the same character when you're changing an expression.

```
init:
    image eileen happy = "eileen-happy.png"
    image eileen sad = "eileen-sad.png"

label start:
    show eileen happy
    show eileen sad
```

When the second 'show' command is run, Ren'Py looks at the first part of the image name - in this case 'eileen' - and examines all the images already in the scene for an image with that tag. The previous 'eileen happy' also had 'eileen' for the first part of the image name, so this is removed from the scene before the second show takes place. This ensures that only one Eileen is on-screen at the same time.

Of course, it's not particularly useful to only be able to show things in the centre of the screen, so the show command can also be followed by an 'at' giving a position at which to place the new image in the scene:

```
    show eileen at left
```

This will show the 'eileen' image at the left of the screen. The positions that Ren'Py understands out of the box are as follows:

- left - at the left side, down against the bottom of the screen.
- right - at the right side, down against the bottom of the screen.
- center - in the middle, down against the bottom of the screen.
- truecenter - centred both horizontally and vertically.
- offscreenleft - just off the left side of the screen, down against the bottom.
- offscreenright - just off the right side of the screen, down against the bottom.

Alternately, you can define your own positions with the Position function

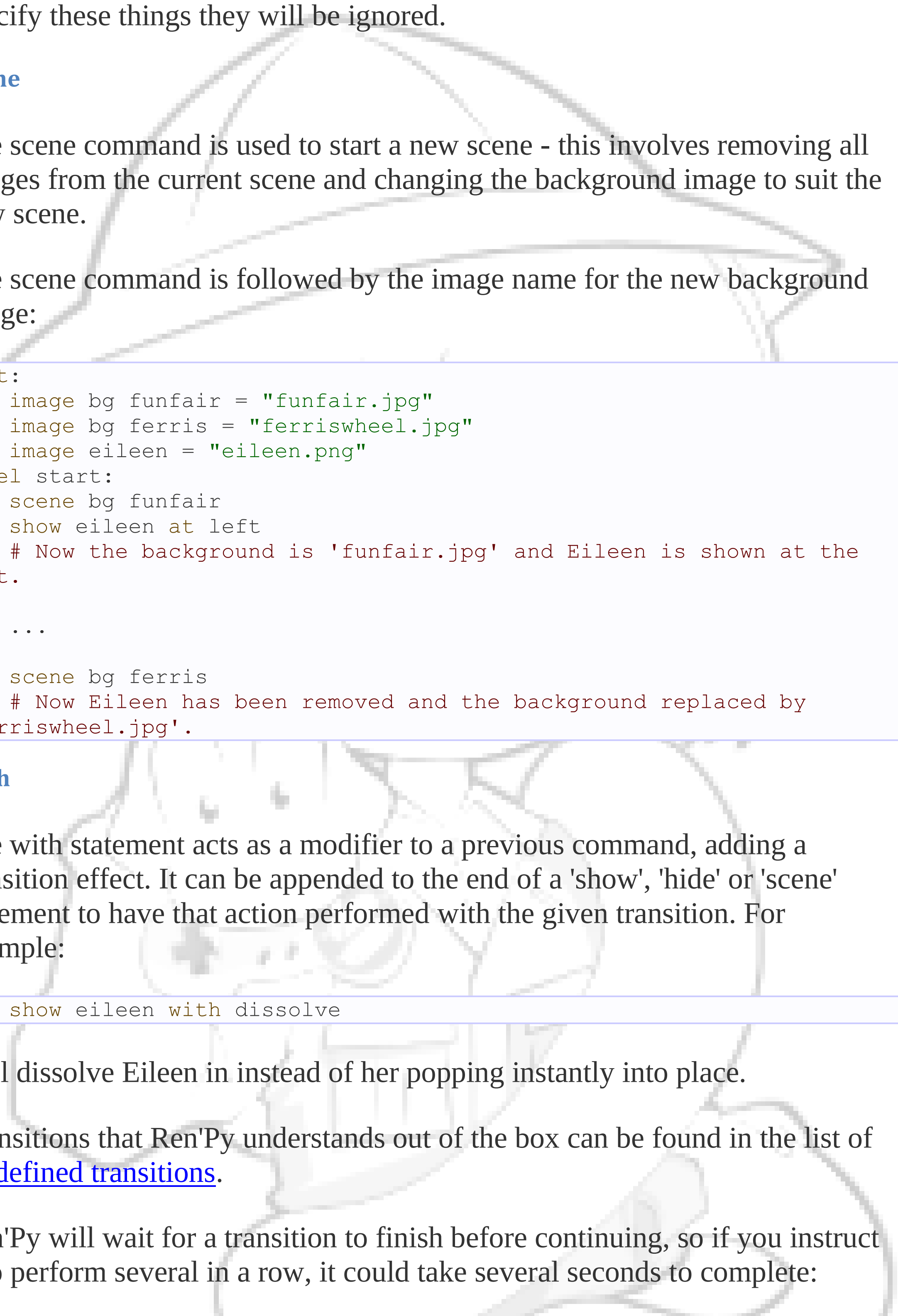
Hide

The hide command is the opposite of the show command - it removes an image from a scene.

The syntax is similar - simply the word 'hide' followed by the name of the image to hide:

```
    show eileen happy at left
    # now there is one character in the scene
    show lucy upset at right
    # now there are two characters in the scene
    hide eileen
    # now there is only one character remaining
    hide lucy
    # and now there are none
```

As you can see from the above example, only the first part of the image name is necessary - in the same way that the 'show' command will replace an image in the scene just based on the first part of the image name, equally the hide command only needs this much to find it and hide it. Also note that it is not necessary to specify the position of the character to hide. In fact, even if you specify these things they will be ignored.



Scene

The scene command is used to start a new scene - this involves removing all images from the current scene and changing the background image to suit the new scene.

The scene command is followed by the image name for the new background image:

```
init:
    image bg funfair = "funfair.jpg"
    image bg ferris = "ferriswheel.jpg"
    image eileen = "eileen.png"
label start:
    scene bg funfair
    show eileen at left
    # Now the background is 'funfair.jpg' and Eileen is shown at the
left.

    ...

    scene bg ferris
    # Now Eileen has been removed and the background replaced by
'ferriswheel.jpg'.
```

With

The with statement acts as a modifier to a previous command, adding a transition effect. It can be appended to the end of a 'show', 'hide' or 'scene' statement to have that action performed with the given transition. For example:

```
    show eileen with dissolve
```

Will dissolve Eileen in instead of her popping instantly into place.

Transitions that Ren'Py understands out of the box can be found in the list of predefined transitions.

Ren'Py will wait for a transition to finish before continuing, so if you instruct it to perform several in a row, it could take several seconds to complete:

```
    scene bg funfair with fade
    # Ren'Py waits for the funfair scene to fade in before...
    show eileen at left with dissolve
    # Ren'Py waits for Eileen to fully dissolve in before...
    hide eileen with moveoutleft
    # Ren'Py waits for Eileen to completely disappear off the side of
the screen before...
```

```
    "Who was that?"
```

Of course, sometimes this isn't desirable - you may wish for the scene to fade in, with a character sprite already displayed, in which case you need to stretch the with over more than one statement. If you place the with clause on a line on its own, then instead of being tied to one specific show, hide or scene, it will apply to every such command since the last dialogue or menu - or the last statement which did have its own with clause.

```
    scene bg funfair
    show eileen at left
    show lucy at right
    with fade
```

In the above example, whatever the previous scene was will fade to black, then a scene containing eileen and lucy in front of the 'bg funfair' background will fade in all as one single transition.

```
    scene bg funfair with fade
    show eileen at left
    show lucy at right
    with dissolve
```

In this example, however, the scene command had its own 'with' clause, so the 'with dissolve' at the end only applies to the two 'show' commands. So the previous scene will fade to black, then the new scene at the funfair will fade in, and as soon as that has finished Eileen and Lucy will both dissolve in at the same time.

Audio

Play

The play statement is used for playing sound effects or music tracks.

To play music, follow the play command with the word 'music', then the filename of the music file you wish to play:

```
play music "entertainer.ogg"
```

To play a sound, follow the play command with the word 'sound', then the filename of the sound file you wish to play:

```
play sound "explosion.ogg"
```

When you play music, Ren'Py will loop the music, playing it over and over again; when you play a sound, Ren'Py will play it only once.

Channels

Sound output is divided into 'channels'. You can only be playing one piece of audio - sound or music - on a single channel at once. If you want to play two sounds simultaneously, you will need to specify that the second one is played on a different channel - this can be done by adding 'channel' and then the number of the channel you wish to use to the end of the command:

```
play sound "explosion.ogg" channel 0
play sound "yeehah.ogg" channel 1
```

By default, sounds are played on channel 0, and music is played on channel 7. Of course, this means that if you are only playing one sound at a time, you don't need to worry about channels - sounds can be played over music just fine.

Fade

By default, Ren'Py will stop any audio you already have playing on a channel if you play something new; if you want to specify a fadeout for an already-playing music or sound effect, then add 'fadeout' followed by a time in seconds to fade for to the end of the command.

```
play music "waltz.ogg" fadeout 1
```

Similarly, if you want to fade the new piece of music in, then use the 'fadein' keyword. One can combine any or all of these keywords in the same command:

```
play music "waltz.ogg" fadeout 1 fadein 2
```

The above example will fade whatever music is currently playing out over the course of one second, and fade in "waltz.ogg" over the course of two seconds.

Playlists

Lastly, if you would prefer to give several pieces of music or sounds to be played one after another, you can give the play command a list of files instead of just a single filename. To use a list, use square brackets ([,]) around a list of filenames separated by commas.

```
play music ["waltz.ogg", "tango.ogg", "rumba.ogg"]
```

The above example will play "waltz.ogg", then once it is finished play "tango.ogg", then "rumba.ogg", then start again at the beginning with "waltz.ogg".

Formats

Ren'Py will play files in the following formats:

- Ogg Vorbis (.ogg)
- PCM Wave (.wav)
- FLAC (.flac)
- Shorten (.shn)
- MP3 (.mp3)
- Various mod formats

Audio files should be saved at 44100Hz to avoid crackle.

Stop

Stop is the opposite of the play command; it stops whatever audio is currently playing. Similarly to play, you can choose from either 'stop sound' to stop a sound effect, or 'stop music' to stop playing music.

```
stop sound
```

Similarly to play, you can specify a fadeout for the audio you're stopping by adding the word 'fadeout' followed by a time in seconds.

```
stop music fadeout 1.5
```

The above example fades the currently-playing music out over the course of one and a half seconds.

You can also stop the audio that's playing on a particular channel. Again, this is done in a similar manner to the play command - simply append 'channel' and the channel number you wish to stop after the stop command, e.g.:

```
stop sound channel 2
```

Since you are only stopping audio which has already been played, the default channels are the same - channel 0 for sound effects and channel 7 for music.

Queue

The queue command queues up music files or sound effects to play after the currently-playing audio has finished. Ren'Py will queue the file you specify to play immediately after the currently-playing music has finished.

The simple usage of the command is similar to the play command - the queue command is followed by one of 'sound' or 'music' depending on whether you want to queue a music track or a sound effect, then the filename of the audio file you wish to queue:

```
queue music "waltz.ogg"
```

If you call the queue command a second time before the first piece of music finishes, your newly-queued file will *replace* the existing queued item and be played as soon as the currently-playing audio has finished.

```
play music "waltz.ogg"
queue music "tango.ogg"

"There's a waltz playing in the background..."
queue music "rumba.ogg"
```

In the above example, first "waltz.ogg" is started playing, and then "tango.ogg" is queued. Then, the game displays some dialogue and waits for the user to click to continue.

- If the user clicks before the waltz finishes playing, then "rumba.ogg" is queued - this replaces the already-queued tango and after the waltz has finished, the rumba will follow.
- If the user waits for the waltz to finish before continuing, then the previously-queued tango will play immediately after the waltz, and when the rumba is queued it will be played immediately after the tango finishes.

In either scenario, once the rumba finishes it will continue to loop just the rumba.

If you wish to queue up more than one file at once, then you can supply a list of file names, separated by commas, inside square brackets ([,]) instead of a single file name.

```
play music "rumba.ogg"
queue music ["waltz.ogg", "tango.ogg"]
```

In this example, the rumba plays. Two files are added to the queue, so when the rumba finishes, it will play the waltz next; when the waltz finishes, it will play the tango. When the tango finishes, the entire queue will loop, so the waltz will play, then the tango, then the waltz again, then the tango again, and so on.

Similarly to the play command, you can specify a channel on which to queue the audio, by adding 'channel' followed by the channel number to the end of the command.

```
queue sound "explosion.ogg" channel 2
```

Note that you cannot queue music to a channel which is currently playing sound effects, or sound effects to a channel that is currently playing music. Each channel has its own queue, so you can queue - for example - music on the default channel and sound effects on the default channel and both of your queued items will be played independently.

Labels

Labels are used to divide a Ren'Py script up into sections, and also to allow a non-linear progression through those sections.

Generally, Ren'Py will run each line of the script in the order it finds them, so by way of example:

```
"This line is run first"

"This line is run second"

"This line is run third"

# and so on...
```

However, in order to write a branching story, or perhaps a story in which some parts are repeated, we use labels. Labels contain blocks of Ren'Py script, and consist of the word 'label' followed by a name, followed by a colon(:) - which denotes that it contains a block.

```
label start:
    "This line is under the 'start' label."
    "As is this one."
label next:
    "But this one is under the label 'next'."
label last:
    "And this one is under 'last'."
```

Label names have to be unique across your entire script, because Ren'Py uses them to identify that particular place in the script - you can't have two labels with the same name, because Ren'Py wouldn't be able to tell those two label blocks apart. Label names cannot have spaces in, so it's common to use underscores to separate words in a label name:

```
label not valid:
    # The preceeding label won't work, because it has spaces in the
name.
```

```
label perfectly_valid:
    # The label 'perfectly_valid', on the other hand, is fine.
```

When Ren'Py reaches the end of a label block - for example the line "As is this one." in the above example - it simply continues on to the following label block, so in the example above the four lines are delivered in the order they're written, one after another, as if the label statements weren't there at all. Using the jump and call commands, however, we can move between labels in a non-linear fashion.

Jump

The jump command instructs Ren'Py to skip immediately to the label which is given, ignoring all of the script in between the present position and that label. Jumps can be forward or backward through the file, or even in a different script file in the same game directory.

The jump command consists of the word 'jump' followed by the name of the label to jump to:

```
jump some_label
```

Because the label could be anywhere, the jump command allows us to write scripts in a non-linear order, for example:

```
label start:
    "This line of dialogue will be spoken first."

    jump stage_two

label stage_three:
    "This line will be spoken third."

    jump end

    "This line will never be run."

label stage_two:
    "This line will be spoken second."

    jump stage_three

label end:
    "And this line will be last."
```

Ren'Py cannot execute the line in the middle ("This line will never be run") because it's impossible to step line-after-line from a label to that dialogue line. The 'jump end' command will get in the way and re-direct Ren'Py to the 'end' label before it ever gets to that line of dialogue.

Call and Return

Call instructs Ren'Py to go off and run the code at another label, and then return and carry on from this point afterwards.

The call command consists of the word 'call', followed by the name of the label to call:

```
call subroutine
```

At the other end, the command 'return' tells Ren'Py to go back to the point of the call:

```
label subroutine:
    "Do some stuff."

    return
```

This means that if there is a part of your script which you want to re-use at multiple points, then you can use call to visit that part from anywhere else in your script and return to those different places without having to know each of their label names.

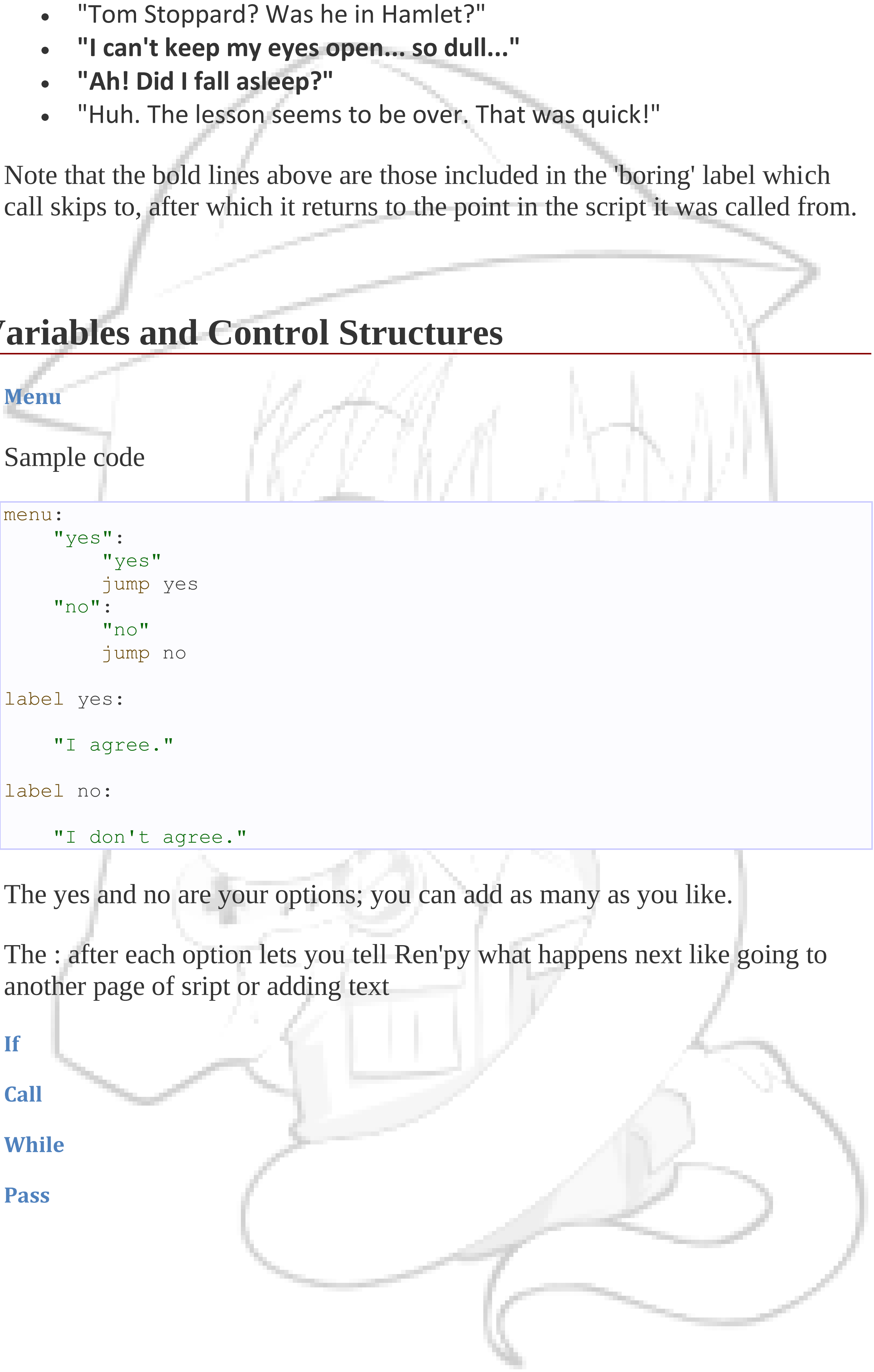
```
label start:
    "The teacher starts droning on about Hamlet..."

    call boring

    "I think I missed something. Now she's talking about Rozencrantz
and... someone..."
    "Tom Stoppard? Was he in Hamlet?"

    call boring

    jump end

label boring:
    "I can't keep my eyes open... so dull..."
    "Ah! Did I fall asleep?"

    return

label end:
    "Huh. The lesson seems to be over. That was quick!"
```

This will produce the following dialogue:

- "The teacher starts droning on about Hamlet..."
- **"I can't keep my eyes open... so dull..."**
- **"Ah! Did I fall asleep?"**
- "I think I missed something. Now she's talking about Rozencrantz and... someone..."

- "Tom Stoppard? Was he in Hamlet?"
- **"I can't keep my eyes open... so dull..."**
- **"Ah! Did I fall asleep?"**
- "Huh. The lesson seems to be over. That was quick!"

Note that the bold lines above are those included in the 'boring' label which call skips to, after which it returns to the point in the script it was called from.



Variables and Control Structures

Menu

Sample code

```
menu:
    "yes":
        "yes"
        jump yes
    "no":
        "no"
        jump no

label yes:

    "I agree."

label no:

    "I don't agree."
```

The yes and no are your options; you can add as many as you like.

The : after each option lets you tell Ren'py what happens next like going to another page of sript or adding text

If

Call

While

Pass

Style Customization Tutorial

By default, Ren'Py games are made with a usable but relatively generic style. The purpose of this tutorial is to teach you how to customize the Ren'Py style system to make your game more attractive.

Before we start, we should note that the style system is only one way to customize your game, and may not be able to accomplish all customizations.

- Layouts define the basic look of the various main and game menu screens, such as what buttons and other displayables are present. There are several built-in layouts, including imagemap-based layouts, and it's possible to write your own.
- Themes handle the largest changes to the style system, defining consistent looks along many styles.
- Once you have chosen your layout and theme, you can further tweak the look of your game using the style system.

This tutorial is divided into two parts. The first part gives canned recipes for the most common style customizations. The second part explains how to customize any style and style property in the system, using the style inspector and style tree tools.

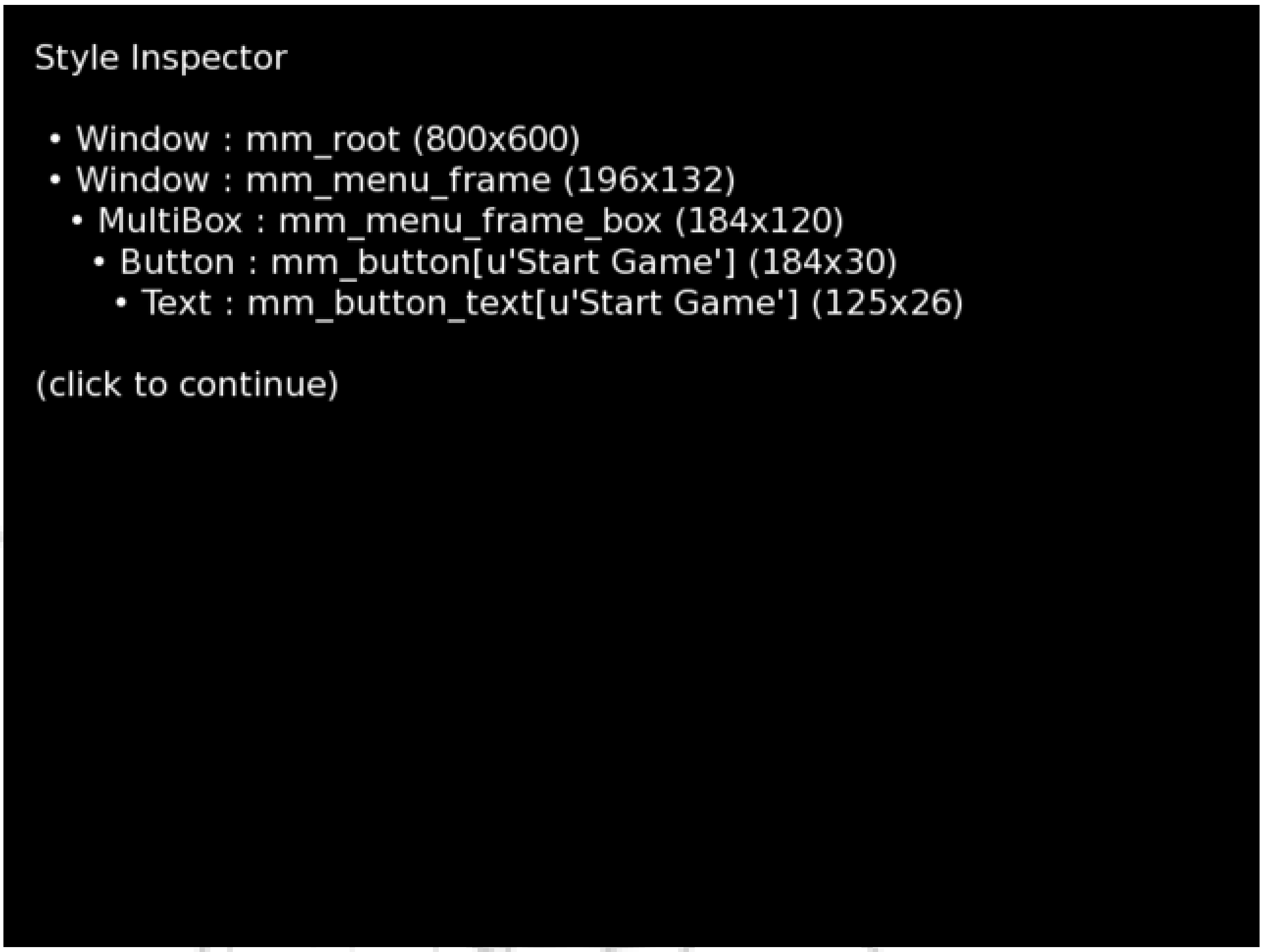
Common Style Customizations

Customize Any Style

The first thing one needs to do to when customizing styles is to determine which style it is you really want to customize. The easiest way to do this is to use the style inspector. It's also possible to use the style heirarchy tool to determine styles.

Style Inspector

To use the style inspector, place the mouse over any element in the game, and press `shift+I`. If nothing happens, then set config.developer to True, and restart your game. Otherwise, a screen similar to the following will pop up, showing the styles and Displayables underneath the mouse.



```
Style Inspector

 • Window : mm_root (800x600)
 • Window : mm_menu_frame (196x132)
   • MultiBox : mm_menu_frame_box (184x120)
     • Button : mm_button[u'Start Game'] (184x30)
       • Text : mm_button_text[u'Start Game'] (125x26)

(click to continue)
```

This screenshot was created by hovering over the "Start Game" button on the demo game main menu, and pressing `shift+I`.

Each line contains the following three fields:

1. The kind of displayable that is being shown.
2. The name of the style that is being used by that displayable.
3. The size of the displayable.

The lines are arrange in the order that things are drawn on the screen, so that the first line is the farthest from the user, and the last line is closest to the user. Indentation is used to represent nesting of displayables. Such nesting may affect the positioning of the various displayables.

Style Hierarchy

Styles may inherit from each other. The precise nature of style inheritance varies based on the layouts and themes that are being used. For a full accounting of styles the system knows about, hit `shift+D` and choose "Style Hierarchy". This will display a list of styles that the system knows about.

The styles that can be expected by all games to exist are:

- **style.default** (Everything) — The root of the style hierarchy. It must have a value for all style properties, and all properties take their default value from here.
    - **style.bar** (Bar) — Horizontal bars intended to convey information without allowing the user to adjust their value.
    - **style.button** (Button) — Normal sized buttons.
        - **style.small_button** (Button) — Smaller than normal buttons.
    - **style.button_text** (Text) — Text inside buttons.
        - **style.small_button_text** (Text) — Text inside smaller than normal buttons.
    - **style.centered_text** (Text) — Text displayed by the pre-defined centered character.
    - **style.centered_window** (Window) — Window containing text displayed by the pre-defined centered character.
    - **style.frame** (Window) — Frames are windows intended to contain buttons, labels, and other ui components. Used by default by ui.frame.
        - **style.menu_frame** (Window) — Frames that are part of the main or game menus.
    - **style.gm_root** (Window) — An empty Window placed at the root of the game menu.
    - **style.hbox** (Box) — A box that lays out its children from left to right.
    - **style.hyperlink_text** (Text) — Hyperlinked text, using the {a} text tag.
    - **style.image_button** (Button) — ui.imagebuttons.
    - **style.input_prompt** (Text) — Prompt text used by renpy.input.
    - **style.input_text** (Input) — Input fields produced by renpy.input and ui.input.
    - **style.label** (Window) — Window surrounding the text of a label. Labels are small amounts of text used in the game menu.
    - **style.label_text** (Text) — The text of a label.
    - **style.large_button** (Button) — A large button, like those used for slots in the file picker.
    - **style.large_button_text** (Text) — Text used by large buttons.
    - **style.menu** (Box) — Box containing an in-game menu, like those shown using the menu statement.
    - **style.menu_caption** (Text) — Text of a caption used in an in-game menu.

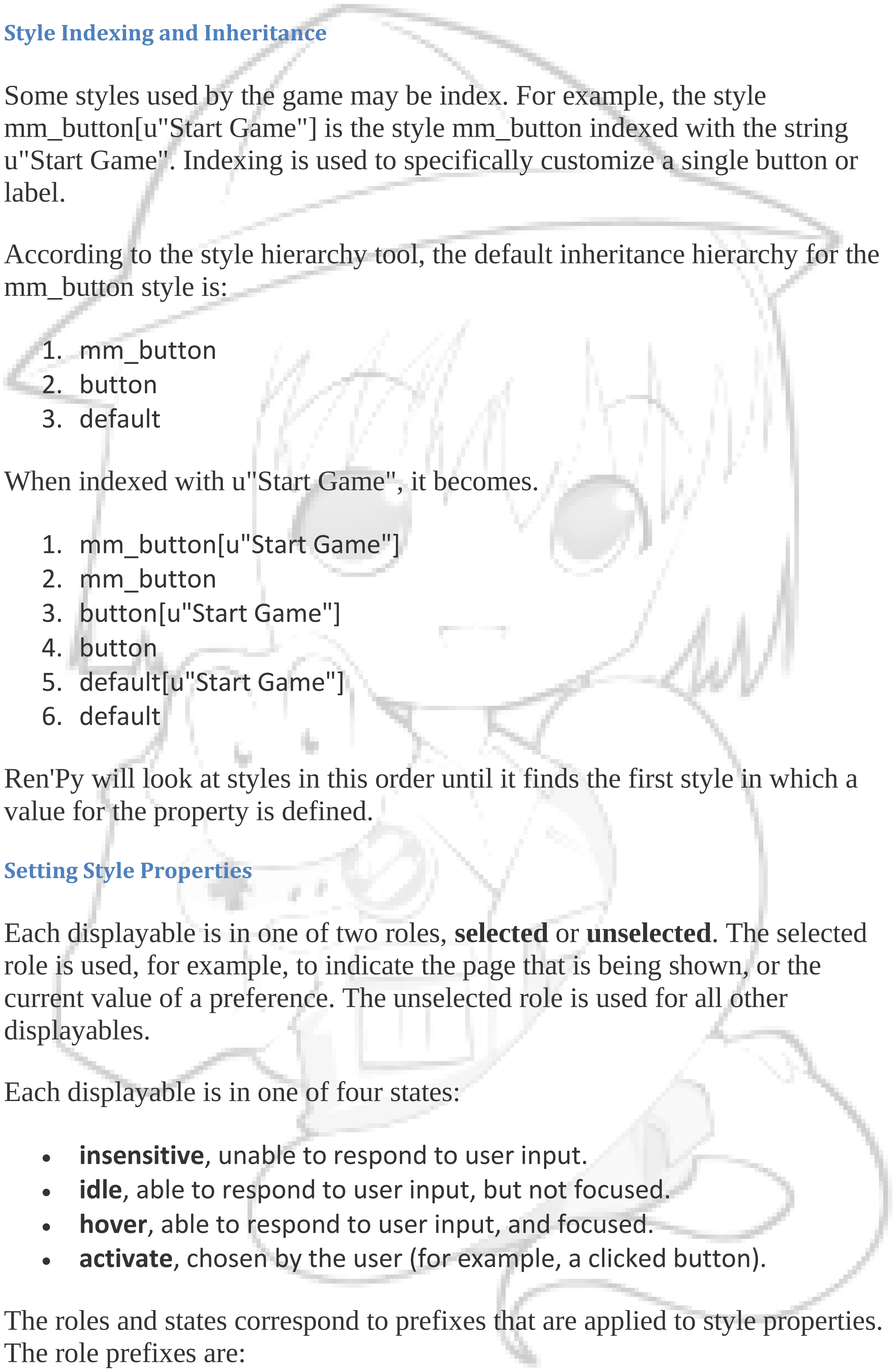
- **style.menu_choice** (Text) — Text of a menu choice.
    - **style.menu_choice_chosen** (Text) — Text of a menu choice that has been chosen in some earlier game.
- **style.menu_choice_button** (Button) — Button containing a menu choice.
    - **style.menu_choice_chosen_button** (Button) — Button containing a menu choice that has been chosen in some earlier game.
- **style.menu_window** (Window) — Window containing an in-game menu.
- **style.mm_root** (Window) — Empty window shown at the root of the main menu.
- **style.prompt** (Window) — A window containing prompt text. Prompt text is used in the game menu, and is generally longer than label text.
- **style.prompt_text** (Text) — Prompt text.
- **style.say_dialogue** (Text) — Dialogue in the say statement.
- **style.say_label** (Text) — The name of the character speaking in the say statement.
- **style.say_thought** (Text) — Dialogue spoken by the default `narrator`.
- **style.scrollbar** (Bar) — Horizontal scrollbars thay may or may not be intended to be adjusted by the user.
- **style.slider** (Bar) — Horizontal bars that are intended to be adjusted by the user.
- **style.vbar** (Bar) — Vertical bars intended to convey information without allowing the user to adjust their value.
- **style.vbox** (Box) — A box that lays out its children from top to bottom.
- **style.vscrollbar** (Bar) — Vertical scrollbars thay may or may not be intended to be adjusted by the user.
- **style.vslider** (Bar) — Vertical bars that are intended to be adjusted by the user.
- **style.window** (Window) — Window that's used to show dialogue and other in-game text.

layout.button_menu

Invoking layout.button_menu, as the default game template does, makes a couple of changes to the default style hierarchy.

- **menu_choice** is reparented to **button_text**.
- **menu_choice_button** is reparented to **button**.

This makes menu choices look like buttons.



Style Indexing and Inheritance

Some styles used by the game may be index. For example, the style mm_button[u"Start Game"] is the style mm_button indexed with the string u"Start Game". Indexing is used to specifically customize a single button or label.

According to the style hierarchy tool, the default inheritance hierarchy for the mm_button style is:

1. mm_button
2. button
3. default

When indexed with u"Start Game", it becomes.

1. mm_button[u"Start Game"]
2. mm_button
3. button[u"Start Game"]
4. button
5. default[u"Start Game"]
6. default

Ren'Py will look at styles in this order until it finds the first style in which a value for the property is defined.

Setting Style Properties

Each displayable is in one of two roles, **selected** or **unselected**. The selected role is used, for example, to indicate the page that is being shown, or the current value of a preference. The unselected role is used for all other displayables.

Each displayable is in one of four states:

- **insensitive**, unable to respond to user input.
- **idle**, able to respond to user input, but not focused.
- **hover**, able to respond to user input, and focused.
- **activate**, chosen by the user (for example, a clicked button).

The roles and states correspond to prefixes that are applied to style properties. The role prefixes are:

- "" (no prefix) - set for both selected and unselected roles.
- "selected_" - set for only the selected role.

The state prefixes are:

- "" (no prefix) - set for all states.
- "insensitive_" - set for the insensitive state.
- "idle_" - set for the idle state.
- "hover_" - set for the hover and activate states.
- "activate_" - set for the activate state.

To set a property, one assigns a value to the python expression:

style.*<style name>*.*<role prefix><state prefix><property>*

For example:

```
init python:
    style.mm_button.background = "#f00"
    style.mm_button.hover_background = "#ff0"
    style.mm_button.selected_hover_background = "#00f"
    style.mm_button[u"Start Game"].background = "#0f0"
```

The first line sets the background of all main menu buttons to be red. The second changes the background of hovered main menu buttons to be yellow. The third changes selected and hovered main menu buttons to be blue - this doesn't actually do anything, since main menu buttons are never selected. Finally, the last one changes the index style to have a green background. Since the indexed style is hire in the inheritance order, it will take precedence over all other styles.

Note that the order of statements matters. The code:

```
init python:
    style.mm_button.hover_background = "#ff0"
    style.mm_button.background = "#f00"
```

Will cause hovered main menu buttons to have a red background. This is because the second statement, which sets all roles and states, takes precedence over the first statement. Generally, it makes sense to set the unprefixed properties first, and the prefixed properties second.

A Simple Ren'Py Script

The following is a simple but complete Ren'Py script. The colors are added to make it easier to read, and aren't part of the script proper.

```
image bg whitehouse = "whitehouse.jpg"
image eileen happy = "eileen_happy.png"
image eileen upset = "eileen_upset.png"
define e = Character('Eileen')

label start:

    scene bg whitehouse
    show eileen happy

    e "I'm standing in front of the White House."

    show eileen upset

    e "I once wanted to go on a tour of the West Wing, but you have to
       know somebody to get in."

    "For some reason, she really seems upset about this."

    e "I considered sneaking in, but that probably isn't a good idea."
```

The beginning of this script consists of declarations of resources used throughout the game. First, three images are defined:

```
image bg whitehouse = "whitehouse.jpg"
image eileen happy = "eileen_happy.png"
image eileen upset = "eileen_upset.png"
```

These definitions create what in Ren'Py terminology are called Displayables, which can then be referred to in other script statements, such as `show`, by name. It's worth noting that the names must be valid Python variable names, which cannot start with numbers.

In the next line in the init block, a Character variable is defined:

```
defined e = Character('Eileen')
```

In this case, the statement is creating a variable, `e`, and assigning to it the result of a call to the Character function with the argument `'Eileen'`. In simpler terms, it creates a shortcut for use when writing lines of dialogue in the script, so that the full name Eileen doesn't have to be typed out every time she says something.

The next line introduces the `label` statement:

```
label start:
```

A label provides a spot for other Ren'Py script statements to call or jump to. By default, the Ren'Py main menu jumps to the label `start` when you start a new game.

Following the `start` label is the part of the script where you write the actual interactive game, so to speak. First, a `scene` statement:

```
    scene bg whitehouse
```

This scene statement simply adds the previously defined Displayable `bg whitehouse` to the screen.

Next, a `show` statement:

```
    show eileen happy
```

This show statement adds the previously defined `eileen happy` to the screen, on top of the image shown by the preceding scene statement.

```
    e "I'm standing in front of the White House."
```

This line is an example of a `say` statement, although you will notice that it doesn't start with an actual 'say' keyword. Since `say` statements make up the bulk of a game script, they use the most basic syntax possible.

Here, the statement starts with the previously defined Python variable `e`, which is used to indicate to the Ren'Py interpreter that the line of dialogue should be shown on screen with the name defined in the call to its Character function - 'Eileen', of course.

Next comes another `show` statement:

```
    show eileen upset
```

This line demonstrates one of the subtle advantages to using a show statement in combination with a multi-word Displayable label. Recall that in the init block, there were `eileen happy` and `eileen upset` definitions. When the show statement executes, it first checks whether there are already any images on the screen where the first word of the name matches the first word of the new image to show. If it finds any, the previous image is hidden and replaced with the new one.

Put simply, the `eileen happy` image already on the screen is hidden and replaced with `eileen upset`, since they both start with `eileen`.

Next is another line of Eileen dialogue:

```
    e "I once wanted to go on a tour of the West Wing, but you have to
       know somebody to get in."
```

This line happens to be split onto two physical script lines, but will be joined when shown on screen. Note that script line breaks will not be carried into the displayed line - if you want to force a linebreak on screen, you will need to use an explicit newline (`\n`).

Next, another say statement, in its simplest form:

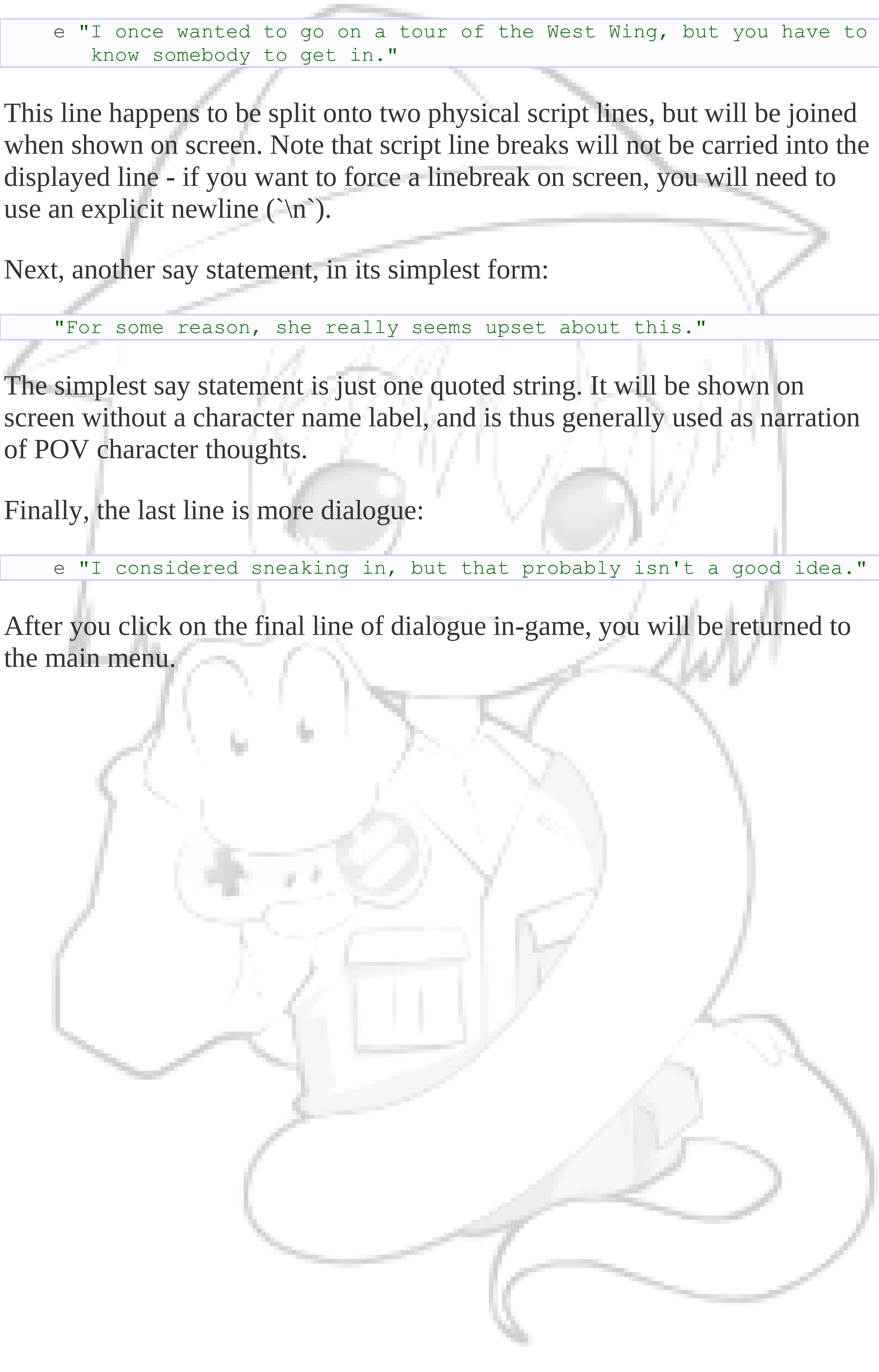
```
    "For some reason, she really seems upset about this."
```

The simplest say statement is just one quoted string. It will be shown on screen without a character name label, and is thus generally used as narration of POV character thoughts.

Finally, the last line is more dialogue:

```
    e "I considered sneaking in, but that probably isn't a good idea."
```

After you click on the final line of dialogue in-game, you will be returned to the main menu.



Script, Line, and Block Structure

Ren'Py scripts consist of one or more .rpy files. These script files may be read in any order, and all of them together make up a Ren'Py script. Please see Files and Directories for information about where Ren'Py searches for .rpy files.

Each of these files is divided into a series of **logical lines**. The first logical line of a file begins at the start of a file, and another logical line begins after each logical line ends, until the end of the file is reached. By default, a logical line is terminated by the first newline encountered. However, a line will not terminate if any of the following are true:

- The newline is immediately preceded by a backslash. In this case, the backslash is removed, and the newline is ignored.
- An opening parenthesis, bracket, or brace has been encountered without encountering the corresponding closing character.
- The newline occurs within a string.

Ren'Py script files also can include **comments**. A comment begins with a hash mark that is not contained within a string, and continues to, but does not include, the next newline character. Some examples are:

```
# This line contains only a comment.
scene bg whitehouse  # This line contains a statement as well.
```

If, after eliminating comments, a logical line is empty, that logical line is ignored.

Logical lines are then combined into **blocks**. Two logical lines are in the same block if the lines have the same indentation preceding them, and no logical line with a lesser amount of indentation occurs between the two lines. Indentation may only consist of spaces, not tabs. In the following example:

```
line 1
    line a
    line b
line 2
    line c
    line d
```

In this example, here are three blocks. One block contains lines 1 and 2, another lines a and b, and the third contains lines c and d. This example can also serve to illustrate the concept of a *block associated with a line*. A block is associated with a line if the block starts on the next logical line following the line. For example, the block containing lines a and b is associated with line #

There are three kinds of blocks in an Ren'Py program. The most common is a block containing Ren'Py statements. Blocks may also contain menuitems or python code. The top-level block (the one that contains the first line of a file) is always a block of Ren'Py statements.

Syntax Constructs

Before we can describe Ren'Py statements, we must first describe a number of syntactic constructs those statements are built out of. In this subsection, we describe such constructs.

Keywords are words that must literally appear in the source code. They're used ito introduce statements, or to delimit parts of statements. You'll see keywords throughout the descriptions of statements. In grammar rules, keywords are in quotes. The keywords used by Ren'Py are:

```
at
behind
call
elif
else
expression
hide
if
image
init
jump
label
menu
onlayer
pass
python
return
scene
set
show
transform
while
with
zorder
```

A **name** consists of a letter or underscore (_) followed by zero or more letters, numbers, or underscores. For this purpose, unicode characters between U+00a0 and U+fffd are considered to be letters. A name may not be a keyword.

A **string** begins with a quote character (one of ", ', or `), contains some sequence of characters, and ends with the same quote character. Inside a Ren'Py string, whitespace is collapsed into a single space, unless preceded by a backslash (as \ ). Backslash is used to escape quotes, special characters such as % (written as \%) and { (written as \{). It's also used to include newlines, using the \n sequence.

A **simple_expression** is a Python expression that starts with a name, a string, or any Python expression in parentheses. This may be followed by any number of the following:

- A dot followed by a name.
- A parenthesized python expression.

A **python_expression** is an arbitrary python expression that may not include a colon. These expressions are generally used to express the conditions in the if and while statements.

An **image_name** consists of one or more names, separated by spaces. The first name in an image_name is known as the **image tag**.

Name Munging

Before parsing a file, Ren'Py looks for names of the form `__word`, where word does not contain __. That is, starting with two or three underscores, and not containing another two consecutive underscores. It munges these names in a filename-specific manner, generally ensuring that they will not conflict with the same name in other files. For example, in the file script.rpy, `__v` is munged into `_m1_script__v` (the name is prefixed with the munged filename).

Names beginning with an odd number of underscores are reserved for Ren'Py, while those beginning with an even number of underscores can safely be used in scripts.

Grammar Rules

We will be giving grammar rules for some of the statements. In these rules, a word in quotes means that that word is literally expected in the script. Parenthesis are used to group things together, but they don't correspond to anything in the script. Star, question mark, and plus are used to indicate that the token or group they are to the right of can occur zero or more, zero or one, or one or more times, respectively.

If we give a name for the rule, it will be separated from the body of the rule with a crude ascii-art arrow (->).

Statements

Call Statement

The call statement is used to transfer control to the statement with the given name. It also pushes the name of the statement following this one onto the call stack, allowing the return statement to return control to the statement following this one.

```
call_statement -> "call" name ( "from" name )?
call_statement -> "call" "expression" simple_expression ( "from" name
)?

call_statement -> "call" name "(" arguments ")" ( "from" name )?
call_statement -> "call" "expression" simple_expression "pass" "("
arguments ")" ( "from" name )?
```

If the expression keyword is present, the expression is evaluated, and the string so computed is used as the name of the statement to call. If the expression keyword is not present, the name of the statement to call must be explicitly given.

If the optional from clause is present, it has the effect of including a label statement with the given name as the statement immediately following the call statement. An explicit label is required here to ensure that saved games with return stacks can return to the proper place when loaded on a changed script. From clauses should be included for all calls in released games.

As from clauses may be distracting when a game is still under development, we provide with Ren'Py a program, called add_from, that adds from clauses to all bare calls in any game directory. It can be found in tools/add_from, although it needs to be run from the base directory. The easiest way to do this on windows is by running tools/game_add_from.bat. It should be run before a final release of your game is made. Be sure to make a backup of your game directories before running add_from. Also note that add_from produces .bak files for all files it can change, so delete them when you're satisfied that everything worked.

```
e "First, we will call a subroutine."

call subroutine from _call_site_1

# ...

label subroutine:

    e "Next, we will return from the subroutine."

    return
```

The call statement may take arguments, which are processed as described in PEP 3102. If the return statement returns a value, that value is stored in the _return variable, which is dynamically scoped to each context.

When using a call expression with an arguments list, the pass keyword must be inserted between the expression and the arguments list. Otherwise, the arguments list will be parsed as part of the expression, not as part of the call.

Define Statement

The define statement causes its expression to be evaluated, and assigned to the supplied name. If not inside an init block, the define statement will automatically be run with init priority 0.

```
define_statement -> "define" name "=" python_expression
define e = Character("Eileen")
```

Hide Statement

The `hide` statement is used to hide an image from the screen.

hide_statement -> "hide" image_name ("onlayer" name)?

A `hide` statement operates on the layer supplied in the `onlayer` clause of the *image_spec*, defaulting to "master" if no such clause has been supplied. It finds an image beginning with the image tag of the image name, and removes it from that layer.

Please note that the `hide` statement is rarely used in practice. Show can be used by itself when a character is changing emotion, while `scene` is used to remove all images at the end of a scene. Hide is only necessary when a character leaves before the end of a scene.

If Statement

The `if` statement is used to conditionally execute a block of statements.

```
if_statement -> "if" python_expression ":"
elif_clause -> "elif" python_expression ":"
else_clause -> "else" ":"
```

The `if` statement is the only statement which consists of more than one logical line in the same block. The initial `if` statement may be followed by zero or more `elif` clauses, concluded with an optional `else` clause. The expression is evaluated for each clause in turn, and if it evaluates to a true value, then the block associated with that clause is executed. If no expression evaluates to true, then the block associated with the `else` clause is executed. (If an `else` clause exists, execution immediately continues with the next statement.) In any case, at the end of the block, control is transferred to the statement following the if statement.

```
if points >= 10:

   e "Congratulations! You're getting the best ending!"

elif points >= 5:

   e "It's the good ending for you."

else:

   e "Sorry, you're about to get the bad ending."
```

Image Statement

The `image` statement is used to declare images to Ren'Py. Image statements can either appear in init blocks, or are implicitly placed in an init block with priority 990. *(changed in 6.9.0)*

```
image_statement -> "image" image_name "=" python_expression
```

An `image` statement binds an image name to a displayable. The displayable is computed by the supplied python expression, with the result of the expression being passed to the im.Image function in loose mode. This means that if the assignment is a single string, it is interpreted as an image filename. Displayables are passed through unmolested. Once an image has been defined using an `image` statement, it can be used by the `scene`, `show`, and `hide` statements.

For a complete list of functions that define displayables, see the displayables page.

```
image eileen happy = "eileen/happy.png"
image eileen upset = "eileen/upset.png"
```

Init Statement

The `init` statement is used to execute blocks of Ren'Py statements before the script executes. Init blocks are used to define images and characters, to set up unchanging game data structures, and to customize Ren'Py. Code inside init blocks should not interact with the user or change any of the layers, and so should not contain `say`, `menu`, `scene`, `show`, or `hide` statements, as well as calls to any function that can do these things.

```
init_statement -> "init" (number)? ":"
```

An `init` statement is introduced with the keyword `init`, followed by an optional priority number, and a mandatory colon. If the priority is not given, it defaults to 0. Priority numbers should be in the range -999 to 999. Numbers outside of this range are reserved for Ren'Py code.

The priority number is used to determine when the code inside the init block executes. Init blocks are executed in priority order from low to high. Within a file, init blocks with the same priority are run in order from the top of the file to the bottom. The order of evaluation of priority blocks with the same priority between files is undefined.

The init blocks are all run once, during a special init phase. When control reaches the end of an init block during normal execution, execution of that block ends. If an init statement is encountered during normal execution, the init block is not run. Instead, control passes to the next statement.

Jump Statement

The `jump` statement is used to transfer control to the statement with the given name.

```
jump_statement -> "jump" name
jump_statement -> "jump" "expression" simple_expression
```

If the `expression` keyword is present, the expression is evaluated, and the string so computed is used as the name of the statement to jump to. If the `expression` keyword is not present, the name of the statement to jump to must be explicitly given.

Unlike `call`, `jump` does not push the target onto any stack. As a result, there's no way to return to where you've jumped from.

```
label loop_start:

e "Oh no! It looks like we're trapped in an infinite loop."

jump loop_start
```

Label Statement

Label statements allow a name to be assigned to a program point. They exist solely to be called or jumped to, whether by script code or the Ren'Py config.

```
label_statement -> "label" name ":"
label_statement -> "label" name "(" parameters ")" ":"
```

A `label` statement may have a block associated with it. In that case, control enters the block whenever the label statement is reached, and proceeds with the statement after the label statement whenever the end of the block is reached.

The label statement may take an optional list of parameters. These parameters are processed as described in PEP 3102, with two exceptions:

- The values of default parameters are evaluated at call time.
- The variables are dynamically, rather than lexically, scoped.

Inside a called label, variables can be declared dynamic using the renpy.dynamic function:

Function: **renpy.dynamic** (*vars):

This declares a variable as dynamically scoped to the current Ren'Py call. The first time renpy.dynamic is called in the current call, the values of the variables named in the supplied strings are stored. When we return from the current call, the variables are given the values they had at the time when renpy.dynamic was called. If the variables are undefined when renpy.dynamic is called, they are undefined after the current call returns. If renpy.dynamic is called twice for the same variable in a given call, it has no effect the second time.

Menu Statement

Menus are used to present the user with a list of choices that can be made. In a visual novel, menus are the primary means by which the user can influence the story.

```
menu_statement -> "menu" ( name )? ":"
```

A `menu` statement is introduced by the keyword `menu`, an optional name, and a colon. If the name is supplied, it is treated as a label for this `menu` statement, as if the menu statement was preceded by a label statement.

A `menu` statement must have a block associated with it. This is a menuitem block that must contain one or more menuitems in it. There are several kinds of menuitems that can be contained in a menuitem block.

```
caption_menuitem -> string
```

The first kind of menuitem is a string. This string is placed into a menu as a caption that cannot be selected. In general, captions are used to indicate what the menu is for, especially when it is not clear from the choices.

```
choice_menuitem -> string ( "if" python_expression )? ":"
```

The second kind of menuitem gives a choice the user can make. Each choice must have a block of Ren'Py statements associated with it. If the choice is selected by the user, then block of statements associated with the choice is executed. A choice may also have an optional `if` clause that includes a Python expression. This clause gives a condition that must be satisfied for the choice to be presented to the user. A terminating colon is used to indicate that this menuitem is a choice.

```
set_menuitem -> "set" variable_name
```

The third kind of menuitem provides a variable in which to store the list of choices the user has made, and prevent the user making the same choice if the menu is visited multiple times. This variable must be defined before the menu statement, and should be an empty list, [ ]. When the user chooses a choice from the menu, that choice will be stored in the list. When the game reaches another menu statement using the same variable name in a `set` clause (or reaches the same menu again), any choices matching items in the list will not be shown.

```
with_menuitem -> "with" simple_expression
```

The final kind of menuitem is a `with` clause. Please see Transitions for more information on `with` clauses.

```
menu what_to_do:

    "What should we do today?"

    "Go to the movies.":
        "We went to the movies."

    "Go shopping.":
        "We went shopping, and the girls bought swimsuits."
        $ have_swimsuits = True

    "Go to the beach." if have_swimsuits:
        "We went to the beach together. I got to see the girls in
their
         new swimsuits."
```

Details

When a menu is to be shown to the user, the first thing that happens is that a list of captions and choices is built up from the menuitems associated with the menu. Each of the choices that has an expression associated with it has that expression evaluated, and if not true, that choice is removed from the list. If no choices survive this process, the menu is not displayed and execution continues with the next statement. Otherwise, the `menu` function is called with the list of choices, displays the menu to the user, and returns a chosen choice. Execution continues with the block corresponding to the chosen choice. If execution reaches the end of that block, it continues with the the statement after the menu.

Pause Statement

The pause statement causes Ren'Py to pause until the mouse is clicked. If the optional expression is given, it will be evaluated to a number, and the pause will automatically terminate once that number of seconds has elapsed.

```
pause_statement -> "pause" ( simple_expression )?
```

Play Statement

The play statement is used to play sound and music. If a file is currently playing, it is interrupted and replaced with the new file.

```
play_statement -> "play" name simple_expression
                       ( "fadeout" simple_expression )?
                       ( "fadein" simple_expression )?
                       ( "loop" | "noloop" )?
                       ( "if_changed" )?
```

The first simple_expression in the play statement is expected to evaluate to either a string containing a filename, or a list of filenames to be played. The name is expected to be the name of a channel. (Usually, this is either "sound", "music", or "movie".) The file or list of files is played using renpy.music.play. The other clauses are all optional. Fadeout gives the fadeout time for currently playing music, in seconds, while fadein gives the time it takes to fade in the new music. Loop causes the music too loop, while noloop forces it not to loop. If_changed causes the music to only change if it is not the currently playing music.

```
play music "mozart.ogg"
play sound "woof.ogg"

"Let's try something more complicated."

play music [ "a.ogg", "b.ogg" ] fadeout 1.0 fadein 1.0
```

Pass Statement

The `pass` statement does not perform an action. It exists because blocks of Ren'Py statements require at least one statement in them, and it's not always sensible to perform an action in those blocks.

```
pass_statement -> "pass"
menu:
    "Should I go to the movies?"
    "Yes":
        call go_see_movie

    "No":
        pass

"Now it's getting close to dinner time, and I'm starving."
```

Python Statement

The `python` statement allows one to execute Python code in a Ren'Py script. This allows one to use Python code to declare things to Ren'Py, to invoke much of Ren'Py's functionality, and to store data in variables that can be accessed by user code. There are two forms of the `python` statement:

```
python_statement -> "$" python_code
python_statement -> "python" ( "hide" )? ":"
```

The first form of a python consists of a dollar sign (`$`) followed by Python code extending to the end of the line. This form is used to execute a single Python statement.

A second form consists of the keyword `python`, optionally the keyword `hide`, and a colon. This is used to execute a block of Python code, supplied after the statement. Normally, Python code executes in a script-global namespace, but if the `hide` keyword is given, a new namespace is created for this block. (The script-global namespace can be accessed from the block, but not assigned to.)

```
$ score += 1

python:
    ui.text("This is text on the screen.")
    ui.saybehavior()
    ui.interact()
```

Init Python Statement. For convenience, we have created the init pythons statement. This statement combines an init statement and a python statement into a single statement, to reduce the indentation required for python-heavy files.

```
init python_statement -> "init" ( number )? "python" ( "hide" )? ":"
```

Queue Statement

The queue statement is used to queue up audio files. They will be played when the channel finishes playing the currently playing file.

```
queue_statement -> "queue" name simple_expression ( "loop" | "noloop"
)?
```

The name is expected to be the name of a channel, while the simple_expression is expected to evaluate to either a string containing a filename, or a list of filenames to be queued up. The files are queued using renpy.music.queue.

Loop causes the queued music to loop, while noloop causes it to play only once.

```
queue sound "woof.ogg"
queue music [ "a.ogg", "b.ogg" ]
```

Return Statement

The `return` statement pops the top location off of the call stack, and transfers control to it. If the call stack is empty, the return statement performs a full restart of Ren'Py.

```
return_statement -> "return"
return_statement -> "return" expression
```

If the optional expression is given to return, it is evaluated, and it's result is stored in the _return variable. This variable is dynamically scoped to each context.

Say Statement

The say statement is used to present text to the user, in the form of dialogue or thoughts. Since the bulk of the of the content of a script will be dialogue or thoughts, it's important that the say statement be as convenient as possible. Because of this, the say statement is the only statement that is not delimited with a keyword or other form of delimiter. Instead, it consists of a string, with an optional *simple_expression* before it to designate who is doing the speaking, and an optional `with` clause after it used to specify a transition.

```
say_statement -> ( simple_expression )? string ( "with"
simple_expression )?
```

There are two forms of the say statement, depending on if the simple expression is provided. The single-argument form consists of a single string (with or without the optional with clause). This form causes the string to be displayed to the user as without any label as to who is saying it. Conventionally, this is used to indicate POV character thoughts or narration.

```
"I moved to my left, and she moved to her right."

"So we were still blocking each other's path."

"I then moved to my right, and at the same time she moved to her
 left."
```

```
"We could be at this all day."
```

The two-argument form of the say statement consist of a *simple_expression*, a string, and optionally a `with` clause. This form of the statement is used to indicate dialogue. The first argument is expected to be an object (usually a Character or DynamicCharacter object) that knows how to show dialogue to the user. The string is then passed to that object, which is responsible for showing it to to the user.

The *simple_expression* can also be a string, rather than an object. Strings are used directly as the name of the character.

```
"Girl" "Hi, my name is Eileen."

e "Starting today, I'll be living here."
```

Details

The two-argument say statement first evaluates the supplied simple expression. It then attempts to call that value (the who value) with the string giving the line of dialogue (the what string). If it can do so, it's finished, as the object that is called is responsible for interacting with the user.

If it can't call the value of the expression, then it copies the `name_only` character object, supplying the given string as a new character name, and then uses that to say the dialogue. (This is done by the `say` and `predict_say` functions found in the store. Changing these functions can change this behavior.)

The single-argument form of the statement simply calls the special function (or object) `narrator` with the string to be shown. This function is responsible for showing the string to the user. Character and DynamicCharacter objects are suitable for use as the `narrator`.

The `with` clause is used to specify a transition; see With Statement and Clauses for details.

Scene Statement

The `scene` statement clears a layer by removing all images from it. It may then show a supplied image to the user. This makes it appropriate for changing the background of a scene.

```
scene_statement -> "scene" ("onlayer" name)?  (...)?
```

The scene statement first clears out all images from a layer, defaulting to the "master" layer if no other layer is specified. If additional arguments are present, then they are handled as if a show statement statement was supplied.

By default, no background is added to the screen, so we recommend that every script begin with a scene statement that shows a full-screen background to the user.

Show Statement

The show statement is used to add an image to a layer. The image must have been defined using the image statement statement.

```
show_statement -> "show" image_name
                  ( "at" transform_list )?
                  ( "as" name )?
                  ( "behind" name_list )?
                  ( "onlayer" name )?
                  ( "with" simple_expression )?
```

When adding an image, the show statement first checks to see if an image with the same tag (by default, first part of the image name) exists in the layer. If so, that image is replaced, without changing the order. This means that it's rarely necessary to hide images.

The show statement takes several optional clauses.

- The at clause takes a comma-separated list of positions and motions. These are created using the position and motion functions.
- The as clause specifies the image tag directly. This allows the same image to be shown on the screen twice.
- The behind takes gives a comma-separated list of image tags. The image will be shown behind all images with this tag.
- The onlayer clause specifies the layer the image will be shown on.
- The with clause specifies a transition that occurs when this image is shown. See the with statement statement for more details.

When an image is shown, Ren'Py checks to see if there was a previous image with that tag, and if that image used a transform. If this is true, Ren'Py does two things:

1. If the new image is not a transform, it wraps it in a transform.
2. The transform is initialized to have the properties of the old transform.

The generally has the effect of "remembering" the position of images shown on the screen. In some cases, this memory effect may override a position encoded into an image. In that case, the image must be hidden and shown again.

```
scene living_room
show eileen happy at left
show golden glow as halo at left behind eileen

e "I'm feeling happy right now."

show eileen upset at left
show darkness as halo at left behind eileen

e "But sometimes, I can get upset for no good reason."
```

The show statement can also be used to display text, using the ParameterizedText displayable. The `text` displayable is defined by default, and uses the centered_text style:

```
show text "Centered text"
```

There is a second form of the show statement that takes an expression that returns a displayable. This can be used to show a displayable directly. The tag of the displayable is undefined, unless given with the as clause.

```
show_statement -> "show" "expression" simple_expression
                  ( "as" name )?
                  ( "onlayer" name )?
                  ( "at" transform_list )?
                  ( "behind" name_list )?
                  ( "with" simple_expression )?
show expression "myimage.png"
```

Stop Statement

The stop statement is used to stop playing sound and music.

```
stop_statement -> "stop" name
                       ( "fadeout" simple_expression )?
```

The name is expected to be the name of an audio channel. The sound or music is stopped using renpy.music.stop. The optional fadeout clause expects a time in seconds, and will cause it to take that long to fadeout the music.

```
stop sound
stop music fadeout 1.0
```

Window Statement

The window statement is used to control if a window is shown when a character is not speaking. (For example, during transitions and pauses.)

```
window_statement -> window ( "show" | "hide" ) ( simple_expression )?
```

The "window show" statement causes the window to be shown, while the "window hide" statement hides the window. If the optional simple_expression is given, it's a transition that's used to show and hide the window. If not given, it defaults to config.window_show_transition and config.window_hide_transition. Giving None as the transition prevents it from occuring.

The window itself is displayed by calling config.empty_window. It defaults to having the narrator say an empty string.

```
show bg washington
show eileen happy
with dissolve

window show dissolve

"I can say stuff..."

show eileen happy at right
with move

"... and move, while keeping the window shown."

window hide dissolve
```

With Statement and Clauses

The with statement and with clauses are used to show transitions to the user. These transitions are always from the last screen shown to the user to the current screen. At the end of a with statement, the last screen shown to the user is set to the current screen. The last screen shown to the user can also be updated by say or menu statements, as well as by Python code.

The with statement has the form:

```
with_statement -> "with" simple_expression
```

The *simple_expression* is expected to evaluate to a transition function. If it evaluates to the value *None*, the last screen shown to the user is updated to the current screen, without performing a transition. This is useful to remove transient interface items (like a prior say statement) from partaking in a transition.

For convenience, a number of statements support `with` clauses. In the case of the `scene`, `show`, and `hide` statements, the `with` clause is equivalent to placing a "with None" statement before the `scene`, `show` or `hide` statement, and a "with transition" statement after it. For example, the statement:

```
show eileen happy with dissolve
```

is equivalent to:

```
with None
show eileen happy
with dissolve
```

This behavior can lead to undesired side-effects. The code:

```
show bg whitehouse with dissolve
show eileen happy with dissolve
```

will cause two transitions to occur. To ensure only a single transition occurs, one must write:

```
with None
show bg whitehouse
show eileen happy
with dissolve
```

With clauses can also be applied to `say` and `menu` statements. In this case, the transition occurs when the dialogue or menu is first shown to the user.

For pre-defined transition functions that can be used in any script, see Pre-defined Transitions. For functions that return transition functions, see Transition Constructors.

While Statement

The `while` statement is used to execute a block of Ren'Py statement while a condition remains true.

```
while_statement -> "while" python_expression ":"
```

When a `while` statement is executed, the *python_expression* is evaluated. If it evaluates to true, control is transferred to the first statement in the block associated with this `while` statement. If false, control is instead sent to the statement following the `while` statement.

When control reaches the end of the block associated with the while statement, it returns to the `while` statement. This allows the `while` statement to check the condition again, and evaluate the block if the condition remains true.

```
while not endgame:

   "It's now morning. Time to get up and seize the day."

    call morning
    call afternoon
    call evening

    "Well, time to call it a night."

"Now it's time to wake up and face the endgame."
```

The Animation and Transformation Language (ATL) provides a high-level way of choosing a displayable to show, positioning it on the screen, and applying transformations such as rotation, zoom, and alpha-modification. These can be changed over time, and in response to events.

Ren'Py Script Statements

Blocks of ATL code can be introduced in three ways.

Transform Statement. The first way is using the transform statement. The transform statement creats a position and motion function that can be supplied to the at clause of a show or scene statement. The syntax of the transform statement is:

```
transform_statement -> "transform" name "(" parameters ")" ":"
    atl_block
```

The transform statement should be run at init time. If it is found outside an init block, then it is automatically placed inside an init block with a priority of 0. The transform may have a list of parameters, which must be supplied when it is called. The syntax is the same as for labels, except that we do not allow a variable number of arguments.

Name must be given, and must be a python identifier. The transform created from the ATL block is bound to this name.

Image Statement with ATL block. The second way to use ATL is as part of an image statement with ATL block. This binds an image name to the given transform. As there's no way to supply parameters to this transform, it's only useful if the transform defines an animation. The syntax for an image statement with ATL block is:

```
image_statement -> "image" image_name ":"
    atl_block
```

Scene and Show statements with ATL block. The final way to use ATL is as part of a scene or show statement. This wraps the image being shown inside an ATL transformation.

```
atl_scene_statement -> scene_statement ":"
    atl_block

atl_show_statement -> show_statement ":"
    atl_block
```

ATL Syntax and Semantics

An ATL block consists of one or more logical lines, all at the same indentation, and indented relative to the statement containing the block. Each logical line in an ATL block must contain one or more ATL statements.

There are two kinds of ATL statements: simple and complex. Simple statements do not take an ATL block. A single logical line may contain one or more ATL statements, separated by commas. A complex statement contains a block, must be on its own line. The first line of a complex statement always ends with a colon (":").

By default, statements in a block are executed in the order in which they appear, starting with the first statement in the block. Execution terminates when the end of the block is reached. Time statements change this, as described in the appropriate section below.

Execution of a block terminates when all statements in the block have terminated.

If an ATL statement requires evaluation of an expression, such evaluation occurs when the transform is first added to the scene list. (Such as when using a show statement or ui function.)

ATL Statements

The following are the ATL statements.

Interpolation Statement

```
interpolation_statement ->
    ( warper simple_expression | "warp" simple_expression
simple_expression )?
    ( property simple_expression ( "knot" simple_expression )*
    | "clockwise"
    | "counterclockwise"
    | "circles" simple_expression
    | simple_expression )*
```

The interpolation statement is the main way that ATL controls transformations.

The first part of a the interpolation statement is used to select a a function that time-warps the interpolation. (That is, a function from linear time to non-linear time.) This can either be done by giving the name of a warper registered with ATL, or by giving the keyword "warp" followed by an expression giving a function. Either case is followed by a number, giving the number of seconds the the interpolation should take.

If no warp function is given, the interpolation is run for 0 seconds, using the pause function.

The warper and duration are used to compute a completion fraction. This is done by dividing the time taken by the interpolation by the duration of the interpolation. This is clamped to the duration, and then passed to the warper. The result returned by the warper is the completion fraction.

The interpolation statement can then contain a number of of other clauses. When a property and value are present, then the value is the value the property will obtain at the end of the statement. The value can be obtained in several ways:

- If the value is followed by one or two knots, then spline motion is used. The starting point is the value of the property at the start of the interpolation, the end point is the property value, and the knots are used to control the spline.
- If the interpolation statement contains a "clockwise" or "counterclockwise" clause, circular motion is used, as described below.
- Otherwise, the value is linearly interpolated between the start and end locations, using the completion fraction.

The "clockwise", "counterclockwise", and "circle" clauses are used in circular motion, described below.

If a simple expression is present, it should evaluate to a transform with only a single interpolation statement, without a warper, splines, or circular motion. The properties from the transform are processes as if they were included in this statement.

Some sample interpolations are:

```
    show logo base:
         # Show the logo at the upper right side of the screen.
         xalign 0.0 yalign 1.0

         # Take 1.0 seconds to move things back to the left.
         linear 1.0 xalign 0.0

         # Take 1.0 seconds to move things to the location specified
in the
         # truecenter transform. Use the ease warper to do this.
         ease 1.0 truecenter

         # Just pause for a second.
         pause 1.0

         # Set the location to circle around.
         alignaround (.5, .5)
```

```
        # Use circular motion to bring us to spiral out to the top of
        # the screen. Take 2 seconds to do so.
        linear 2.0 yalign 0.0 clockwise circles 3

        # Use a spline motion to move us around the screen.
        linear 2.0 align (0.5, 1.0) knot (0.0, .33) knot (1.0, .66)
```

An important special case is that the pause warper, followed by a time and nothing else, causes ATL execution to pause for that amount of time.

Some properties can have values of multiple types. For example, the xpos property can be an int, float, or absolute. The behavior is undefined when an interpolation has old and new property values of different types.

Time Statement

```
time_statement -> "time" simple_expression
```

The time statement is a simple control statement. It contains a single simple_expression, which is evaluated to give a time, expressed as seconds from the start of execution of the containing block.

When the time given in the statement is reached, the following statement begins to execute.This transfer of control occurs even if a previous statement is still executing, and causes any prior statement to immediately terminate.

Time statements are implicitly preceded by a pause statement with an infinite time. This means that if control would otherwise reach the time statement, it waits until the time statement would take control.

When there are multiple time statements in a block, they must strictly increase in order.

```
image backgrounds:
    "bg band"
    time 2.0
    "bg whitehouse"
    time 4.0
    "bg washington"
```

Expression Statement

```
expression_statement -> simple_expression ("with" simple_expression)?
```

An expression statement is a simple statement that starts with a simple expression. It then contains an optional with clause, with a second simple expression.

The first simple expression may evaluate to a transform (defined with the transform statement) or a displayable. If it's a transform, that transform is executed. With clauses are ignored when a transform is supplied.

The first simple expression may evaluate to an integer or floating point number. In that case, it's taken as a number of seconds to pause execution for.

Otherwise, the expression is interpreted to be a displayable. This displayable replaces the child of the transform when this clause executes, making it useful for animation. If a with clause is present, the second expression is evaluated as a transition, and the transition is applied to the old and new displayables.

```
image atl example:
     # Display logo_base.png
     "logo_base.png"

     # Pause for 1.0 seconds.
     1.0

     # Show logo_bw.png, with a dissolve.
     "logo_bw.png" with Dissolve(0.5, alpha=True)

     # Run the move_right tranform.
     move_right
```

Pass Statement

```
pass_statement -> "pass"
```

The pass statement is a simple statement that causes nothing to happen. This can be used when there's a desire to separate statements, like when there are two sets of choice statements that would otherwise be back-to-back.

Repeat Statement

```
repeat_statement -> "repeat" (simple_expression)?
```

The repeat statement is a simple statement that causes the block containing it to resume execution from the beginning. If the expression is present, then it is evaluated to give an integer number of times the block will execute. (So a block ending with "repeat 2" will execute at most twice.)

The repeat statement must be the last statement in a block.

```
show logo base:
    xalign 0.0
    linear 1.0 xalign 1.0
    linear 1.0 xalign 0.0
    repeat
```

Block Statement

```
block_statement -> "block" ":"
    atl_block
```

The block statement is a complex statement that contains a block of ATL code. This can be used to group statements that will repeat.

```
label logo base:
    alpha 0.0 xalign 0.0 yalign 0.0
    linear 1.0 alpha 1.0

    block:
        linear 1.0 xalign 1.0
        linear 1.0 xalign 0.0
        repeat
```

Choice Statement

```
choice_statement -> "choice" (simple_expression)? ":"
    atl_block
```

The choice statement is a complex statement that defines one of a set of potential choices. Ren'Py will pick one of the choices in the set, and execute the ATL block associated with it, and then continue execution after the last choice in the choice set.

Choice statements are greedily grouped into a choice set when more than one choice statement appears consecutively in a block. If the *simple_expression* is supplied, it is a floating-point weight given to that block, otherwise 1.0 is assumed.

```
image eileen random:
    choice:
        "eileen happy"
    choice:
        "eileen vhappy"
    choice:
        "eileen concerned"

    pause 1.0
    repeat
```

Parallel Statement

The parallel statement is used to define a set of ATL blocks to execute in parallel.

```
parallel_statement -> "parallel" ":"
    atl_block
```

Parallel statements are greedily grouped into a parallel set when more than one parallel statement appears consecutively in a block. The blocks of all parallel statements are then executed simultaneously. The parallel statement terminates when the last block terminates.

The blocks within a set should be independent of each other, and manipulate different properties. When two blocks change the same property, the result is undefined.

```
show logo base:
    parallel:
        xalign 0.0
        linear 1.3 xalign 1.0
        linear 1.3 xalign 0.0
        repeat
    parallel:
        yalign 0.0
        linear 1.6 yalign 1.0
        linear 1.6 yalign 0.0
        repeat
```

Event Statement

```
event_prod_statement -> "event" name
```

The event statement is a simple statement that causes an event with the given name to be produced.

When an event is produced inside a block, the block is checked to see if an event handler for the given name exists. If it does, control is transferred to the event handler. Otherwise, the event propagates to any containing event handler.

On Statement

The On statement is a complex statement that defines an event handler. On statements are greedily grouped into a single statement.

```
on_statement -> "on" name ":"
    atl_block
```

The on statement is used to handle events. When an event is handled, handling of any other event ends and handing of the new event immediately starts. When an event handler ends without another event occuring, the "default" event is produced (unless were already handing the "default" event).

Execution of the on statement will never naturally end. (But it can be ended by the time statement, or an enclosing event handler.)

```
show logo base:
     on show:
        alpha 0.0
        linear .5 alpha 1.0
     on hide:
        linear .5 alpha 0.0
```

Contains Statement

The contains statement sets the displayable contained by this ATL transform. (The child of the transform.) There are two variants of the contains statement.

Contains Expression. The contains expression variant takes an expression, and sets that expression as the child of the transform. This is useful when an ATL transform wishes to contain, rather than include, a second ATL transform.

```
contains_statement -> "contains" expression
transform an_animation:
     "1.png"
     pause 2
     "2.png"
     pause 2
     repeat

image move_an_animation:
     contains an_animation

     # If we didn't use contains, we'd still be looping and would
never reach here.
     xalign 0.0
     linear 1.0 yalign 1.0
```

Contains Block. The contains block allows one to define an ATL block that is used for the child of this ATL transform. One or more contains block statements will be greedily grouped together, wrapped inside a Fixed, and set as the child of this transform.

```
contains_statement -> "contains" ":"
    atl_block
```

Each block should define a displayable to use, or else an error will occur. The contains statement executes instantaneously, without waiting for the children to complete. This statement is mostly syntactic sugar, as it allows arguments to be easily passed to the children.

```
image test double:
     contains:
          "logo.png"
          xalign 0.0
          linear 1.0 xalign 1.0
          repeat

     contains:
          "logo.png"
          xalign 1.0
          linear 1.0 xalign 0.0
          repeat
```

Function Statement

The function statement allows ATL to use Python functions to control the ATL properties.

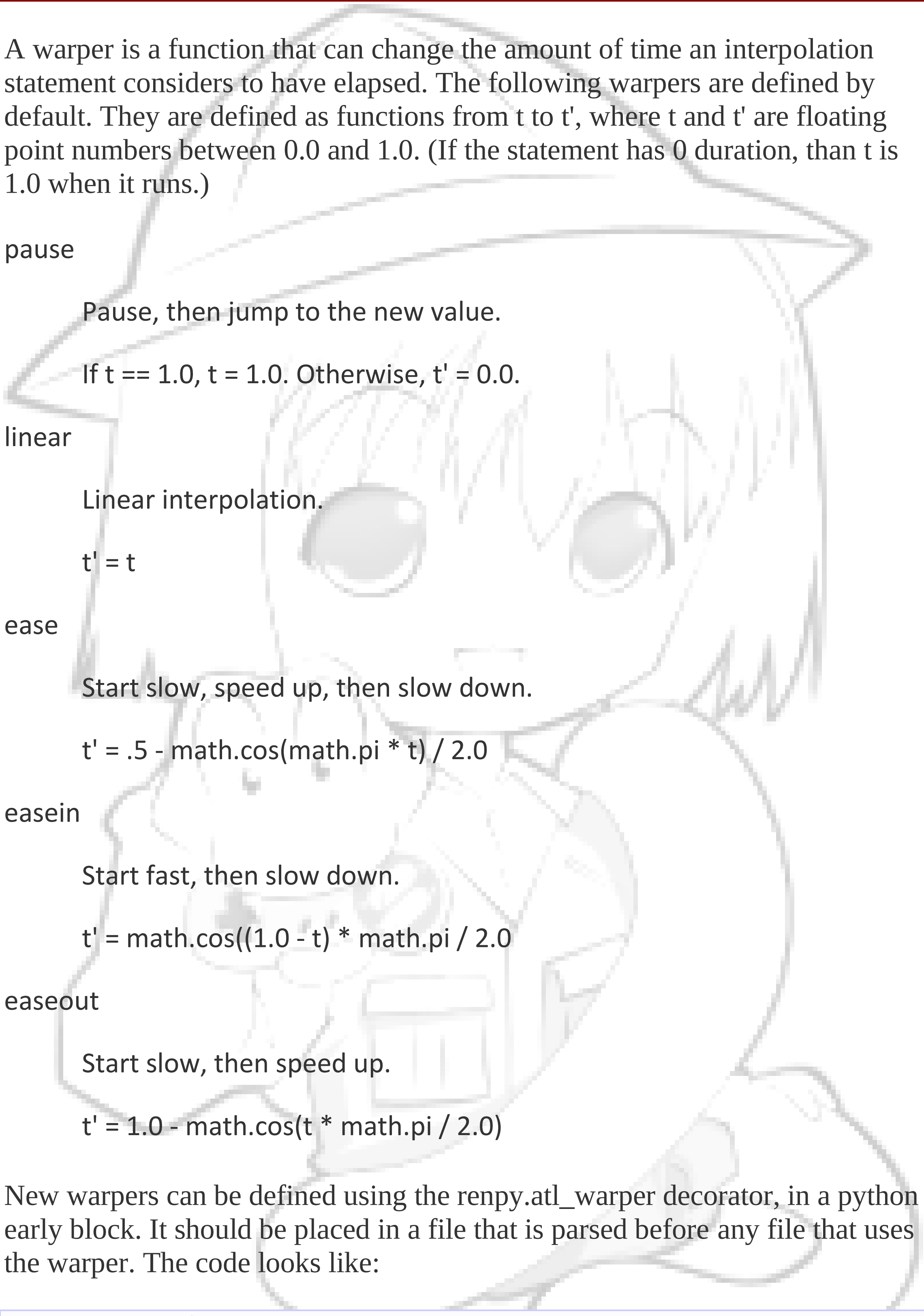
```
function_statement -> "function" expression
```

The functions have the same signature as those used with Transform:

- The first argument is a transform object. Transform properties can be set on this object.
- The second argument is the shown timebase, the number of seconds since the function began executing.
- The third argument is the the animation timebase, which is the number of seconds something with the same tag has been on the screen.
- If the function returns a number, it will be called again after that number of seconds has elapsed. (0 seconds means to call the function as soon as possible.) If the function returns None, control will pass to the next ATL statement.

```
init python:
    def slide_function(trans, st, at):
        if st > 1.0:
            trans.xalign = 1.0
            return None
        else:
            trans.xalign = st
            return 0

label start:
    show logo base:
        function slide_function
        pause 1.0
        repeat
```

Warpers



A warper is a function that can change the amount of time an interpolation statement considers to have elapsed. The following warpers are defined by default. They are defined as functions from t to t', where t and t' are floating point numbers between 0.0 and 1.0. (If the statement has 0 duration, than t is 1.0 when it runs.)

pause

Pause, then jump to the new value.

If t == 1.0, t = 1.0. Otherwise, t' = 0.0.

linear

Linear interpolation.

t' = t

ease

Start slow, speed up, then slow down.

t' = .5 - math.cos(math.pi * t) / 2.0

easein

Start fast, then slow down.

t' = math.cos((1.0 - t) * math.pi / 2.0

easeout

Start slow, then speed up.

t' = 1.0 - math.cos(t * math.pi / 2.0)

New warpers can be defined using the renpy.atl_warper decorator, in a python early block. It should be placed in a file that is parsed before any file that uses the warper. The code looks like:

```
python early hide:

    @renpy.atl_warper
    def linear(t):
        return t
```

Transform Properties

The following transform properties exist.

When the type is given as position, it may be an int, absolute, or float. If it's a float, it's interpreted as a fraction of the size of the containing area (for pos) or displayable (for anchor).

Note that not all properties are independent. For example, xalign and xpos both update some of the same underlying data. In a parallel statement, only one block should adjust horizontal position, and one should adjust vertical positions. (These may be the same block.) The angle and radius properties set both horizontal and vertical positions.

pos

type: (position, position)

default: (0, 0)

The position, relative to the top-left corner of the containing area.

xpos

type: position

default: 0

The horizontal position, relative to the left side of the containing area.

ypos

type: position

default: 0

The vertical position, relative to the top of the containing area.

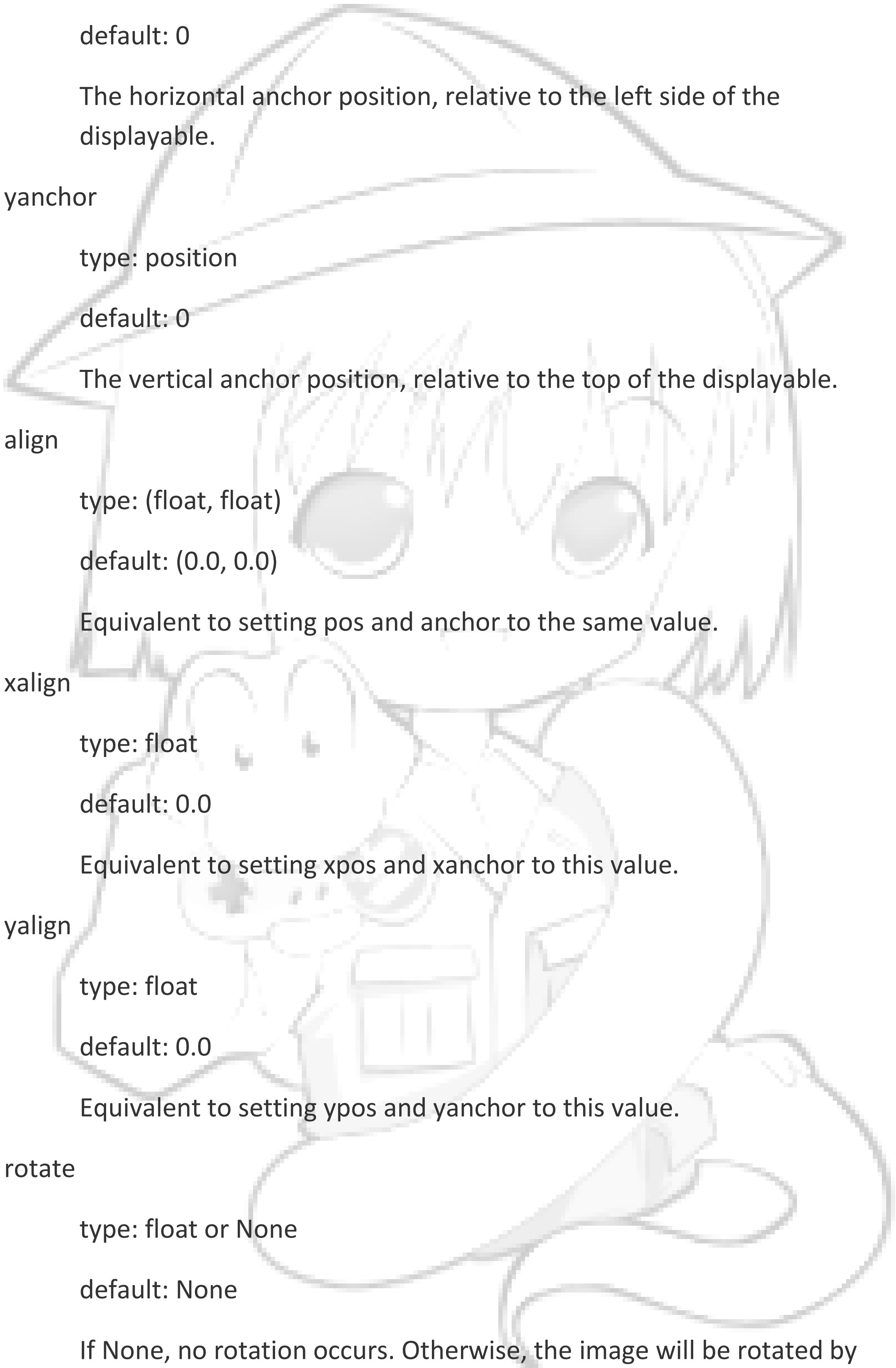
anchor

type: (position, position)

default: (0, 0)

The anchor position, relative to the top-left corner of the displayable.

xanchor



type: position

default: 0

The horizontal anchor position, relative to the left side of the displayable.

yanchor

type: position

default: 0

The vertical anchor position, relative to the top of the displayable.

align

type: (float, float)

default: (0.0, 0.0)

Equivalent to setting pos and anchor to the same value.

xalign

type: float

default: 0.0

Equivalent to setting xpos and xanchor to this value.

yalign

type: float

default: 0.0

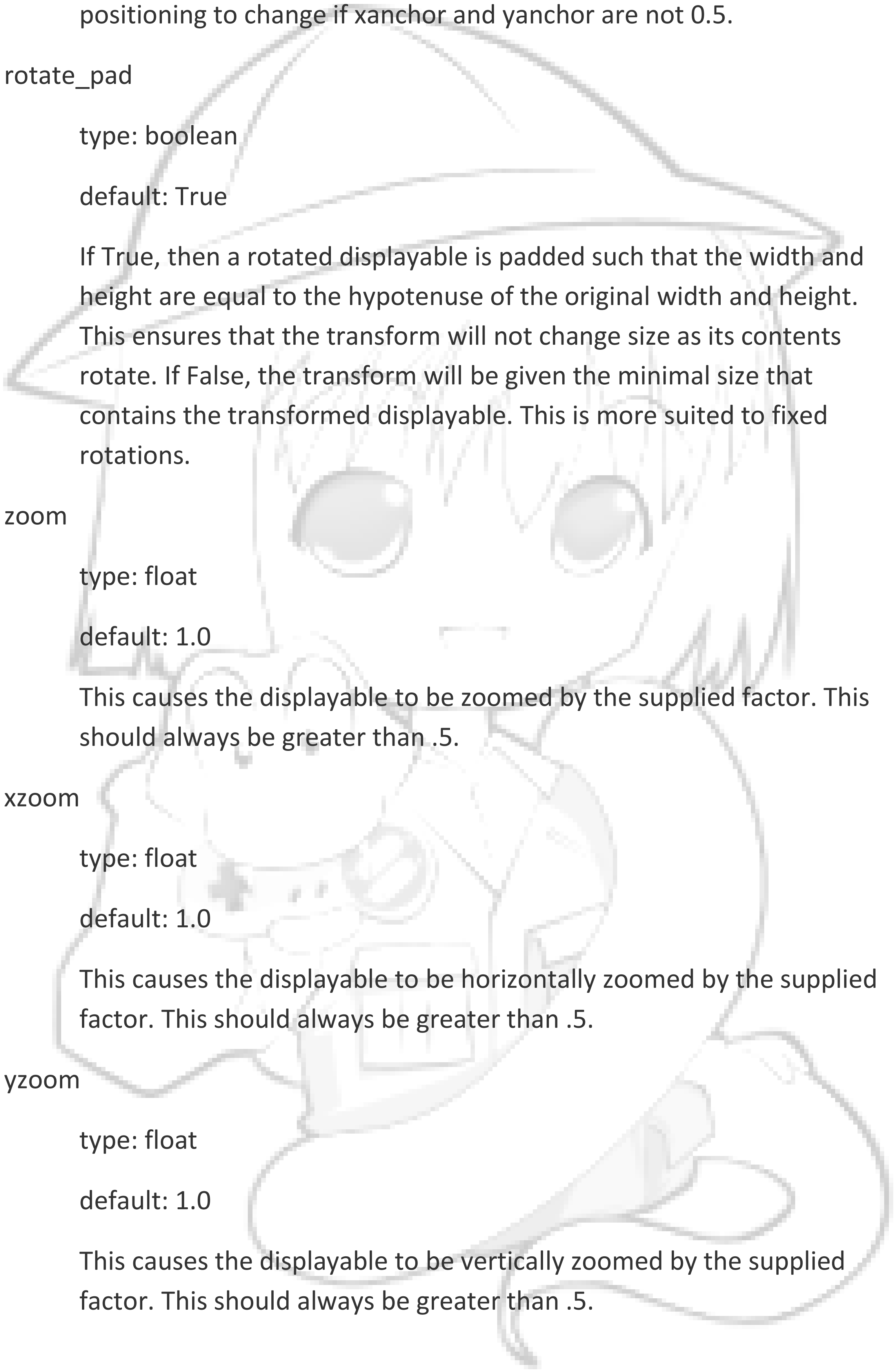
Equivalent to setting ypos and yanchor to this value.

rotate

type: float or None

default: None

If None, no rotation occurs. Otherwise, the image will be rotated by this many degrees clockwise. Rotating the displayable causes it to be



resized, according to the setting of rotate_pad, below. This can cause positioning to change if xanchor and yanchor are not 0.5.

rotate_pad

type: boolean

default: True

If True, then a rotated displayable is padded such that the width and height are equal to the hypotenuse of the original width and height. This ensures that the transform will not change size as its contents rotate. If False, the transform will be given the minimal size that contains the transformed displayable. This is more suited to fixed rotations.

zoom

type: float

default: 1.0

This causes the displayable to be zoomed by the supplied factor. This should always be greater than .5.

xzoom

type: float

default: 1.0

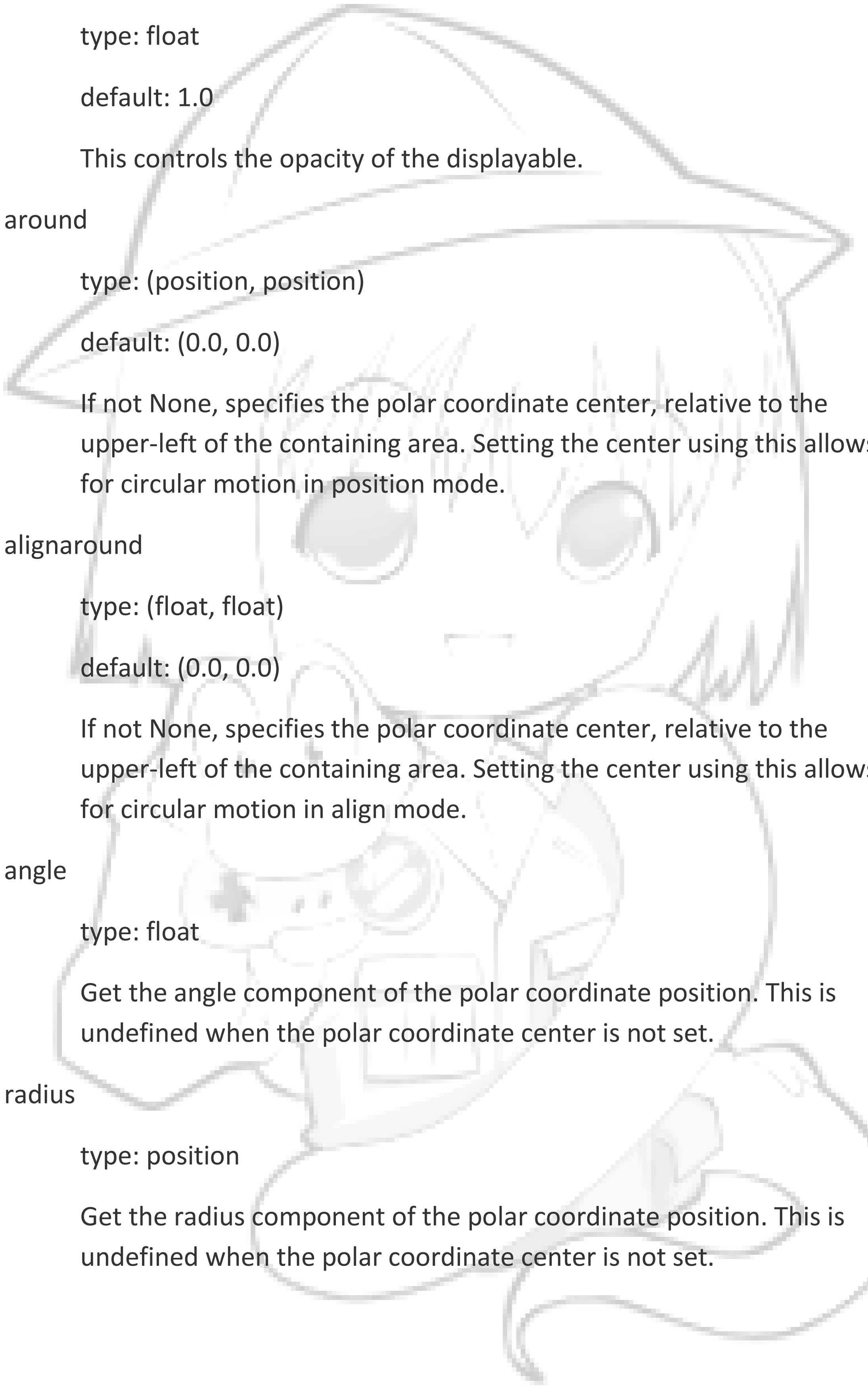
This causes the displayable to be horizontally zoomed by the supplied factor. This should always be greater than .5.

yzoom

type: float

default: 1.0

This causes the displayable to be vertically zoomed by the supplied factor. This should always be greater than .5.



alpha

type: float

default: 1.0

This controls the opacity of the displayable.

around

type: (position, position)

default: (0.0, 0.0)

If not None, specifies the polar coordinate center, relative to the upper-left of the containing area. Setting the center using this allows for circular motion in position mode.

alignaround

type: (float, float)

default: (0.0, 0.0)

If not None, specifies the polar coordinate center, relative to the upper-left of the containing area. Setting the center using this allows for circular motion in align mode.

angle

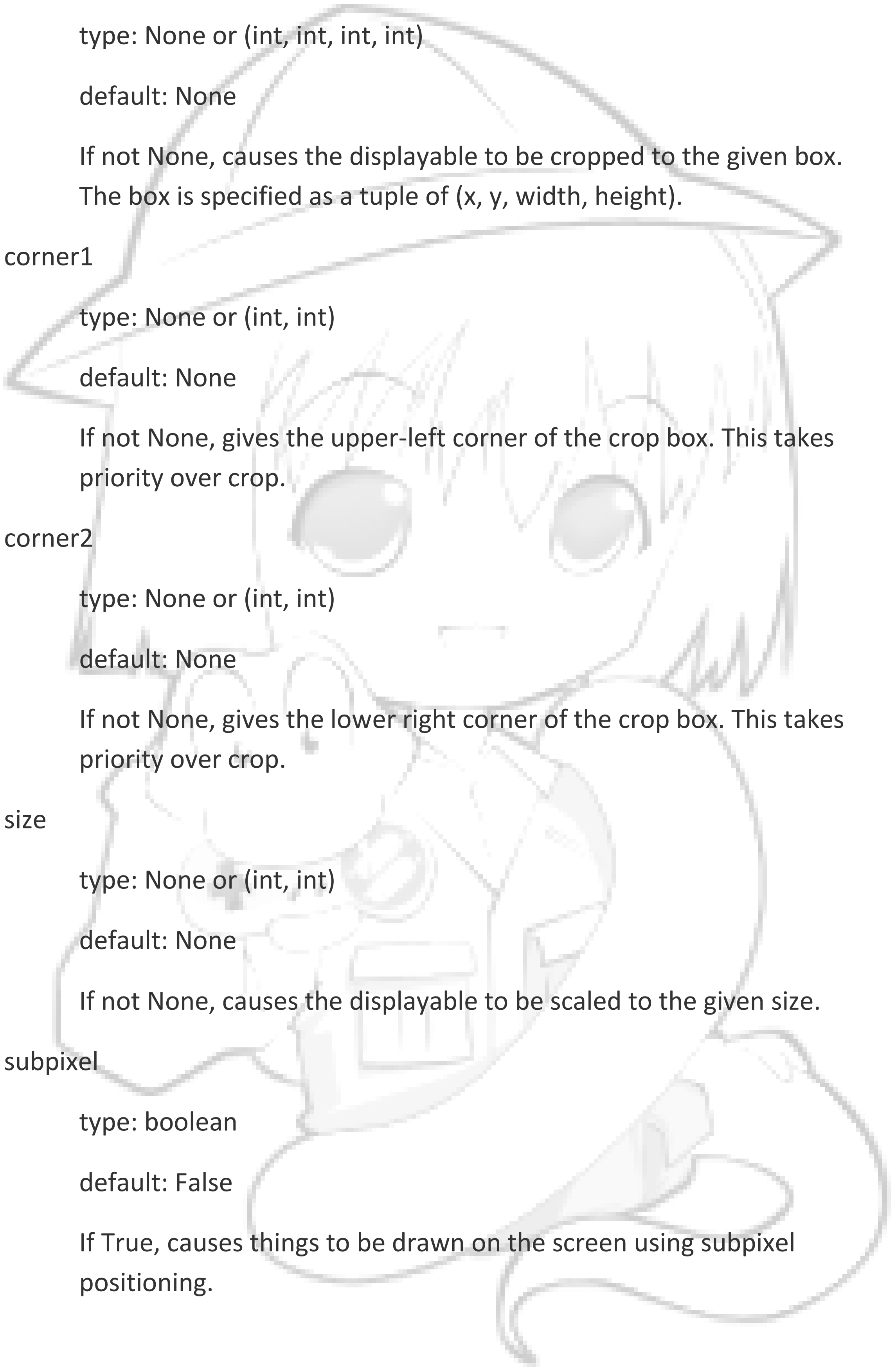
type: float

Get the angle component of the polar coordinate position. This is undefined when the polar coordinate center is not set.

radius

type: position

Get the radius component of the polar coordinate position. This is undefined when the polar coordinate center is not set.



crop

type: None or (int, int, int, int)

default: None

If not None, causes the displayable to be cropped to the given box. The box is specified as a tuple of (x, y, width, height).

corner1

type: None or (int, int)

default: None

If not None, gives the upper-left corner of the crop box. This takes priority over crop.

corner2

type: None or (int, int)

default: None

If not None, gives the lower right corner of the crop box. This takes priority over crop.

size

type: None or (int, int)

default: None

If not None, causes the displayable to be scaled to the given size.

subpixel

type: boolean

default: False

If True, causes things to be drawn on the screen using subpixel positioning.

delay

type: float

default: 0.0

If this transform is being used as a transition, then this is the duration of the transition.

These properties are applied in the following order:

1. crop, corner1, corner2
2. size
3. rotate
4. zoom, xzoom, yzoom
5. position properties

Circular Motion

When an interpolation statement contains the "clockwise" or "counterclockwise" keywords, the interpolation will cause circular motion. Ren'Py will compare the start and end locations and figure out the polar coordinate center. Ren'Py will then compute the number of degrees it will take to go from the start angle to the end angle, in the specified direction of rotation. If the circles clause is given, Ren'Py will ensure that the appropriate number of circles will be made.

Ren'Py will then interpolate the angle and radius properties, as appropriate, to cause the circular motion to happen. If the transform is in align mode, setting the angle and radius will set the align property. Otherwise, the pos property will be set.

External Events

The following events can triggered automatically:

start

A pseudo-event, triggered on entering an on statement, if no event of higher priority has happened.

show

Triggered when the transform is shown using the show or scene statement, and no image with the given tag exists.

replace

Triggered when transform is shown using the show statement, replacing an image with the given tag.

hide

Triggered when the transform is hidden using the hide statement. The image will not actually hide until the ATL block finishes.

hover

idle

selected_hover

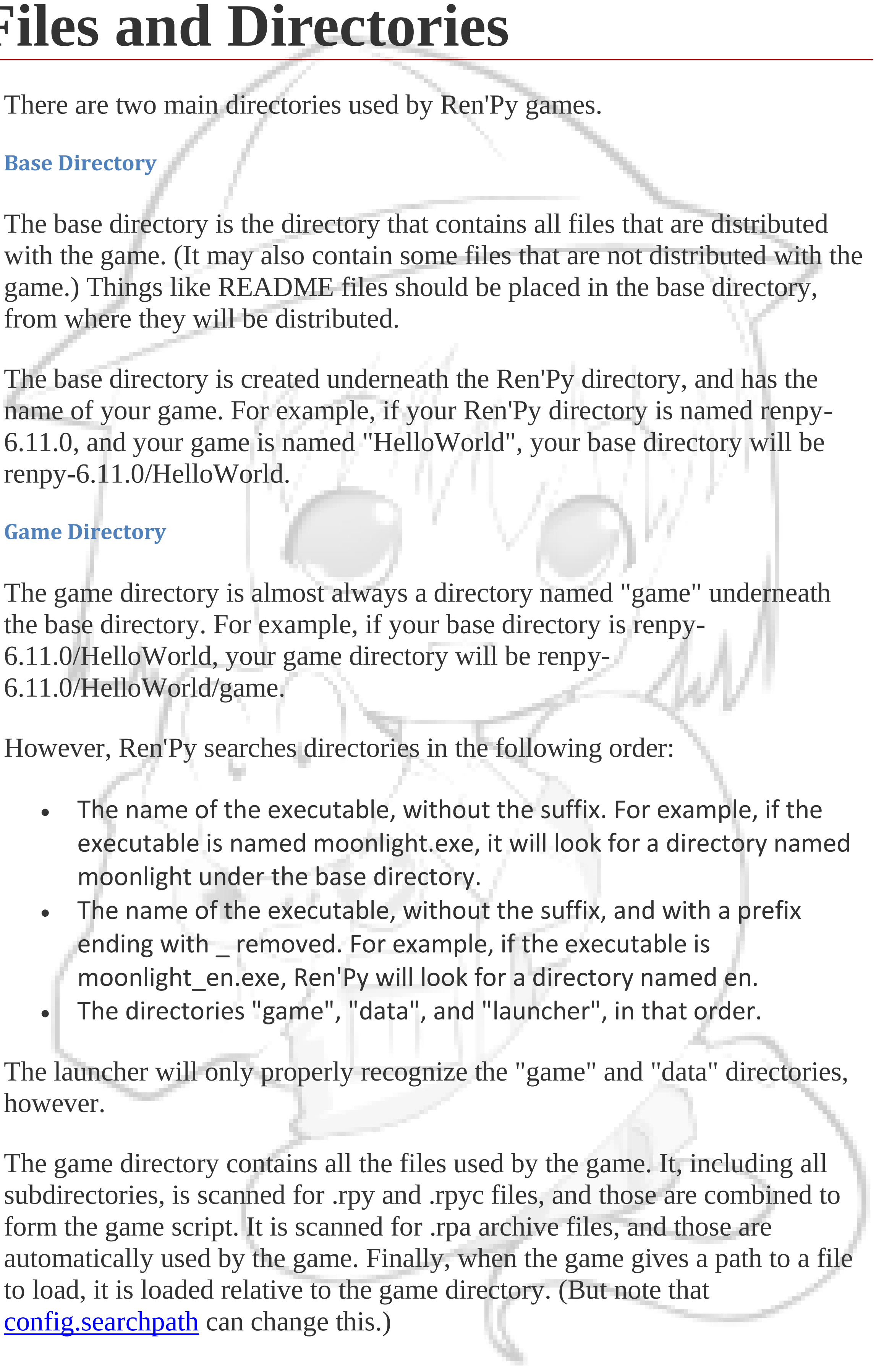
selected_idle

Triggered when button containing this transform, or a button contained by this transform, enters the named state.

Python Equivalent

The Python equivalent of an ATL transform is a Transform. There is no way to create ATL code programatically.

Files and Directories



There are two main directories used by Ren'Py games.

Base Directory

The base directory is the directory that contains all files that are distributed with the game. (It may also contain some files that are not distributed with the game.) Things like README files should be placed in the base directory, from where they will be distributed.

The base directory is created underneath the Ren'Py directory, and has the name of your game. For example, if your Ren'Py directory is named renpy-6.11.0, and your game is named "HelloWorld", your base directory will be renpy-6.11.0/HelloWorld.

Game Directory

The game directory is almost always a directory named "game" underneath the base directory. For example, if your base directory is renpy-6.11.0/HelloWorld, your game directory will be renpy-6.11.0/HelloWorld/game.

However, Ren'Py searches directories in the following order:

- The name of the executable, without the suffix. For example, if the executable is named moonlight.exe, it will look for a directory named moonlight under the base directory.
- The name of the executable, without the suffix, and with a prefix ending with _ removed. For example, if the executable is moonlight_en.exe, Ren'Py will look for a directory named en.
- The directories "game", "data", and "launcher", in that order.

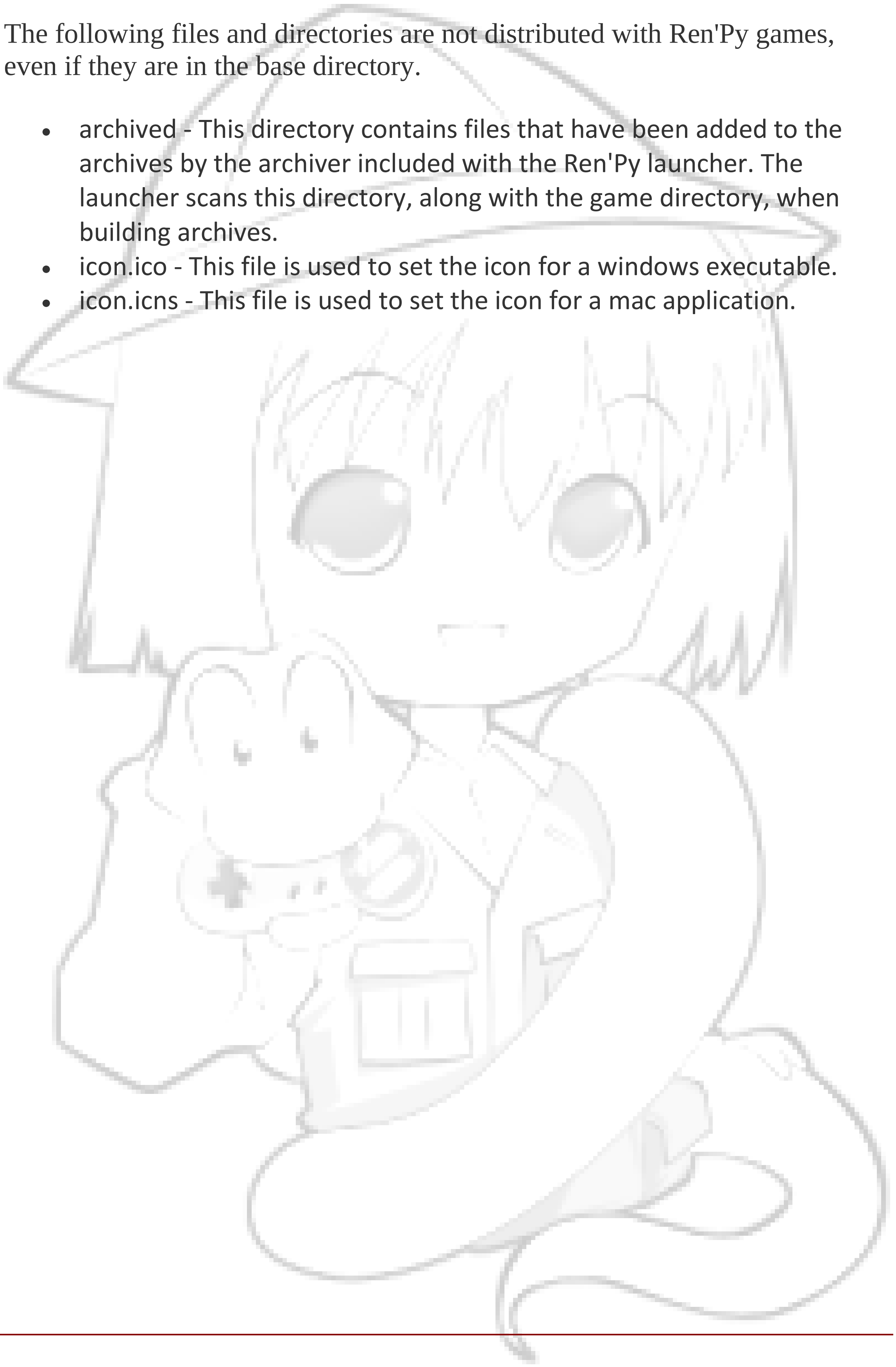
The launcher will only properly recognize the "game" and "data" directories, however.

The game directory contains all the files used by the game. It, including all subdirectories, is scanned for .rpy and .rpyc files, and those are combined to form the game script. It is scanned for .rpa archive files, and those are automatically used by the game. Finally, when the game gives a path to a file to load, it is loaded relative to the game directory. (But note that config.searchpath can change this.)

Ignored Files

The following files and directories are not distributed with Ren'Py games, even if they are in the base directory.

- archived - This directory contains files that have been added to the archives by the archiver included with the Ren'Py launcher. The launcher scans this directory, along with the game directory, when building archives.
- icon.ico - This file is used to set the icon for a windows executable.
- icon.icns - This file is used to set the icon for a mac application.



Defining Characters

When showing dialogue to the user using the two-argument form of the say statement, the first argument to the say statement should almost always be a variable bound to a Character object. These objects implement (or call functions that implement) much of the logic required to show dialogue to the user.

Characters are declared by binding variables to the Character objects. Such declarations should take place inside init blocks, to ensure that the bindings are not saved with the game state, as this would prevent revised versions of a game from picking up changes to the init block. Perhaps the simplest definition is:

```
init:
    $ e = Character("Eileen")
```

This creates a new character object and binds it to the \`e\` variable. When the say statement:

```
e "Hello, World!"
```

runs, the line of text "Hello, World!" will be preceded by the label "Eileen", indicating who is talking. It's common to customize the color of this label. This can be done by supplying the \`color\` keyword argument to Character, as in:

```
init:
    $ e = Character("Eileen", color=(200, 255, 200, 255))
```

The color argument is actually setting the color property on the label. The color property takes an RGBA tuple consisting of four numbers, enclosed in parenthesis and separated by commas, with each number ranging from 0 to 255.

The Character function is defined as follows:

Function: **Character** (name, kind=adv, **kwargs):

Creates a new character object. This object is suitable for use as the first argument of a two-argument say statement. When it is used as such, it shows a window containing the character name and the second argument to the say statement. This behavior is customized by passing parameters to this function.

name is the name of the character that is speaking. It can be either a string containing the character name, or None to indicate that the label should not be shown. A name value consisting of a single space (" ") indicates that the label should be a blank line, which allows a narrator character to line up narration with character dialogue. name is the only requirement to Character.

kind, if supplied, gives the name of another Character object. The default values for all the settings of the newly-created character object are taken from that character object. If not given, defaults to `adv`. Another useful value is `nvl`, which by default causes nvl-mode interactions.

Keyword Arguments. In addition to name, `Character` takes keyword arguments that control its behavior.

dynamic - If true, then *name* is interpreted as a string containing a python expression that is evaluated to yield the name of the character.

image - If true, then *name* is interpreted as an image. This image is added to the dialogue window in place of the label.

condition - If present, this should be a string containing a python expression. This expression is evaluated whenever a line is said through this character object. If it evaluates to false, the line is not shown to the user.

interact - If true, the default, causes an interaction to occur when a line is shown to the user. If false, the interaction does not take place, and ui.interact (or some other means) must be called to cause an interaction to occur.

with_none - If True, causes a "with None" statement to be run after each interaction. If None (the default), checks config.implicit_with_none to determine if a "with None" should be run.

type - The type of interaction. See renpy.last_interact_type for how this is used.

Prefixes and Suffixes. The following keyword arguments can be used to add a prefix or suffix to everything said through a character object. These can be used when lines of dialogue need to be enclosed in quotes, as the preferred alternative to adding those quotes to every line of dialogue in the script.

who_prefix - Text that is prepended to the name of the character when forming the label of the dialogue.

who_suffix - Text that is appended to the name of the character when forming the label of the dialogue.

what_prefix - Text that is prepended to the line of dialogue before it is shown to the user.

what_suffix - Thext that is appended to the line of dialogue before it is shown to the user.

Click-to-continue. These keyword arguements are used to control the click-to-continue indicator:

ctc - If present, this argument takes a displayable that is used as the click-to-continue indicator. If not present or None, then no click-to-continue indicator is displayed.

ctc_pause - If given, a click-to-continue indicator that's shown when the display of text has been paused with the {p} or {w} text tags.

ctc_timedpause - If given and not None, a click-to-continue indicator that is displayed when a {p} or {w} tag has a time associated with it. One can use ctc_pause=Null() to disable ctc_pause only in the case of a timed pause.

ctc_position - If "nestled", the click-to-continue indicator is displayed nestled in with the end of the text. If "fixed", the click-to-continue indicator is displayed directly on the screen, with its various position properties determining where it is actually shown.

Functions. The following are keyword arguments that allow one to massively customize the behavior of a character object:

show_function - The function that is called to display each step of a dialogue to the user. (A dialogue may be broken down into steps by pause text tags.) It should have the same signature as renpy.show_display_say.

predict_function - The function that is called to predict the images from this dialogue. It should have the signature of renpy.predict_display_say.

callback - A callback or list of callbacks that are called at various points during the the process of displaying dialogue. The callback is called with a single positional argument, which indicates the event that occured, and the following keyword arguments:

- *interact* - True if this dialogue is being displayed interactively.
- *type* - The value of *type* supplied to Character.

Additional keyword arguments and unknown events should be ignored, for future expansion. The events that cause the callback to be called are:

- "begin" - called at the start of showing dialogue.
- "show" - called before each segment of dialogue is shown.
- "show_done" - called after each segment of dialog is shown (but before the interaction occurs).
- "slow_done" - called when slow text finishes showing.
- "end" - called at the end of showing dialogue.

Note that when interact=False, slow_done can occur after end. If *callback* is not given or None, it defaults to config.character_callback, if not None. The character callbacks in config.all_character_callbacks are called for all characters, before any more specific callbacks.

Default Show Function The following are keyword arguments that are processed by the default show_function.

show_two_window - Places the name of the character saying the dialogue into its own window.

show_side_image - Specifies a displayable that is show with this dialogue. The displayable should set its position properties in a way to place it where the user wants it. Similarly, the window properties should be set in a way that provides room for the side image. The function ShowingSwitch may be useful for causing this displayable to vary based on what other images are visible.

show_two_window_vbox_properties - If supplied, a dictionary containing properties that are added to the vbox containing both windows, when show_two_window is true.

show_who_window_properties - If supplied, a dictionary containing properties that are added to the window containing the name of who is speaking, when show_two_window is true.

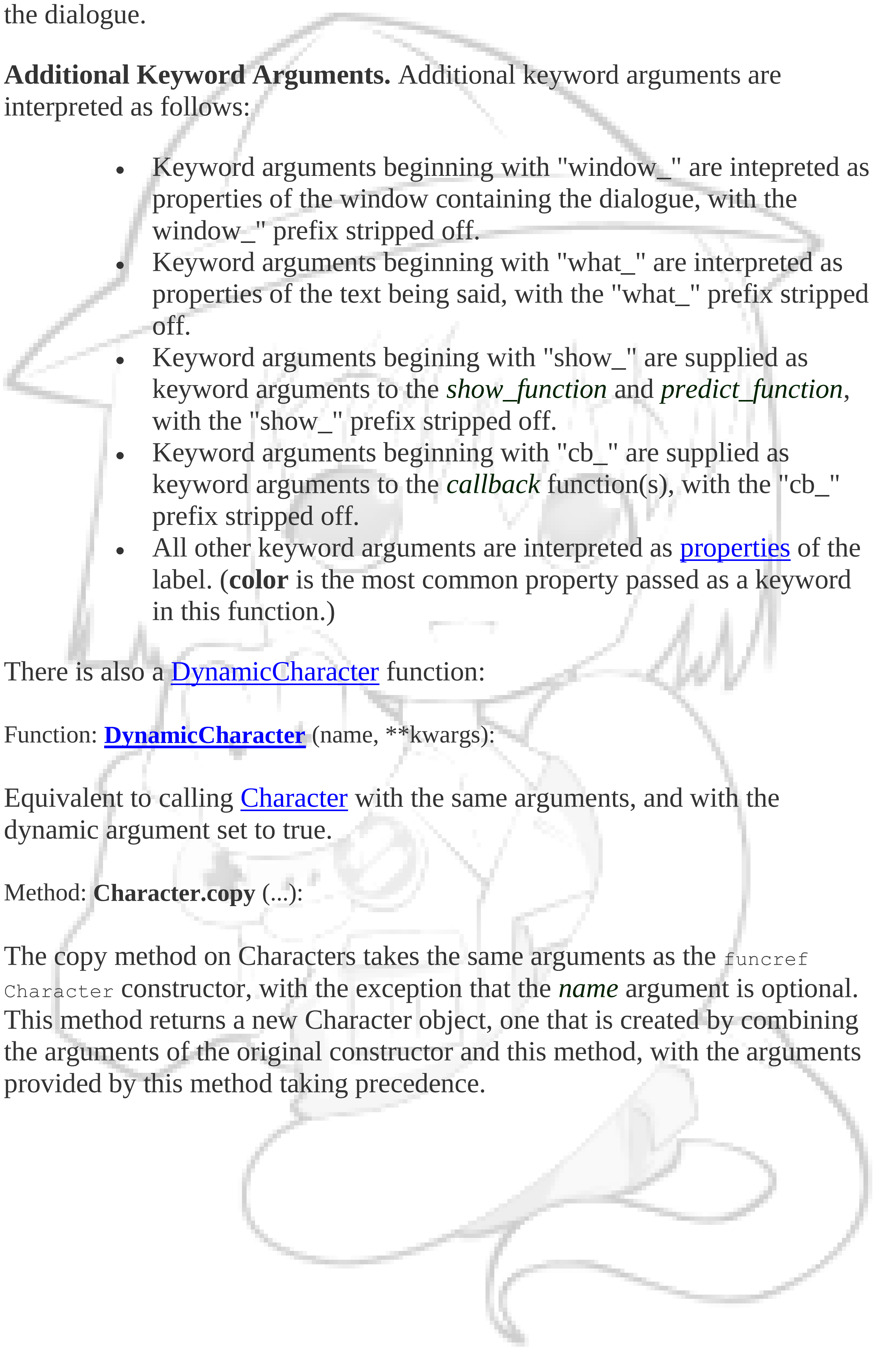
show_say_vbox_properties - If supplied, a dictionary containing properties that are added to the vbox inside the main window.

show_transform - If supplied, a transform that is applied to window(s) being shown.

Styles. The following keyword arguments control the styles used by parts of the dialogue:

who_style - Defaults to 'say_label', the style of the label.

what_style - Defaults to 'say_dialogue', the style of the text being said.



window_style - Defaults to 'say_window', the style of the window containing the dialogue.

Additional Keyword Arguments. Additional keyword arguments are interpreted as follows:

- Keyword arguments beginning with "window_" are intepreted as properties of the window containing the dialogue, with the window_" prefix stripped off.
- Keyword arguments beginning with "what_" are interpreted as properties of the text being said, with the "what_" prefix stripped off.
- Keyword arguments begining with "show_" are supplied as keyword arguments to the *show_function* and *predict_function*, with the "show_" prefix stripped off.
- Keyword arguments beginning with "cb_" are supplied as keyword arguments to the *callback* function(s), with the "cb_" prefix stripped off.
- All other keyword arguments are interpreted as properties of the label. (**color** is the most common property passed as a keyword in this function.)

There is also a DynamicCharacter function:

Function: **DynamicCharacter** (name, **kwargs):

Equivalent to calling Character with the same arguments, and with the dynamic argument set to true.

Method: **Character.copy** (...):

The copy method on Characters takes the same arguments as the `funcref Character` constructor, with the exception that the *name* argument is optional. This method returns a new Character object, one that is created by combining the arguments of the original constructor and this method, with the arguments provided by this method taking precedence.

Calling Character Objects

Character objects may be called directly, as if they were functions. They take one positional parameter, the line of dialogue to be show to the user. They also take keyword arguments:

- *interact*, which determines if an interaction should take place.
- *force_ctc*, forces the click-to-continue indicator to be displayed even if interact is False.

This use can programatically replace say statements. For example, the say statement:

```
e "Hello, World!"
```

is equivalent to:

```
$ e("Hello, World!", interact=True)
```

Pre-Defined Characters

We give you a few pre-defined characters to work with.

- **centered** is a character that will cause what it says to be displayed centered,

in the middle of the screen, outside of any window.

- **extend** will cause the last character to speak to say a line of dialogue consisting of the last line of dialogue spoken, "{fast}", and the dialogue given to extend. This can be used to have the screen change over the course of dialogue. Extend is aware of NVL-mode, and treats it correctly.

```
# Show the first line of dialogue, wait for a click, change
expression, and show
# the rest.

show eileen concerned
e "Sometimes, I feel sad."
show eileen happy
extend " But I usually quickly get over it!"

# Similar, but automatically changes the expression when the first
line is finished
# showing. This only makes sense when the user doesn't have text speed
set all the
# way up.
```

```
show eileen concerned
e "Sometimes, I feel sad.{nw}"
show eileen happy
extend " But I usually quickly get over it!"
```

Text

This section covers aspects of text in Ren'Py. It first covers interpolation, supported by the say and menu statements, which allows values to be substituted into text. It next discusses text tags, which allow the style of portions of strings of text to be customized. Finally, it covers how Ren'Py handles fonts.

Interpolation

Interpolation is supported by the `say` and `menu` statements. These statements support python string interpolation over the contents of the store. The strings used by the statements support conversion specifiers of the form `%(variable)s`, where `variable` is the name of a variable and `s` is a conversion. Useful conversions include 's', which interpolates a string; 'd', which interpolates an integer; and 'f', which interpolates a floating point number. Conversions may also include characters that modify how the string is converted. More information about conversions can be found at the Python string formatting operations reference.

In strings where interpolation is supported, percent characters (%) must be duplicated (to %%) to prevent them from being interpreted as introducing interpolation.

```
$ name = 'Samantha'
$ age = 19
$ withyou = 110

girl "My name is %(name)s, and I am %(age)d years old. I'm with you
%(withyou)d%%"
```

When interpolation is not supported, the effect can often be faked by putting placeholder %s, %d, and %f with % (variable list) outside the string (see example below).

```
ui.text("%s's Vital Statistics: %d" % (name, blood))
```

Text tag processing is performed after interpolation, so it's important to ensure interpolation does not introduce text tags.

Text Tags

Text displayed by Ren'Py supports text tags. While styles can only be applied to an entire Text displayable, allow only a portion of the text in the displayable to be customized. As text tags are part of the Text displayable, they may be used in any string that is displayed on the screen. However, some of the text tags will only have effect if used in the appopriate context.

Text tags should be used fairly sparingly. If you find you're using text tags on every line of the game, it's quite possible text tags are not the right solution for the problem at hand. Investigate text styles as a possible alternative to text tags.

Text tags start with a left brace ({), and continue to the matching right brace (}). Immediately following the left brace is the tag name. The name may be followed by an argument, which is separated from the tag name by an equals sign (=). Some tags require an argument, while others require that the argument be omitted. Text tags are sensitive to case and white space.

Some tags require a corresponding closing tag. A closing tag is any text tag where the name begins with a slash (/). Closing tags should be properly nested: "{b}{i}this is okay{/i}{/b}", while "{b}{i}this is wrong{/b}{/i}". While improper nesting of text tags will generally not cause an error, this problem may cause undesirable rendering results. The text between a tag and the corresponding closing tag is called the enclosed text. All tags must be closed by the end of the text string.

To include a single left brace in text, two left braces ({{) musty be included instead.

Ren'Py supports the following text tags:

- {a=*argument*} and {/a} cause their enclosed text to be rendered as a hyperlink. The enclosed text may contain text tags, but may not contain another hyperlink. A hyperlink is rendered as a button containing the enclosed text. The text in the button does not participate in line-breaking. Rather, if the text grows too big for the line it is on, the whole button is moved to the start of the next line.

  When a a hyperlink is clicked, the callback function defined in config.hyperlink_callback is called with the text of the argument of the hyperlink.
  The default hyperlink checks to see if the hyperlink begins with "http:". If it does, the link is opened in the user's web browser. If not, the hyperlink is interpreted as a label, which is called in a new context. This is an appropriate behavior when using hyperlinks to definitions of unfamiliar terms.
  Hyperlinks should not be used as a general control-flow tool, as they transfer control to a new context when clicked. If the user saves while

inside that context, when he loads the game he will be returned to the screen containing the hyperlink.

```
init:
    $ definition = Character(None, window_yfill=True,
window_xmargin=20,
                             window_ymargin=20,
window_background=Solid((0, 0, 0, 192)))

label start:

    "A game that instructs on how to make a game? Isn't that a sort of
{a=define_quine}Quine{/a}?"

    # ...

label define_quine:

    definition "Quine:\n\nA program that prints itself to its output."

    return
```

- {b}text{/b} renders the enclosed text in a bold font.
- {color=spec}text{/color} changes the color of the enclosed text. The color may be a hex triple or a hex sextuple, optionally preceded by a hash mark. (The same format that color() accepts. f00, ff0000, #f00, and #ff0000 are all valid representations of red.)
- {fast} causes the immediated display of text before it. It can be used to specify where slow text shold begin to display from. It only has an effect if the text speed preference is not infinite. Use this if you have two lines of dialogue where the second is an extension of the first, as a variant of the pause effect that allows for changes to occur while paused. This tag does not take a closing tag.
- {font=filename.ttf}text{/font} renders the enclosed text in the supplied font.
- {i}text{/i} makes the enclosed text in an italic font.
- {image=filename} causes the supplied image to be loaded and included into text. The image should are treated as if they were text, and should not be taller than a single line of text. {image} does not take a closing tag.
- {nw}, at the end of a block of text, causes the current interaction to terminate immediately once the text is fully shown. It only makes sense to use this in dialogue text, where the effect is to cause slow text to dismiss the interaction once it has been fully displayed.
- {p} causes the display of text to be paused until the user clicks to continue, or until an auto-forward occurs. This allows text to be displayed incrementally. {p} does not take closing tags, and causes a newline to be inserted after the given text. The logic to handle pausing is implemented in the Character object, and may not work for other text widgets.

- {plain}text{/plain} makes the enclosed text plain, eliminating bold, italic, and underline styles
- {s}text{/s} makes the encolosed text struck-through. *(new in 6.6.2)*
- {size=spec}text{/size} changes the size of the text. The supplied spec may be a number, in which case it is the number of pixels high that the text will be. If it is a number preceded by a plus (like "+10"), it means to increase the size by that number of pixels, while if it is preceded by a minus it means to decrease.
- {u}text{/u} renders the enclosed text with an underline.
- {w} is similar to {p}, except that it does not cause a newline. We also support the form {w=number}, which pauses for *number* seconds. (For example, {w=.5} pauses for .5 seconds before continuing.
- {=*style*}text{/=*style*}} applies the named style to the enclosed text. Think of this as a way to design your own text tags. *(new in 6.6.2)*

The last {fast} tag is the one that is used. {nw}, {p}, and {w} tags only take effect if they are after the last {fast} tag in the text.

The following is an example of a say statement that uses many text tags. Use of this many text tags is not recommended in a high-quality game.

```
"Using text tags, we can make text {size=+12}bigger{/size} or
 {size=-8}smaller{/size}. We can make it {b}bold{/b}, {i}italic{/i},
 or {u}underlined{/u}. We can even change its
{color=#f88}color{/color}."
```

Fonts

The Text displayable attempts to find an an appropriate font using information about the font name, boldness, italics, underline and size. This information is supplied to the Text displayable using style properties, but may then be modified using text tags. Ren'Py translates this into a font using the following algorithm.

- A (name, boldness, italics) triple is looked up in config.font_replacement_map. If present, it is expected to map to a triple giving replacement values for name, boldness, and italics. This lookup is not recursive, so only one lookup is performed. This lookup allows specific bold or italic font shapes to be used.
- If a SFont has been registered with the same name, size, boldness, italics, and underline, that SFont is used.
- Otherwise, Ren'Py interprets the font name as the filename of a truetype font. This filename is searched for in the searchpath and archives. If found, it is used as the font file.
- Otherwise, Ren'Py searches for a font with the given filename in the system font directory. If found, that file is used.

- Otherwise, Ren'Py interprets the filename as the name of a truetype font, which is loaded if found.

The truetype font loading code will automatically scale and underline the font as required. If you have not provided bold and italic font mappings, it will also artificially thicken and slant the font when necessary.

For best results, fonts should be truetype files that ship with the game. This ensures that required fonts are always present on the user's system.

We support TrueType collections "0@font.ttc" is the first font in the collection, "1@font.ttc" the second, and so on. *(new in 6.10.0)*

Image-Based Fonts

Along with the usual TrueType fonts, Ren'Py supports image-based SFonts, MudgeFonts, and BMFonts.

Image-based fonts have several advantages and several disadvantages. One of the advantages is that these fonts are bitmap-based, which means it is easy to create custom fonts and to find free examples (in contrast, the licenses of high quality TrueType fonts rarely permit redistribution). Also, since the fonts are images, it's possible to apply effects to them that Ren'Py would not otherwise support. The downsides of image-based fonts come from the fact that Ren'Py doesn't render them, but instead merely copies characters out of them. Because of this, one needs to supply another image to Ren'Py if one wants the font to be scaled, made bold, made italic, or underlined. Ren'Py will recolor the image fonts, with white being mapped to the supplied color, and other colors interpolated on a channel-by-channel basis.

Please note that you must register an image-based font for each combination of font, size, bold, italic, and underline your game uses. They can then be used by setting the font property of a style to the name of the font.

To use SFonts, they must first be registered with Ren'Py using the renpy.register_sfont function. For more information about SFonts, see [1].

To use MudgeFonts, they must be registered with the renpy.register_mudgefont function. For more information about MudgeFonts, see [2].

To use BMFonts, they must be registered with the renpy.register_bmfont function. For more information about BMFonts, see [3]. We recommend BMFont for bitmap fonts, as BMFonts support unicode and proper kerning.

Legal issues regarding the use of copyrighted outline fonts in creating image-based fonts are discussed in Bitmap Fonts and Copyright.

Displayables

A **displayable** is a python object implementing an interface that allows it to be displayed to the screen. Displayables can be assigned to an image name using an `image` statement, and can then be shown to the user using the `show` and `hide` statements. They can be supplied as an argument to the ui.add function. They are also used as the argument to certain style properties. In this section, we will describe functions that create displayables.

When Ren'Py requires colors to be specified for a displayable, it allows them to be specified in a number of forms. In the following examples, lowercase letters stand for hexadecimal digits, while uppercase letters are short for numbers between 0 and 255. r, g, b, and a are short for red, green, blue, and alpha.

- A string of the form "#rgb". (yellow = "#ff0", blue = "#00f")
- A string of the form "#rgba". (yellow = "#ff0f", blue = "#00ff")
- A string of the form "#rrggbb". (yellow = "#ffff00", blue = "#0000ff")
- A string of the form "#rrggbbaa". (yellow = "#ffff00ff", blue = "#0000ffff")
- A 4-tuple of the form (R, G, B, A). (yellow = (255, 255, 0, 255), blue = (0, 0, 255, 255))

When Ren'Py expects a displayable to be specified, it allows them to be in the following forms:

- A displayable may be given directly, in which case it is used.
- A string beginning with "#" is used to create a solid object, with the color interpreted as above.
- Strings with at least one dot in them are interpreted as the filenames of images, and are used to create image objects.
- Other strings are interpreted as the name of an image defined with the image statement.

Image Manipulators

An Image is a type of displayable that contains bitmap data that can be shown to the screen. All images are displayables, but not all displayables are images. Images differ from displayables in that they can be statically computed, and stored in the image cache. This loading occurs sometime before the image is to be used (but not at the time the Image is created). This generally makes Images faster than arbitrary displayables. An image manipulator is a function that returns an Image.

When an image manipulator requires another as input, the second image manipulator can be specified in a number of ways.

- It may be supplied as a string, which is interpreted as a filename and loaded with im.Image.
- It may be a directly specified image manipulator, in which case it is passed through unchanged.

Run-length Encoding. Ren'Py supports the use of run-length encoding to efficiently draw images with large amounts of transparency. All image manipulators support an *rle* argument. When True, run-length encoding is used. When false, it is not used. When None (the default), Ren'Py will randomly sample 10 points in the image image to try to find transparency, and use rle only if transparency is found. In exchange for increased load time, and additional memory proportional to the number of non-transparent pixels in the image, run-length encoding can draw images to the screen in time proportional to the number of non-transparent pixels in the image. When an image is mostly transparent, this can lead to a significant speedup in drawing time.

```
init:
    image eileen happy = Image("eileen_happy.png", rle=True)
```

Caching. By default, image manipulators are cached in memory before they are loaded. Supplying the cache=False argument to an image manipulator will prevent this caching. This should only be used when an image manipulator is never used directly (such as by an image statement).

```
init:
    image eileen happy = im.AlphaMask(Image("eileen_happy.base.jpg", 
cache=False),
                                      Image("eileen_happy.mask.jpg", 
cache=False))
```

Image Manipulator List. The image manipulators are:

Function: **im.Image** (filename, **properties):

This image manipulator loads an image from a file.

filename - The filename that the image will be loaded from.

Note that Image is an alias for im.Image.

Function: **Image** (...):

An alias for im.Image.

Function: **im.Alpha** (image, alpha, **properties):

Returns an alpha-mapped version of the image. Alpha is the maximum alpha that this image can have, a number between 0.0 (fully transparent) and 1.0 (opaque).

If an image already has an alpha channel, values in that alpha channel are reduced as appropriate.

Function: **im.AlphaMask** (base, mask, **properties):

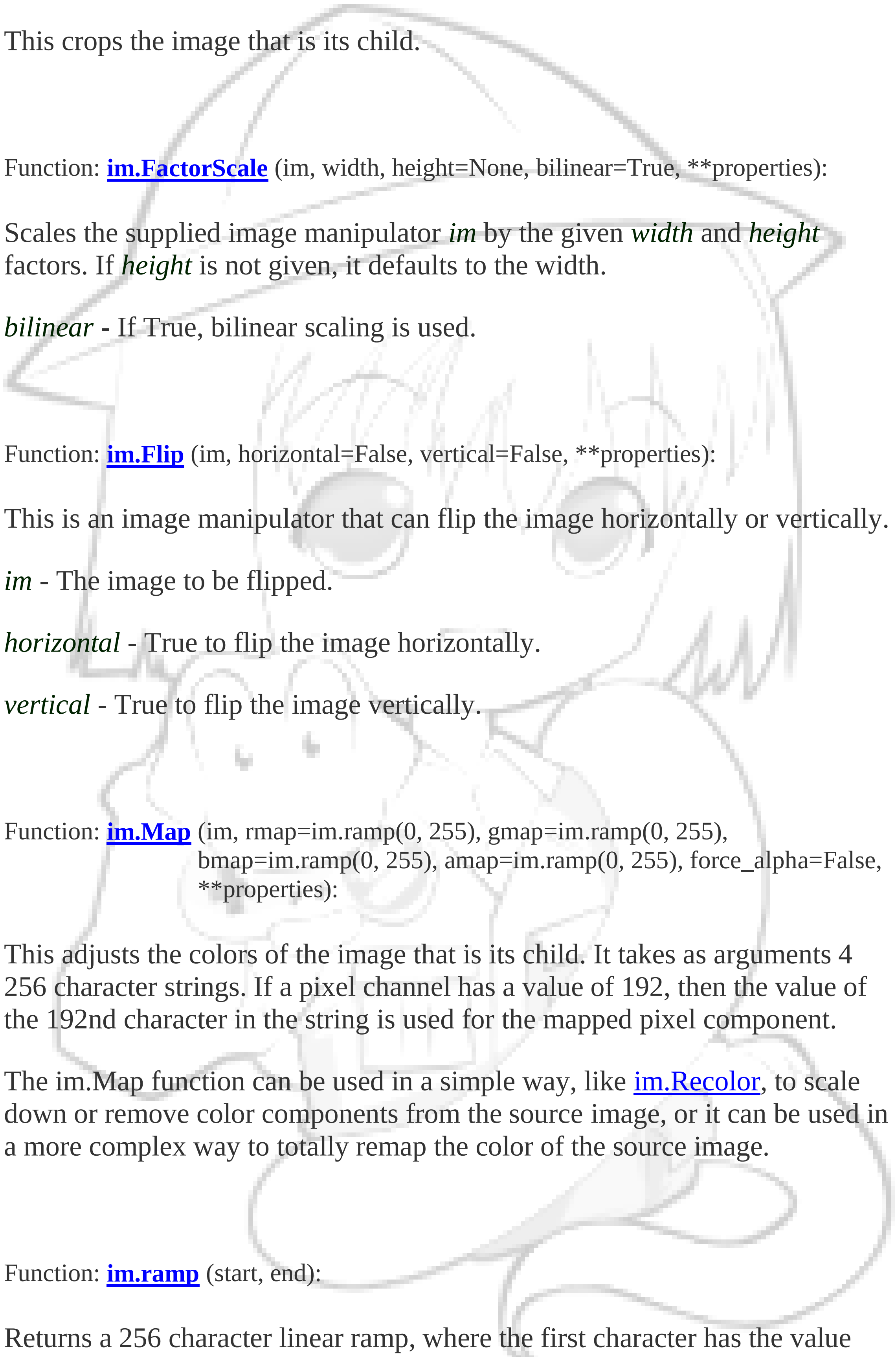
base and *mask* should be image manipulators. This function takes the red channel from *mask*, and applies it to the alpha channel of *base* to create a new image.

Function: **im.Composite** (size, *args, **properties):

This image manipulator composites one or more images together.

This takes a variable number of arguments. The first argument is size, which is either the desired size of the image (in pixels), or None to indicate that the size should be the size of the first image.

It then takes an even number of further arguments. (For an odd number of total arguments.) The second and other even numbered arguments contain position tuples, while the third and further odd-numbered arguments give images (or image manipulators). A position argument gives the position of the image immediately following it, with the position expressed as a tuple giving an offset from the upper-left corner of the image. The images are composited in bottom-to-top order, with the last image being closest to the user.



Function: **im.Crop** (im, x, y, w, h, **properties):

This crops the image that is its child.

Function: **im.FactorScale** (im, width, height=None, bilinear=True, **properties):

Scales the supplied image manipulator *im* by the given *width* and *height* factors. If *height* is not given, it defaults to the width.

bilinear - If True, bilinear scaling is used.

Function: **im.Flip** (im, horizontal=False, vertical=False, **properties):

This is an image manipulator that can flip the image horizontally or vertically.

im - The image to be flipped.

horizontal - True to flip the image horizontally.

vertical - True to flip the image vertically.

Function: **im.Map** (im, rmap=im.ramp(0, 255), gmap=im.ramp(0, 255), bmap=im.ramp(0, 255), amap=im.ramp(0, 255), force_alpha=False, **properties):

This adjusts the colors of the image that is its child. It takes as arguments 4 256 character strings. If a pixel channel has a value of 192, then the value of the 192nd character in the string is used for the mapped pixel component.

The im.Map function can be used in a simple way, like im.Recolor, to scale down or remove color components from the source image, or it can be used in a more complex way to totally remap the color of the source image.

Function: **im.ramp** (start, end):

Returns a 256 character linear ramp, where the first character has the value start and the last character has the value end. Such a ramp can be used as a map argument of im.Map.

Function: **im.Scale** (im, width, height, bilinear=True, **properties):

This is an image manipulator that scales another image manipulator to the specified *width* and *height*.

bilinear - If True, bilinear interpolation is used. If False, nearest-neighbor filtering is used.

Function: **im.Recolor** (im, rmul=255, gmul=255, bmul=255, amul=255, force_alpha=False, **properties):

This adjusts the colors of the supplied image, *im*. It takes four arguments, *rmul*, *gmul*, *bmul* and *amul*, corresponding to the red, green, blue and alpha channels, with each being an integer between 0 and 255. Each channel has its value mapped so that 0 is 0, 255 is the argument supplied for that channel, and other values are linearly mapped in-between.

Function: **im.Twocolor** (im, white, black, force_alpha=False, **properties):

This takes as arguments two colors, white and black. The image is mapped such that pixels in white have the white color, pixels in black have the black color, and shades of gray are linearly interpolated inbetween. The alpha channel is mapped linearly between 0 and the alpha found in the white color, the black color's alpha is ignored.

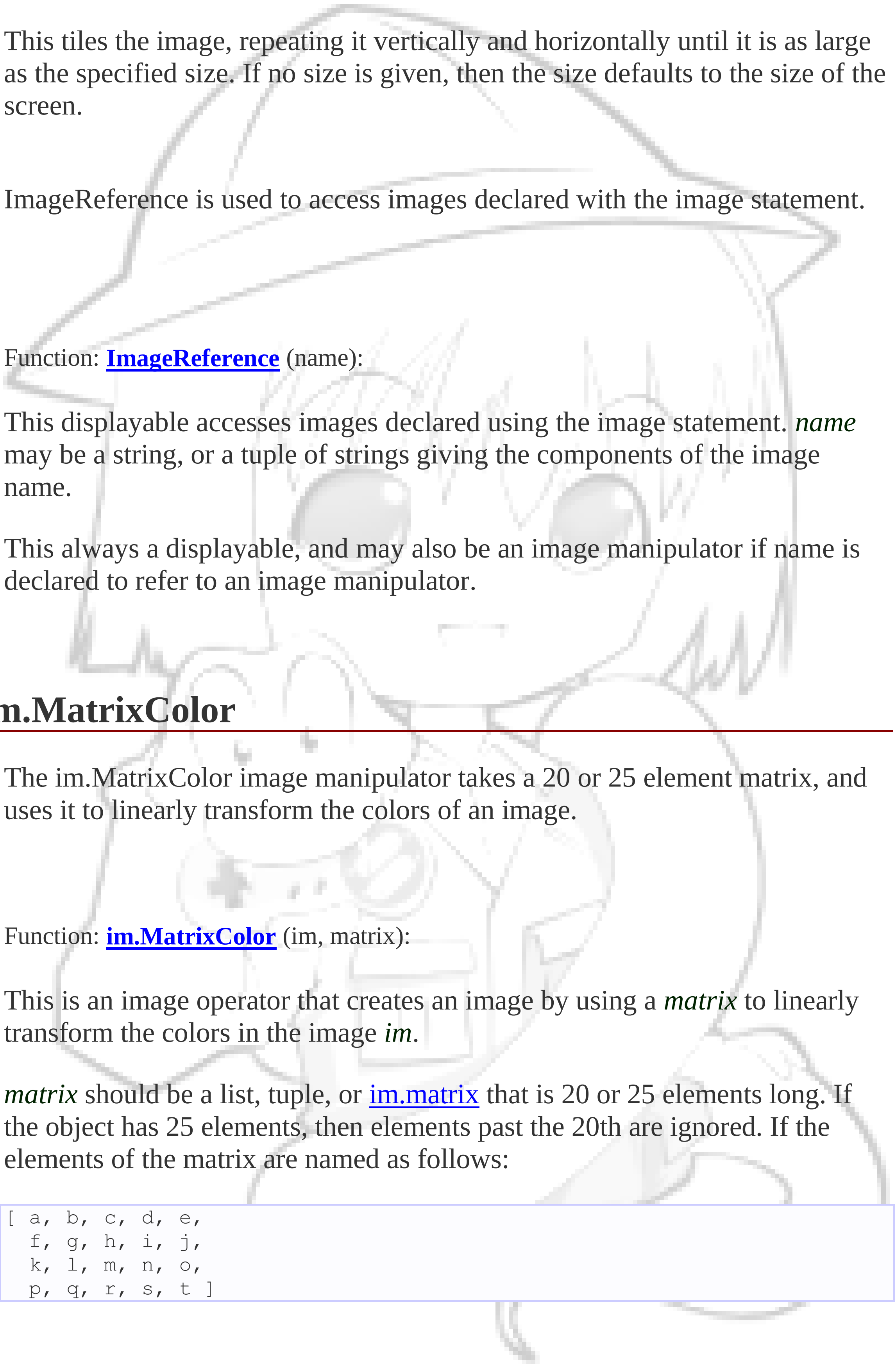
Function: **im.Rotozoom** (im, angle, zoom, **properties):

This is an image manipulator that is a smooth rotation and zoom of another image manipulator.

im - The image to be rotozoomed.

angle - The number of degrees counterclockwise the image is to be rotated.

zoom - The zoom factor. Numbers that are greater than 1.0 lead to the image becoming larger.



Function: **im.Tile** (im, size=None, **properties):

This tiles the image, repeating it vertically and horizontally until it is as large as the specified size. If no size is given, then the size defaults to the size of the screen.

ImageReference is used to access images declared with the image statement.

Function: **ImageReference** (name):

This displayable accesses images declared using the image statement. *name* may be a string, or a tuple of strings giving the components of the image name.

This always a displayable, and may also be an image manipulator if name is declared to refer to an image manipulator.

im.MatrixColor

The im.MatrixColor image manipulator takes a 20 or 25 element matrix, and uses it to linearly transform the colors of an image.

Function: **im.MatrixColor** (im, matrix):

This is an image operator that creates an image by using a *matrix* to linearly transform the colors in the image *im*.

matrix should be a list, tuple, or im.matrix that is 20 or 25 elements long. If the object has 25 elements, then elements past the 20th are ignored. If the elements of the matrix are named as follows:

```
[ a, b, c, d, e,
  f, g, h, i, j,
  k, l, m, n, o,
  p, q, r, s, t ]
```

and R, G, B, and A are the red, green, blue, and alpha components of the color, respectively, then the transformed color R', G', B', A' is computed as follows:

```
R' = (a * R) + (b * G) + (c * B) + (d * A) + (e * 255)
G' = (f * R) + (g * G) + (h * B) + (i * A) + (j * 255)
B' = (k * R) + (l * G) + (m * B) + (n * A) + (o * 255)
A' = (p * R) + (q * G) + (r * B) + (s * A) + (t * 255)
```

R', G', B', and A' are clamped to the range [0, 255].

It's often convenient to specify the matrix using an im.matrix object. These objects support a number of mathematical operations, such as matrix and scalar multiplication and scalar addition. Multiplying matrices together lets you perform multiple color manipulations at once, at the cost of a single MatrixColor operation.

Function: **im.matrix** (matrix):

Constructs an im.matrix object from the given matrix. im.matrix objects represent 5x5 matrices, and support a number of mathematical operations. The operations supported are matrix multiplication, scalar multiplication, element-wise addition, and element-wise subtraction. These operations are invoked using the standard mathematical operators (*, *, +, and -), respectively. If two im.matrix objects are multiplied, matrix multiplication is performed, otherwise scalar multiplication is used.

matrix is a 20 or 25 element list. If the list is 20 elements long, it is padded with [0, 0, 0, 0, 1] to make a 5x5 matrix, suitable for multiplication.

The following functions produce im.matrix objects:

Function: **im.matrix.identity** ():

Returns an identity im.matrix (one that does not change color or alpha).

Function: **im.matrix.saturation** (level, desat=(0.2126, 0.7152, 0.0722)):

Constructs an im.matrix that alters the saturation of an image. The alpha channel is untouched.

level - The amount of saturation in the resulting image. 1.0 is the unaltered image, while 0.0 is grayscale.

desat - This is a 3-element tuple that controls how much of the red, green, and blue channels will be placed into all three channels of a fully desaturated image. The default is based on the constants used for the luminance channel of an NTSC television signal. Since the human eye is mostly sensitive to green, more of the green channel is kept then the other two channels.

Function: **im.matrix.desaturate** ():

Returns a matrix that desaturates the image (make it grayscale). This is equivalent to calling im.matrix.saturation(0).

Function: **im.matrix.tint** (r, g, b):

Constructs an im.matrix that tints an image, while leaving the alpha channel intact. *r*, *g*, and *b*, should be numbers between 0 and 1, and control what fraction of the given channel is placed into the final image. (For example, if r is .5, and a pixel has a red component of 100, the corresponding pixel will have a red component of 50.)

Function: **im.matrix.invert** ():

Constructs an im.matrix that inverts the red, green, and blue channels of the source image, while leaving the alpha channel alone.

Function: **im.matrix.brightness** (b):

Constructs an im.matrix that alters the brightness of an image, while leaving the alpha channel intact. *b* should be between -1 and 1, with -1 being the darkest possible image and 1 the brightest.

Function: **im.matrix.contrast** (c):

Constructs an im.matrix that alters the contrast of an image. *c* should be greater than 0.0, with values greater than 1.0 increasing contrast and less than 1.0 decreasing it.

Function: **im.matrix.opacity** (o):

Constructs an im.matrix that alters the opacity of an image, while leaving the red, green, and blue channels alone. An *o* of 0.0 is fully transparent, while 1.0 is fully opaque.

Function: **im.matrix.hue** (h):

Returns a matrix that rotates the hue by h degrees, while preserving luminosity.

Matrices constructed with these functions can be composed using matrix multiplication. For example, one can desaturate an image, and then tint it light blue.

```
init:
    image city blue = im.MatrixColor("city.jpg",
im.matrix.desaturate() * im.matrix.tint(0.9, 0.9, 1.0))
```

It's far more efficient to multiply matrices rather than composing im.MatrixColor operations, since the matrix multiplication requires about 125 multiply operations, while the im.MatrixColor uses 16 per pixel times the number of pixels in the image.

There exist two image manipulators that wrap common uses of im.MatrixColor:

Function: **im.Grayscale** (im, desat=(0.2126, 0.7152, 0.0722)):

This image operator converts the given image to grayscale. *desat* is as for im.matrix.saturation.

Function: **im.Sepia** (im, tint=(1.0, .94, .76), desat=(0.2126, 0.7152, 0.0722)):

This image operator sepia-tones an image. *desat* is as for im.matrix.saturation. *tint* is decomposed and used as the parameters to im.matrix.tint.

Backgrounds

There are two displayables that are eminently suitable for use as backgrounds:

Function: **Frame** (image, xborder, yborder, tile=False, bilinear=False):

Returns a displayable that is a frame, based on the supplied image filename. A frame is an image that is automatically rescaled to the size allocated to it. The image has borders that are only scaled in one axis. The region within xborder pixels of the left and right borders is only scaled in the y direction, while the region within yborder pixels of the top and bottom axis is scaled only in the x direction. The corners are not scaled at all, while the center of the image is scaled in both x and y directions.

image - The image (which may be a filename or image object) that will be scaled.

xborder - The number of pixels in the x direction to use as a border.

yborder - The number of pixels in the y direction to use as a border.

tile - If true, then tiling is performed rather then scaling.

bilinear - If true (and tile is false), bilinear scaling is performed instead of nearest-neighbor scaling.

For better performance, have the image share a dimension length in common with the size the frame will be rendered at. We detect this and avoid scaling if possible.

Function: **Solid** (color):

Returns a Displayable that is solid, and filled with a single color. A Solid expands to fill all the space allocated to it, making it suitable for use as a background.

color - The color that the display will be filled with, given either as an RGBA tuple, or an html-style string

Text

Text can be used to show text to the user, as a displayable.

Function: **Text** (text, slow=False, slow_done=None, slow_speed=None, slow_start=0, slow_abortable=False, style='default', **properties):

A displayable that can format and display text on the screen.

text - The text that will be displayed on the screen.

slow - If True, the text will be typed at the screen at a rate determined by the slow_cps property, if set, or the "Text Speed" preference. If None (the default), then it will be typed at a speed determined by the slow_cps property. If False, then it will appear instantly.

style - A style that will be applied to the text.

properties - Additional properties that are applied to the text.

slow_done - A callback that occurs when slow text is done.

slow_speed - The speed of slow text. If none, it's taken from the preferences.

slow_offset - The offset into the text to start the slow text.

slow_abortable - If True, clicking aborts the slow text.

Ren'Py also supports a parameterized text object, which shows text as if it was an image on the screen. But care should be taken, as it's almost always better to use a Character object to show text. By default there is one ParameterizedText image, named `text` declared, but the user can declare more than one to show multiple text blocks on the screen at once.

Function: **ParameterizedText** (style='default', **properties):

This can be used as an image. When used, this image is expected to have a single parameter, a string which is rendered as the image.

Dynamic

DynamicDisplayable can be used in styles to change the displayable shown over the course of the game.

Function: **DynamicDisplayable** (function, *args, **kwargs):

This displayable evaluates a function, and uses the result of that function to determine what to show.

function should be function that should accept at least two arguments. The first argument is the time (in seconds) that the DynamicDisplayable has been shown for. The second argument is the number of seconds that a displayable with the same image tag has been shown for. Additional positional and keyword arguments passed to DynamicDisplayable are also given to the function. The function is expected to return a 2-tuple. The first element of this tuple should be a displayable. The second should be either the time (in seconds) that the return value is valid for, or None to indicate the return value is valid indefinitely.

The function is evaluated at least once for each interaction. It is also evaluated again when the specified time has elapsed.

Note that DynamicDisplayable does not accept properties. Instead, it uses the properties of the displayable returned by *function*.

Note that ConditionSwitch is a wrapper around DynamicDisplayable, and may be simpler depending on your needs.

For compatibility with pre-5.6.3 versions of Ren'Py, DynamicDisplayable also accepts a string as the *function* argument. In this case, the string is evaluated once per interaction, with the result being used as the displayable for that interaction.

Function: **ConditionSwitch** (*args, **properties):

This is a wrapper around DynamicDisplayable that displays the first displayable matching a condition. It takes an even number of positional arguments, with odd arguments being strings containing python conditions, and odd arguments being displayables. On each interaction, it evaluates the conditions in order to find the first that is true, and then displays that displayable. It is an error for no condition to be true.

If supplied, keyword arguments are used to position the chosen displayable.

Function: **ShowingSwitch** (*args, **kwargs):

This chooses a displayable to show based on which images are being shown on the screen. It expects an even number of positional arguments. Odd positional arguments are expected to be image names, while even positional arguments are expected to be displayables. An image matches if it is the prefix of a shown image. If the image name is None, it always matches. It is an error if no match occurs.

This takes the keyword argument *layer*, which specifies the layer that will be checked to see if the images appear on it, defaulting to "master". Other keyword arguments are used to position the chosen displayable.

Shown image names are tracked by the predictive image loading mechanism, and so ShowingSwitch will properly predictively load images.

Since the ShowingSwitch matches on the *prefix* of a shown image, you can "trick" it into thinking a particular image is displayed using null versions of images.

```
init:
    image eileen happy hidden = Null()
    image eileen concerned hidden = Null()
```

Then if Eileen's not visible but you want her side image to show a particular expression when she speaks, you can use code like this:

```
show eileen concerned hidden
e "Are you sure that's a good idea?"
"Oh! I didn't realise Eileen was there."
show eileen concerned
```

Animations

This section has been made somewhat obsolete by the introduction of ATL in Ren'Py 6.10.

Ren'Py provides several kinds of animation displayables.

These animation functions take an *anim_timebase* parameter, that determines which timebase to use. The animation timebase, used when anim_timebase is True, starts at the instant of the first frame from which the tag of the image containing this animation has been shown on the screen. This can be used to switch between two animations, in a way that ensures they are synchronized to the same timebase. The displayable timebase, used when anim_timebase=False, starts at the first frame after the displayable is shown, and can be used to ensure the entire animation is seen, even if an image with the same tag was already on the screen.

The displayable timebase is set to zero for children of a Button, each time the button is focused or unfocused. This means that animations that are children of the button (including backgrounds of the button) that have anim_timebase=False will be restarted when the button changes focus.

The animation functions are:

Function: **Animation** (*args, **properties):

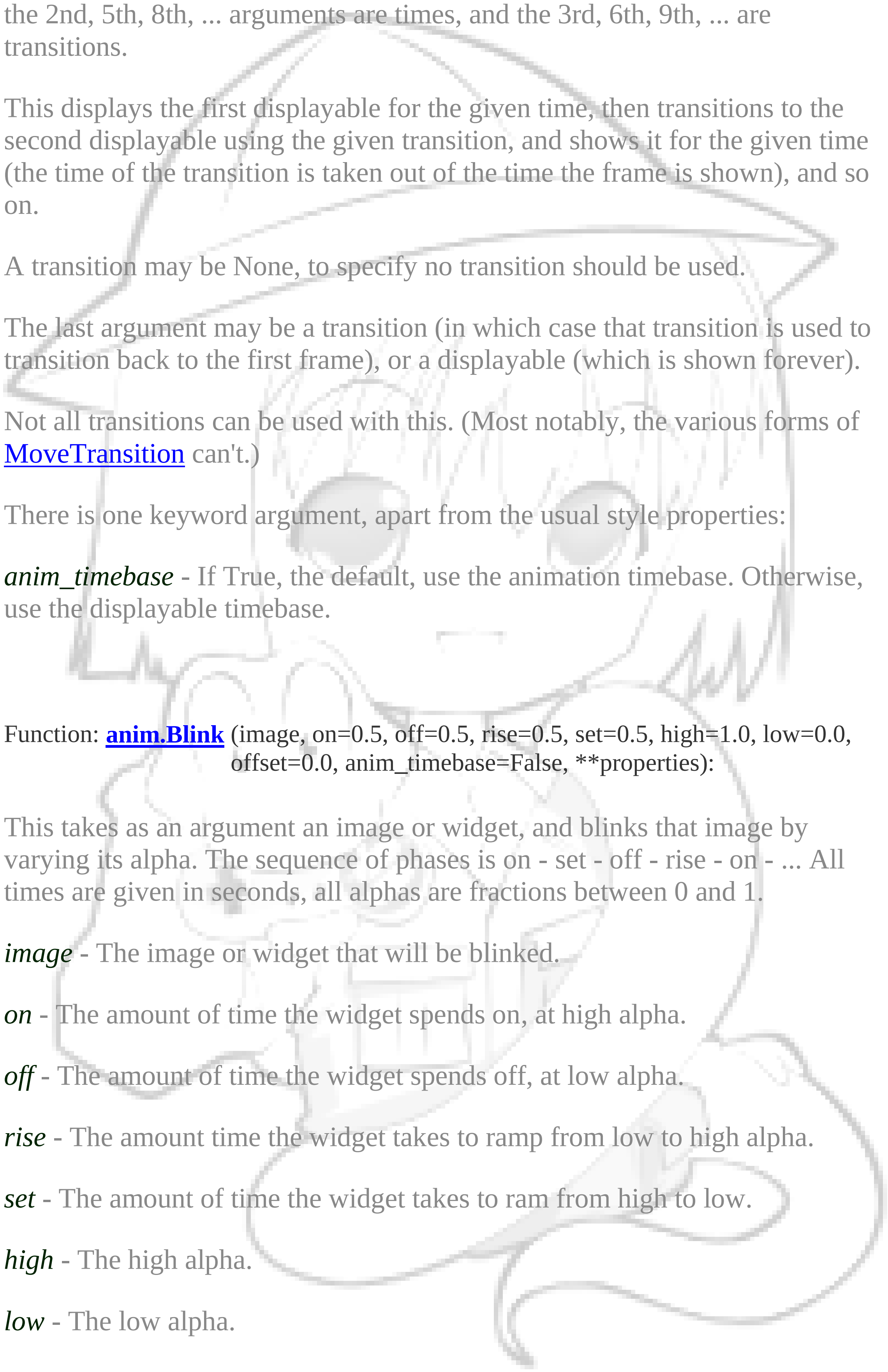
A Displayable that draws an animation, which is a series of images that are displayed with time delays between them.

Odd (first, third, fifth, etc.) arguments to Animation are interpreted as image filenames, while even arguments are the time to delay between each image. If the number of arguments is odd, the animation will stop with the last image (well, actually delay for a year before looping). Otherwise, the animation will restart after the final delay time.

anim_timebase - If True, the default, use the animation timebase. Otherwise, use the displayable timebase.

Function: **anim.TransitionAnimation** (*args, **kwargs):

A displayable that draws an animation with each frame separated by a transition.



This takes arguments such that the 1st, 4th, 7th, ... arguments are displayables, the 2nd, 5th, 8th, ... arguments are times, and the 3rd, 6th, 9th, ... are transitions.

This displays the first displayable for the given time, then transitions to the second displayable using the given transition, and shows it for the given time (the time of the transition is taken out of the time the frame is shown), and so on.

A transition may be None, to specify no transition should be used.

The last argument may be a transition (in which case that transition is used to transition back to the first frame), or a displayable (which is shown forever).

Not all transitions can be used with this. (Most notably, the various forms of MoveTransition can't.)

There is one keyword argument, apart from the usual style properties:

anim_timebase - If True, the default, use the animation timebase. Otherwise, use the displayable timebase.

Function: **anim.Blink** (image, on=0.5, off=0.5, rise=0.5, set=0.5, high=1.0, low=0.0, offset=0.0, anim_timebase=False, **properties):

This takes as an argument an image or widget, and blinks that image by varying its alpha. The sequence of phases is on - set - off - rise - on - ... All times are given in seconds, all alphas are fractions between 0 and 1.

image - The image or widget that will be blinked.

on - The amount of time the widget spends on, at high alpha.

off - The amount of time the widget spends off, at low alpha.

rise - The amount time the widget takes to ramp from low to high alpha.

set - The amount of time the widget takes to ram from high to low.

high - The high alpha.

low - The low alpha.

offset - A time offset, in seconds. Use this to have a blink that does not start at the start of the on phase.

anim_timebase - If True, use the animation timebase, if false, the displayable timebase.

Function: **anim.SMAnimation** (initial, *args, **properties):

This creates a state-machine animation. Such an animation is created by randomly traversing the edges between states in a defined state machine. Each state corresponds to an image shown to the user, with the edges corresponding to the amount of time an image is shown, and the transition it is shown with.

Images are shown, perhaps with a transition, when we are transitioning into a state containing that image.

initial - The name (a string) of the initial state we start in.

showold - If the keyword parameter showold is True, then the old image is shown instead of the new image when in an edge.

anim_timebase - If True, we use the animation timebase. If False, we use the displayable timebase.

This accepts as additional arguments the anim.State and anim.Edge objects that are used to make up this state machine.

Function: **anim.State** (name, image, *atlist, **properties):

This creates a state that can be used in an anim.SMAnimation.

name - A string giving the name of this state.

image - The displayable that is shown to the user while we are in (entering) this state. For convenience, this can also be a string or tuple, which is interpreted with Image.

image should be None when this State is used with motion, to indicate that the image will be replaced with the child of the motion.

atlist - A list of functions to call on the image. (In general, if something can be used in an at clause, it can be used here as well.)

If any keyword arguments are given, they are used to construct a Position object, that modifies the position of the image.

Function: **anim.Edge** (old, delay, new, trans=None, prob=1):

This creates an edge that can be used with a anim.SMAnimation.

old - The name (a string) of the state that this transition is from.

delay - The number of seconds that this transition takes.

new - The name (a string) of the state that this transition is to.

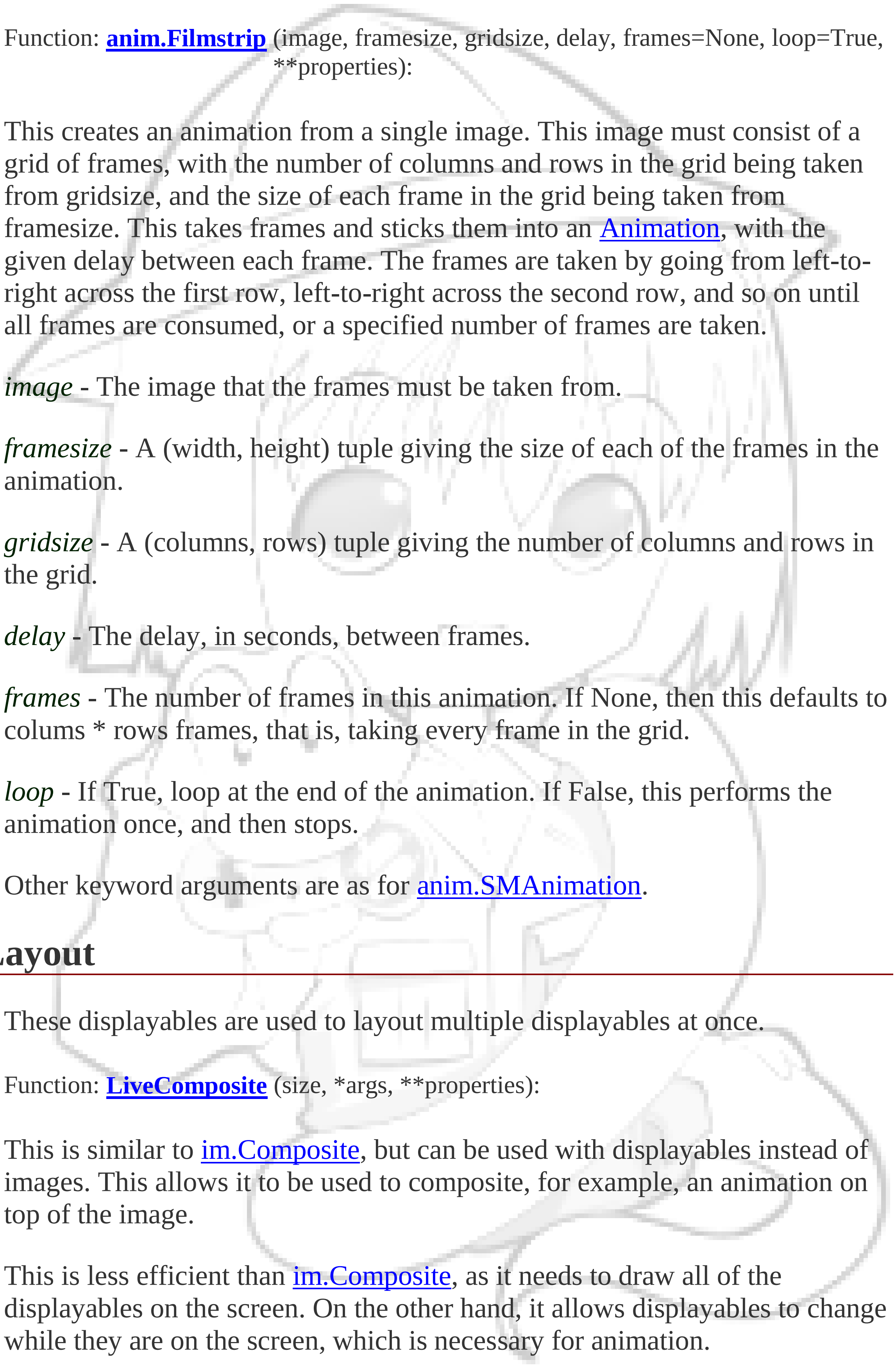
trans - The transition that will be used to show the image found in the new state. If None, the image is show immediately.

prob - The number of times this edge is added. This can be used to make a transition more probable then others. For example, if one transition out of a state has prob=5, and the other has prob=1, then the one with prob=5 will execute 5/6 of the time, while the one with prob=1 will only occur 1/6 of the time. (Don't make this too large, as memory use is proportional to this value.)

We present two examples of this in action. The first shows how one can create a character that ocassionally, randomly, performs a 3-frame blink about once a minute.

```
init:
    image blinking = anim.SMAnimation("a",
        anim.State("a", "eyes_open.png"),

        # This edge keeps us showing the eyes open for a second.
        anim.Edge("a", 1.0, "a", prob=60),

        # This edge causes the eyes to start closing...
        anim.Edge("a", 0.25, "b"),

        # ..because it brings us here.
        anim.State("b", "eyes_half.png"),

        # And so on...
        anim.Edge("b", 0.25, "c"),
        anim.State("c", "eyes_closed.png"),
        anim.Edge("c", 0.25, "d"),
        anim.State("d", "eyes_half.png"),

        # And back to a.
        anim.Edge("d", 0.5, "a")
        )
```

Remember, State can take a Position, and Edge can take a transition. This lets you move things around the screen, dissolve images into others, and do all sorts of complicated, unexpected, things. (But be careful... not all transitions do what you'd expect when used with SMAnimation.)



The anim.Filmstrip function is not deprecated.

Function: **anim.Filmstrip** (image, framesize, gridsize, delay, frames=None, loop=True, **properties):

This creates an animation from a single image. This image must consist of a grid of frames, with the number of columns and rows in the grid being taken from gridsize, and the size of each frame in the grid being taken from framesize. This takes frames and sticks them into an Animation, with the given delay between each frame. The frames are taken by going from left-to-right across the first row, left-to-right across the second row, and so on until all frames are consumed, or a specified number of frames are taken.

image - The image that the frames must be taken from.

framesize - A (width, height) tuple giving the size of each of the frames in the animation.

gridsize - A (columns, rows) tuple giving the number of columns and rows in the grid.

delay - The delay, in seconds, between frames.

frames - The number of frames in this animation. If None, then this defaults to colums * rows frames, that is, taking every frame in the grid.

loop - If True, loop at the end of the animation. If False, this performs the animation once, and then stops.

Other keyword arguments are as for anim.SMAnimation.

Layout

These displayables are used to layout multiple displayables at once.

Function: **LiveComposite** (size, *args, **properties):

This is similar to im.Composite, but can be used with displayables instead of images. This allows it to be used to composite, for example, an animation on top of the image.

This is less efficient than im.Composite, as it needs to draw all of the displayables on the screen. On the other hand, it allows displayables to change while they are on the screen, which is necessary for animation.

This takes a variable number of arguments. The first argument is size, which must be a tuple giving the width and height of the composited widgets, for layout purposes.

It then takes an even number of further arguments. (For an odd number of total arguments.) The second and other even numbered arguments contain position tuples, while the third and further odd-numbered arguments give displayables. A position argument gives the position of the displayable immediately following it, with the position expressed as a tuple giving an offset from the upper-left corner of the LiveComposite. The displayables are drawn in bottom-to-top order, with the last being closest to the user.

Function: **Fixed** (*args, **properties):

A layout that expands to take the size allotted to it. Each displayable is allocated the entire size of the layout, with the first displayable further from the user than the second, and so on.

This function takes both positional and keyword arguments. Positional arguments should be displayables or images to be laid out. Keyword arguments are interpreted as style properties, except for the style keyword argument, which is the name of the parent style of this layout.

Function: **HBox** (*args, **properties):

A layout that lays out displayables from left to right.

This function takes both positional and keyword arguments. Positional arguments should be displayables or images to be laid out. Keyword arguments are interpreted as style properties, except for the style keyword argument, which is the name of the parent style of this layout.

Function: **VBox** (*args, **properties):

A layout that lays out displayables from top to bottom.

This function takes both positional and keyword arguments. Positional arguments should be displayables or images to be laid out. Keyword arguments are interpreted as style properties, except for the style keyword argument, which is the name of the parent style of this layout.

Widget

These are generally used with the ui functions, but may sometimes be used as displayables.

Function: **Bar** (range, value, clicked=None, **properties):

This creates a Bar displayable, the displayable equivalent of a ui.bar. The parameters are the same as for ui.bar.

Function: **Button** (child, clicked=None, hovered=None, unhovered=None, role=, ***properties):*

This creates a button displayable that can be clicked by the user. When this button is clicked or otherwise selected, the function supplied as the clicked argument is called. If it returns a value, that value is returned from ui.interact.

child - The displayable that is contained within this button. This is required.

clicked - A function that is called when this button is clicked.

hovered - A function that is called when this button gains focus.

unhovered - A function that is called when this button loses focus.

role - The role this button undertakes. This can be the empty string, or "selected_".

Function: **Window** (child, **properties):

Creates a Window displayable with the given child and properties.

Function: **Viewport** (child_size=(None, None), xadjustment=None, yadjustment=None, set_adjustments=True, mousewheel=False, draggable=False, style='viewport', **properties):

Creates a viewport displayable. A viewport restricts the size of its child, and allows the child to be displayed at an offset. This can be used for cropping and scrolling the images.

See ui.viewport for parameter meanings.

Function: **Null** (width=0, height=0, **properties):

A displayable that does not display anything. By setting the width and height properties of this displayable, it can be used to take up space.

Particle Motion

Ren'Py supports particle motion. Particle motion is the motion of many particles on the screen at once, with particles having a lifespan that is shorter than a single interaction. Particle motion can be used to have multiple things moving on the screen at once, such as snow, cherry blossoms, bubbles, fireflies, and more. There are two interfaces we've provided for the particle motion engine. The SnowBlossom function is a convenience constructor for the most common cases of linearly falling or rising particles, while the Particles function gives complete control over the particle engine.

SnowBlossom is a function that can be used for the common case of linearly rising or falling particles. Some cases in which it can be used are for falling snow, falling cherry blossoms, and rising bubbles.

Function: **SnowBlossom** (image, count=10, border=50, xspeed=(20, 50), yspeed=(100, 200), start=0, horizontal=False):

This implements the snowblossom effect, which is a simple linear motion up or down, left, or right. This effect can be used for falling cherry blossoms, falling snow, rising bubbles, and drifting dandelions, along with other things.

image - The image that is to be used for the particles. This can actually be any displayable, so it's okay to use an Animation as an argument to this parameter.

count - The number of particles to maintain at any given time. (Realize that not all of the particles may be on the screen at once.)

border - How many pixels off the screen to maintain a particle for. This is used to ensure that a particle is not displayed on the screen when it is created, and that it is completely off the screen when it is destroyed.

xspeed - The horizontal speed of the particles, in pixels per second. This may be a single integer, or it may be a tuple of two integers. In the latter case, the two numbers are used as a range from which to pick the horizontal speed for each particle. The numbers can be positive or negative, as long as the second is larger then the first.

yspeed - The vertical speed of the particles, in pixels per second. This may be a single integer, but it should almost certainly be a pair of integers which are used as a range from which to pick the vertical speed of each particle. (Using a single number will lead to every particle being used in a wave... not what is wanted.) The second number in the tuple should be larger then the first.

start - This is the number of seconds it will take to start all particles moving. Setting this to a non-zero number prevents an initial wave of particles from overwhelming the screen. Each particle will start in a random amount of time less than this number of seconds.

fast - If true, then all particles will be started at once, and they will be started at random places on the screen, rather then on the top or bottom.

horizontal - If true, the particles start on the left or right edge of the screen. If false, they start along the top or bottom edge.

The result of SnowBlossom is best used to define an image, which can then be shown to the user.

```
init:
    image blossoms = SnowBlossom(Animation("sakura1.png", 0.15,
                                           "sakura2.png", 0.15))
```

It may make sense to show multiple snowblossoms at once. For example, in a scene with falling cherry blossoms, one can have small cherry blossoms falling slowly behind a character, while having larger cherry blossoms falling faster in front of her.

If SnowBlossom does not do what you want, it may make sense to define your own particle motion. This is done by calling the Particles function.

Function: **Particles** (factory, style='default', **properties):

Supports particle motion.

factory - A factory object.

The particles function expects to take as an argument a factory object. This object (which should be pickleable) must support two methods.

The create method of the factory object is called once per frame with two arguments. The first is either a list of existing particles, or None if this is the first time this Particles is shown (and hence there are no particles on the screen). The second argument is the time in seconds from some arbitrary point, increasing each time create is called. The method is expected to return a list of new particles created for this frame, or an empty list if no particles are to be created this frame.

The predict method of the factory object is called when image prediction is requested for the Particles. It is expected to return a list of displayables and/or image filenames that will be used.

Particles are represented by the objects returned from each factory function. Each particle object must have an update method. This method is called once per frame per particle, usually with the time from the same arbitrary point as was used to create the object. (The arbitrary point may change when hidden and shown, so particle code should be prepared to deal with this.) The update method may return None to indicate that the particle is dead. Nothing is shown for a dead particle, and update is never called on it. The update method can also return an (xpos, ypos, time, displayable) tuple. The xpos and ypos parameters are a position on the screen to show the particle at, interpreted in the same way as the xpos and ypos style properties. The time is the time of display. This should start with the time parameter, but it may make sense to offset it to make multiple particle animations out of phase. Finally, the displayable is a displayable or image filename that is shown as the particle.

Position and Motion Functions

This section has been made somewhat obsolete by the introduction of ATL in Ren'Py 6.10.

The result of these functions are suitable for use as the argument to the "at" clause of the scene and show statements. The result can also be called (using a second call) to return a displayable.

Function: **Position** (**properties):

Position, when given position properties as arguments, returns a callable that can be passed to the "at" clause of a show or scene statement to display the image at the given location. See the Position Properties section to get a full explanation of how they are used to lay things out.

Function: **Motion** (function, period, repeat=False, bounce=False, time_warp=None, add_sizes=False, anim_timebase=False, **properties):

Motion, when given the appropriate arguments, returns an object that when given as the `at` clause of an image causes an image to be moved on the screen.

function is a function that takes one or two arguments. The first argument is a fraction of the period, a number between 0 and 1. If *add_sizes* is true, *function* should take a second argument, a 4-tuple giving the width and height of the area in which the child will be shown, and the width and height of the child itself.

function should return a tuple containing two or four values. The first two values are interpreted as the xpos and the ypos of the motion. (Please note that if these values are floating point numbers, they are interpreted as a fraction of the screen. If they are integers, they are interpreted as the absolute position of the anchor of the motion.) If four values are returned, the third and fourth values are interpreted as an xanchor and yanchor.

Please note that the function may be pickled, which means that it cannot be an inner function or a lambda, but must be a function defined in an init block of your script. In general, it's better to use a Pan or a Move, rather than defining your own motion.

period is the time, in seconds, it takes to complete one cycle of a motion. If *repeat* is True, then the cycle repeats when it finishes, if False, the motion stops after one period. If *bounce* is True, the argument to the function goes from 0 to 1 to 0 in a single period, if False, it goes from 0 to 1.

time_warp, if given, is a function that takes a fractional time period (a number between 0.0 and 1.0) and returns a fractional time period. This allows non-linear motions. This function may also be pickled.

anim_timebase is true to use the animation timebase, false to use the displayable timebase.

add_sizes was added in 5.6.6.

Function: **Pan** (startpos, endpos, time, repeat=False, bounce=False, time_warp=None, **properties):

Pan, when given the appropriate arguments, gives an object that can be passed to the at clause of an image to cause the image to be panned on the screen. The parameters *startpos* and *endpos* are tuples, containing the x and y coordinates of the upper-left hand corner of the screen relative to the image. *time* is the time it will take this position to move from startpos to endpos. *repeat*, *bounce*, and *time_warp* are as for Motion.

As the current implementation of Ren'Py is quite limited, there are quite a few restrictions that we put on pan. The big one is that there always must be a screen's worth of pixels to the right and below the start and end positions. Failure to ensure this may lead to inconsistent rendering.

Please note that the pan will be immediately displayed, and that Ren'Py will not wait for it to complete before moving on to the next statement. This may lead to the pan being overlayed with text or dialogue. You may want to use a call to renpy.pause to delay for the time it will take to complete the pan.

Finally, also note that when a pan is completed, the image locks into the ending position.

Function: **Move** (startpos, endpos, time, repeat=False, bounce=False, time_warp=None, **properties):

Move is similar to Pan, insofar as it involves moving things. But where Pan moves the screen through an image, Move moves an image on the screen. Specifially, move changes the position style of an image with time.

Move takes as parameters a starting position, an ending position, the amount of time it takes to move from the starting position to the ending position, and extra position properties. The postions may either be pairs giving the xpos and ypos properties, or 4-tuples giving xpos, ypos, xanchor, and yanchor. These properties may be given as integers or floating point numbers, but for any property it's not permissible to mix the two. *repeat*, *bounce*, and *time_warp* are as for Motion.

In general, one wants to use Pan when an image is bigger than the screen, and Move when it is smaller. Both Pan and Move are special cases of Motion.

Function: **SplineMotion** (points, time, anchors=(0.5, 0.5), repeat=False, bounce=False, anim_timebase=False, time_warp=None, **properties):

This creates a spline-based motion, where a spline may consist of linear segments, quadratic beziers, and cubic beziers.

The path is a list of segments, where each segment is a tuple containing between 1 to 3 points, and an optional time. A point is represented by either an (x, y) pair or an (x, y, xanchor, yanchor) tuple. When not specified in a point, the anchors are taken from the *anchors* parameter. The time is represented as a floating point number between 0.0 and 1.0, and gives the time when motion along the segment should be complete.

A linear segment consists of a single point, the point to move to.

A quadratic curve contains two points, the point to move to and the single control point.

A cubic curve contains three points, the point to move to and the two control points.

Any time you don't manually specify is linearly interpolated between the previous and following times that are specified. This allows you to specify that at a specific time, the motion will be at a specific point. By default, the start time is 0.0 and the end time is 1.0, unless you specify something different.

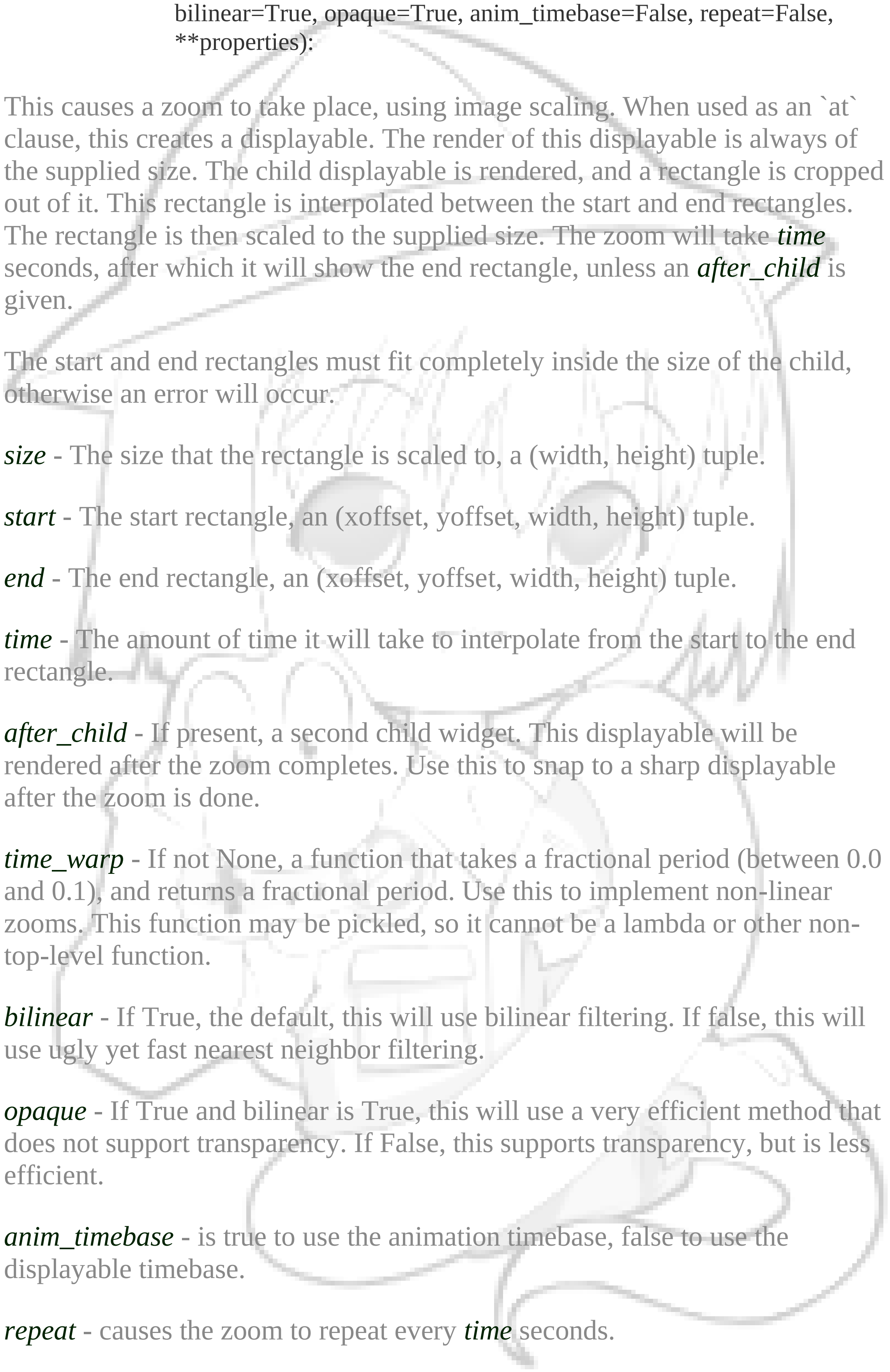
There is a spline editor that helps with editing splines.

Repeat, *bounce*, *anim_timebase*, and *time_warp* are as for Motion.

anim.SMAnimation can also be used to declare complicated motions. Use None instead of an image in States, and supply a move transition when moving between states. A SMAnimation so created can be passed in to the at clause of an image, allowing it to move things around the screen.

These movement clauses can also be used as Transitions, in which case they affect the position of a single layer or the entire screen, as appropriate.

Function: **Zoom** (size, start, end, time, after_child=None, time_warp=None, bilinear=True, opaque=True, anim_timebase=False, repeat=False, **properties):



This causes a zoom to take place, using image scaling. When used as an `at` clause, this creates a displayable. The render of this displayable is always of the supplied size. The child displayable is rendered, and a rectangle is cropped out of it. This rectangle is interpolated between the start and end rectangles. The rectangle is then scaled to the supplied size. The zoom will take *time* seconds, after which it will show the end rectangle, unless an *after_child* is given.

The start and end rectangles must fit completely inside the size of the child, otherwise an error will occur.

size - The size that the rectangle is scaled to, a (width, height) tuple.

start - The start rectangle, an (xoffset, yoffset, width, height) tuple.

end - The end rectangle, an (xoffset, yoffset, width, height) tuple.

time - The amount of time it will take to interpolate from the start to the end rectangle.

after_child - If present, a second child widget. This displayable will be rendered after the zoom completes. Use this to snap to a sharp displayable after the zoom is done.

time_warp - If not None, a function that takes a fractional period (between 0.0 and 0.1), and returns a fractional period. Use this to implement non-linear zooms. This function may be pickled, so it cannot be a lambda or other non-top-level function.

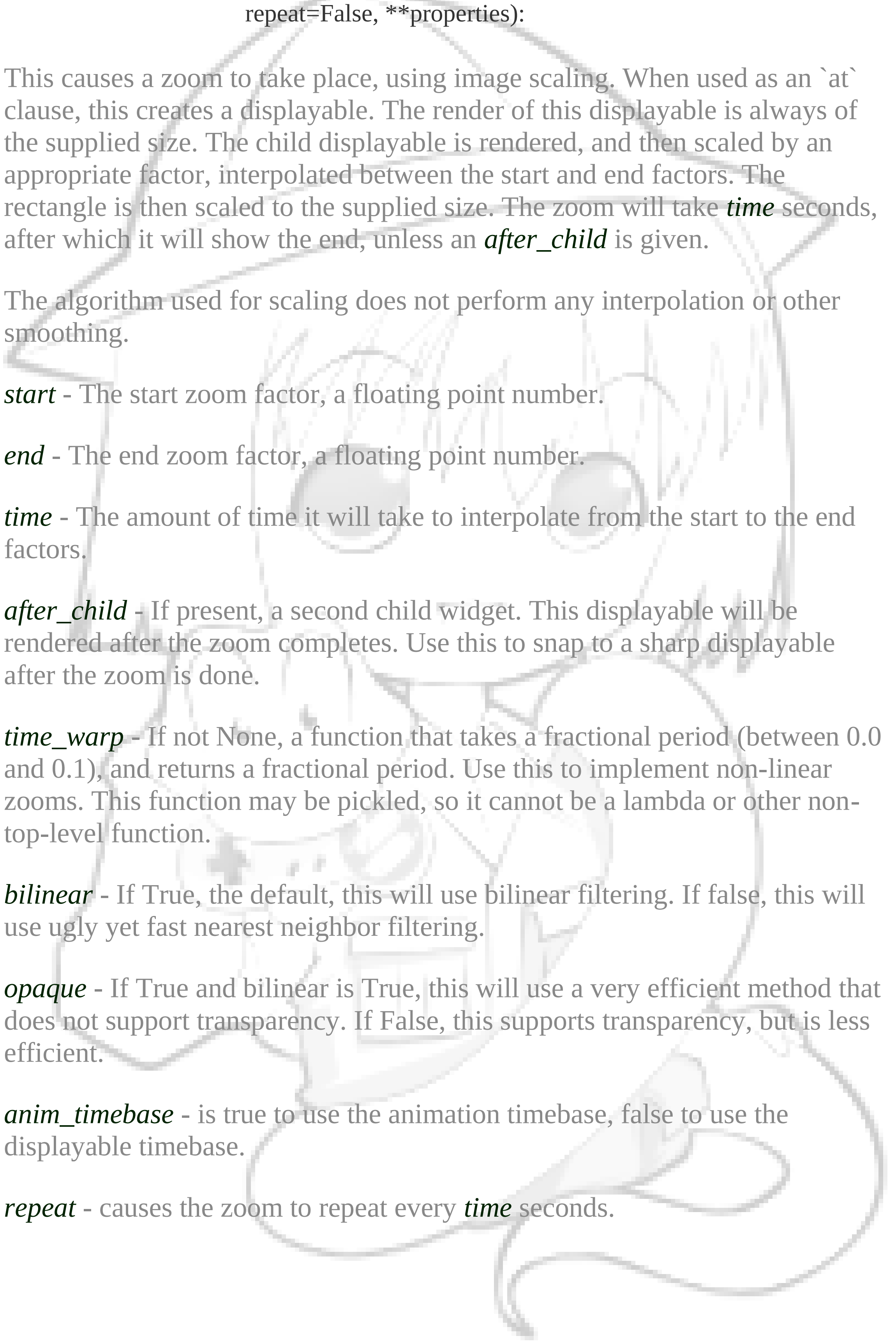
bilinear - If True, the default, this will use bilinear filtering. If false, this will use ugly yet fast nearest neighbor filtering.

opaque - If True and bilinear is True, this will use a very efficient method that does not support transparency. If False, this supports transparency, but is less efficient.

anim_timebase - is true to use the animation timebase, false to use the displayable timebase.

repeat - causes the zoom to repeat every *time* seconds.

Function: **FactorZoom** (start, end, time, after_child=None, time_warp=None, bilinear=True, opaque=True, anim_timebase=False, repeat=False, **properties):



This causes a zoom to take place, using image scaling. When used as an `at` clause, this creates a displayable. The render of this displayable is always of the supplied size. The child displayable is rendered, and then scaled by an appropriate factor, interpolated between the start and end factors. The rectangle is then scaled to the supplied size. The zoom will take *time* seconds, after which it will show the end, unless an *after_child* is given.

The algorithm used for scaling does not perform any interpolation or other smoothing.

start - The start zoom factor, a floating point number.

end - The end zoom factor, a floating point number.

time - The amount of time it will take to interpolate from the start to the end factors.

after_child - If present, a second child widget. This displayable will be rendered after the zoom completes. Use this to snap to a sharp displayable after the zoom is done.

time_warp - If not None, a function that takes a fractional period (between 0.0 and 0.1), and returns a fractional period. Use this to implement non-linear zooms. This function may be pickled, so it cannot be a lambda or other non-top-level function.

bilinear - If True, the default, this will use bilinear filtering. If false, this will use ugly yet fast nearest neighbor filtering.

opaque - If True and bilinear is True, this will use a very efficient method that does not support transparency. If False, this supports transparency, but is less efficient.

anim_timebase - is true to use the animation timebase, false to use the displayable timebase.

repeat - causes the zoom to repeat every *time* seconds.

Function: **RotoZoom** (rot_start, rot_end, rot_delay, zoom_start, zoom_end, zoom_delay, rot_repeat=False, zoom_repeat=False, rot_bounce=False, zoom_bounce=False, rot_anim_timebase=False, zoom_anim_timebase=False, rot_time_warp=None, zoom_time_warp=None, opaque=False, **properties):

This function returns an object, suitable for use in an at cause, that rotates and/or zooms its child.

rot_start - The number of degrees of clockwise rotation at the start time.

rot_end - The number of degrees of clockwise rotation at the end time.

rot_delay - The time it takes to complete rotation.

zoom_start - The zoom factor at the start time.

zoom_end - The zoom factor at the end time.

zoom_delay - The time it takes to complete a zoom.

rot_repeat - If true, the rotation cycle repeats once it is finished.

zoom_repeat - If true, the zoom cycle repeats once it is finished.

rot_bounce - If true, rotate from start to end to start each cycle. If False, rotate from start to end once.

zoom_bounce - If true, zoom from start to end to start each cycle. If False, zoom from start to end once.

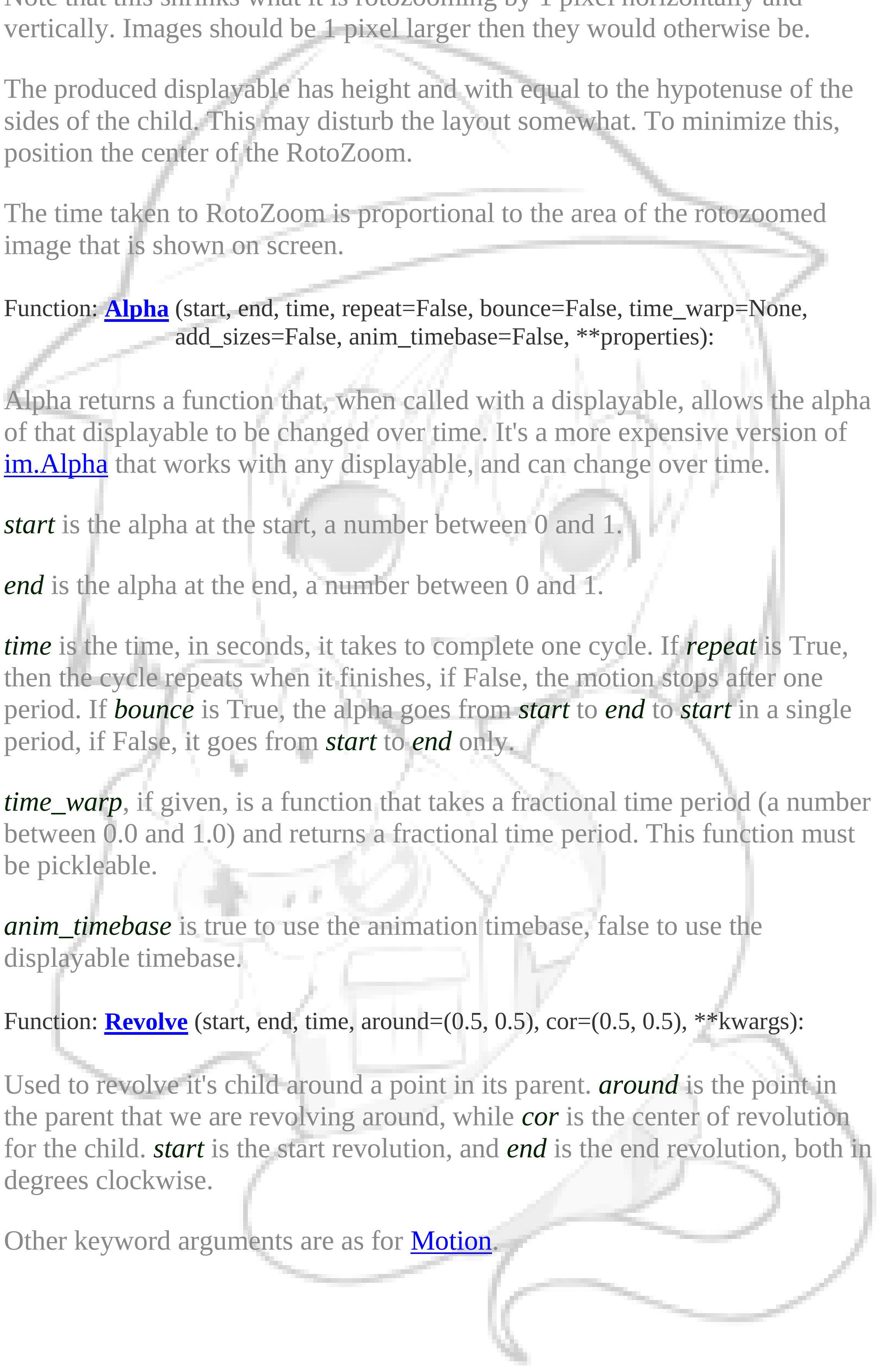
rot_anim_timebase - If true, rotation times are judged from when a displayable with the same tag was first shown. If false, times are judged from when this displayable was first shown.

zoom_anim_timebase - If true, zoom times are judged from when a displayable with the same tag was first shown. If false, times are judged from when this displayable was first shown.

opaque - This should be True if the child is fully opaque, and False otherwise.

RotoZoom uses a bilinear interpolation method that is accurate when the zoom factor is >= 0.5.

Rotation is around the center of the child displayable.



Note that this shrinks what it is rotozooming by 1 pixel horizontally and vertically. Images should be 1 pixel larger then they would otherwise be.

The produced displayable has height and with equal to the hypotenuse of the sides of the child. This may disturb the layout somewhat. To minimize this, position the center of the RotoZoom.

The time taken to RotoZoom is proportional to the area of the rotozoomed image that is shown on screen.

Function: **Alpha** (start, end, time, repeat=False, bounce=False, time_warp=None, add_sizes=False, anim_timebase=False, **properties):

Alpha returns a function that, when called with a displayable, allows the alpha of that displayable to be changed over time. It's a more expensive version of im.Alpha that works with any displayable, and can change over time.

start is the alpha at the start, a number between 0 and 1.

end is the alpha at the end, a number between 0 and 1.

time is the time, in seconds, it takes to complete one cycle. If *repeat* is True, then the cycle repeats when it finishes, if False, the motion stops after one period. If *bounce* is True, the alpha goes from *start* to *end* to *start* in a single period, if False, it goes from *start* to *end* only.

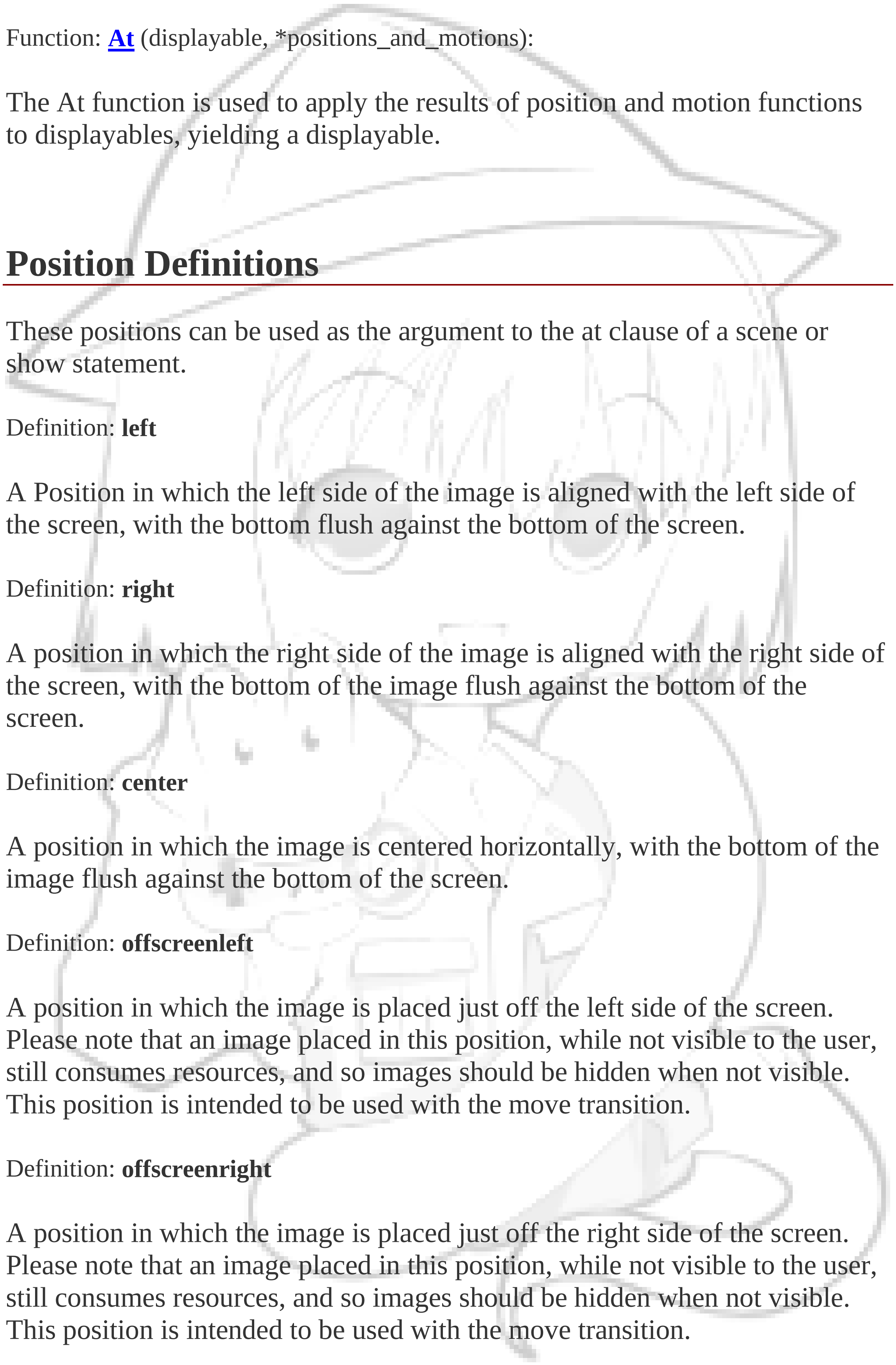
time_warp, if given, is a function that takes a fractional time period (a number between 0.0 and 1.0) and returns a fractional time period. This function must be pickleable.

anim_timebase is true to use the animation timebase, false to use the displayable timebase.

Function: **Revolve** (start, end, time, around=(0.5, 0.5), cor=(0.5, 0.5), **kwargs):

Used to revolve it's child around a point in its parent. *around* is the point in the parent that we are revolving around, while *cor* is the center of revolution for the child. *start* is the start revolution, and *end* is the end revolution, both in degrees clockwise.

Other keyword arguments are as for Motion.



To apply the results of these functions to a displayable, use the At function.

Function: **At** (displayable, *positions_and_motions):

The At function is used to apply the results of position and motion functions to displayables, yielding a displayable.

Position Definitions

These positions can be used as the argument to the at clause of a scene or show statement.

Definition: **left**

A Position in which the left side of the image is aligned with the left side of the screen, with the bottom flush against the bottom of the screen.

Definition: **right**

A position in which the right side of the image is aligned with the right side of the screen, with the bottom of the image flush against the bottom of the screen.

Definition: **center**

A position in which the image is centered horizontally, with the bottom of the image flush against the bottom of the screen.

Definition: **offscreenleft**

A position in which the image is placed just off the left side of the screen. Please note that an image placed in this position, while not visible to the user, still consumes resources, and so images should be hidden when not visible. This position is intended to be used with the move transition.

Definition: **offscreenright**

A position in which the image is placed just off the right side of the screen. Please note that an image placed in this position, while not visible to the user, still consumes resources, and so images should be hidden when not visible. This position is intended to be used with the move transition.

Transform

A Transform allows one to use a Python function to control many aspects of its child. While more complicated, Transform is powerful enough to implement all of the position and motion functions. A Transform may be used as a displayable (by supplying a child) or a position and motion function (omit the child).

Function: **Transform** (child=None, function=None, **properties):

A transform applies rotation, zooming, and alpha-blending to its child, in that order. These operations, along with positioning of the transformed object, are controlled by fields on the Transform object. Transform objects can be composed with minimal overhead.

Parameters. A transform takes these parameters:

child is the child of the Transform. This can be left None, in which case the Transform will act as a Position and Motion function.

function is a callback function that is called before the Transform is rendered, with the Transform object, the shown timebase, and the animation timebase. It can set any of the fields described below, to cause the Transform to alter how it is displayed. It is expected to return a floating-point number giving the amount of time before the Transform should be re-rendered, in seconds. If 0 is returned, the Transform will be re-rendered on the next frame.

Fields. The ATL transform properties are available as fields on a Transform object.

Additional fields are:

- `hide_request` - This is set to true when the Transform is hidden.
- `hide_response` - When `hide_request` is True, this can be set to False to prevent the Transform from being immediately hidden.

Methods. A transform has the following method:

Method: **update** ():

The update method should be called on a transform after one or more fields have been changed outside of the callback function of that Transform.

Method: **set_child** (d):

Sets the child of this Transform to *d*.

Transitions

By default, Ren'Py displays each scene by replacing the old scene with a new one. This is appropriate in general (such as for emotion changes), but it may be boring for large changes, such as a change in location or a character entering or leaving the scene. Ren'Py supports transitions that control how changes to the scene lists are exposed to the user.

Transitions occur between the last scene that was shown to the user, and the current scene that has been updated using the scene, show, or hide statements. A transition runs for a given amount of time, but may be dismissed early by the user. Once a transition is shown, the scene is considered shown for the purposes of future transitions.

Transitions are introduced with the `with` statement. The `with` statement takes an expression that is suitable for use with the `with` statement (that is, a callable that takes as input two scene lists), and runs that transition. Alternatively, if the expression is `None`, then the `with` statement has the effect of showing the scene to the user, and returning instantly. This is useful in conjunction with a future `with` statement, so that only some changes to the scene list will be transitioned in.

An example is in order. First, let us define a few objects that can be passed in as the argument to a with statement:

```
init:
    # Fades to black, then to the new scene.
    fade = Fade(0.5, 0, 0.5)

    # Dissolves between old and new scenes.
    dissolve = Dissolve(0.5)
```

A simple use of with would be to place it after a series of show and hide statements of the program. As an example:

```
scene bg whitehouse
show eileen happy
with fade
```

This series of statements will cause the old scene (displayed before these statements started) to fade to black, and be replaced with the new scene all at once. This is a useful behavior, for example, when we are replacing a scene with a new one, such as when the story changes locations.

```
scene bg whitehouse
with None
show eileen happy
```

```
with dissolve
```

The `with None` statement is useful to break changes to the scene list into parts, as in the example above. When run, the background will be instantly shown, and then the character image will be dissolved in over the background.

Another use of the `with None` statement is to remove transient elements before a transition begins. By default, the scene list includes transient elements like dialogue, thoughts, and menus. `with None` always executes without these elements, and so gets rid of them.

The `show`, `hide`, and `scene` statements all take a with clause. One of these statement with a with clause associated with it is actually converted into three statements: A `with None` statement, the original statement sans the with clause, and the with clause as a with statement. For example:

```
scene bg whitehouse with fade
show eileen happy at left with dissolve
show lucy happy at right with dissolve
```

becomes

```
with None
scene bg whitehouse
with fade
with None
show eileen happy at left
with dissolve
with None
show lucy happy at right
with dissolve
```

This has the effect of fading out the old scene and fading in the new background, then dissolving in the characters one after the other.

```
show albert at far_left
show bernardette at left
show charles at center
show derek at right
show eleanor at far_right
with dissolve
```

Additionally, you can also show multiple images at once by ordering them consecutively and putting the 'with' statement at the end, followed by your transition.

We also allow with clauses to be supplied for say and menu statements. When a with clause is supplied on one of these statements, the transition is used to introduce the say or menu element. For example,

```
e "How are you doing?" with dissolve
```

Will dissolve in a line of dialogue. The line of dialogue will be dismissed immediately, unless it is followed by a with statement or clause that causes it to transition to something else.

There are two variables that control transitions:

Variable: **default_transition** = None

If not none, specifies a default transition that is applied to all say and menu statements that are not provided a with clause. This is only considered if the transitions preference is set to "All".

Variable: **_window_during_transitions** = False

If set to true, this will show an empty line of narration during transitions, provided you are in the outermost context, and there is nothing transient on the screen.

This only works for the default config.with_callback.

This variable may be changed outside of an init block, but a "with None" statement should be run after doing so for change to take effect.

This is more-or-less obsoleted by the window statement.

Pre-Defined Transitions

Definition: **fade**

An instance of the Fade transition that takes 0.5 seconds to fade to black, and then 0.5 seconds to fade to the new screen.

Definition: **dissolve**

An instance of the Dissolve transition that takes 0.5 seconds to complete.

Definition: **pixellate**

An instance of the Pixellate transition, which takes 1 second to complete, and creates pixels as big as 32x32 over the course of 5 steps in either direction.

Definition: **move**

An instance of the MoveTransition transition, this takes 0.5 seconds to move images that changed position to their new locations.

Definition: **moveinright, moveinleft, moveintop, moveinbottom**

These move entering images onto the screen from the appropriate side, taking 0.5 seconds to do so.

Definition: **moveoutright, moveoutleft, moveouttop, moveoutbottom**

These move leaving images off the screen via the appropriate side, taking 0.5 seconds to do so.

Definition: **ease, easeinright, easeinleft, easeintop, easeinbottom, easeoutright, easeoutleft, easeouttop, easeoutbottom**

These are similar to the move- family of transitions, except that they use a cosine-based curve to slow down the start and end of the transition.

Definition: **zoomin**

This zooms in entering images, taking 0.5 seconds to do so.

Definition: **zoomout**

This zooms out leaving images, taking 0.5 seconds to do so.

Definition: **zoominout**

This zooms in entering images and zooms out leaving images, taking 0.5 seconds to do so.

Definition: **vpunch**

When invoked, this transition shakes the screen vertically for a quarter second.

Definition: **hpunch**

When invoked, this transition shakes the screen horizontally for a quarter second.

Definition: **blinds**

Transitions the screen in a vertical blinds effect lasting 1 second.

Definition: **squares**

Transitions the screen in a squares effect lasting 1 second.

Definition: **wiperight**

An instance of CropMove that takes 1 second to wipe the screen right.

Definition: **wipeleft**

An instance of CropMove that takes 1 second to wipe the screen left.

Definition: **wipeup**

An instance of CropMove that takes 1 second to wipe the screen up.

Definition: **wipedown**

An instance of CropMove that takes 1 second to wipe the screen down.

Definition: **slideright**

An instance of CropMove that takes 1 second to slide the screen right.

Definition: **slideleft**

An instance of CropMove that takes 1 second to slide the screen left.

Definition: **slideup**

An instance of CropMove that takes 1 second to slide the screen up.

Definition: **slidedown**

An instance of CropMove that takes 1 second to slide the screen down.

Definition: **slideawayright**

An instance of CropMove that takes 1 second to slide the screen away and to the right.

Definition: **slideawayleft**

An instance of CropMove that takes 1 second to slide the screen away and to the left.

Definition: **slideawayup**

An instance of CropMove that takes 1 second to slide the screen away and to the up.

Definition: **slideawaydown**

An instance of CropMove that takes 1 second to slide the screen away and to the down.

Definition: **irisout**

An instance of CropMove that irises the screen out for 1 second.

Definition: **irisin**

An instance of CropMove that irises the screen in for 1 second.

Transition Constructors

The following are functions that return things useful as transitions. The user should not supply the new_widget or old_widget parameters, as these are supplied by Ren'Py when a transition begins.

Function: **CropMove** (time, mode='fromleft', startcrop=(0.0, 0.0, 0.0, #0), startpos=(0.0, 0.0), endcrop=(0.0, 0.0, #0, #0), endpos=(0.0, 0.0), topnew=True, old_widget=None, new_widget=None):

The CropMove transition works by placing the old and the new image on two layers, called the top and the bottom. (Normally the new image is on the top, but that can be changed in some modes.) The bottom layer is always drawn in full. The top image is first cropped to a rectangle, and then that rectangle drawn onto the screen at a specified position. Start and end crop rectangles and positions can be selected by the supplied mode, or specified manually. The result is a surprisingly flexible transition.

This transition has many modes, simplifying its use. We can group these modes into three groups: wipes, slides, and other.

In a wipe, the image stays fixed, and more of it is revealed as the transition progresses. For example, in "wiperight", a wipe from left to right, first the left edge of the image is revealed at the left edge of the screen, then the center of the image, and finally the right side of the image at the right of the screen. Other supported wipes are "wipeleft", "wipedown", and "wipeup".

In a slide, the image moves. So in a "slideright", the right edge of the image starts at the left edge of the screen, and moves to the right as the transition progresses. Other slides are "slideleft", "slidedown", and "slideup".

There are also slideaways, in which the old image moves on top of the new image. Slideaways include "slideawayright", "slideawayleft", "slideawayup", and "slideawaydown".

We also support a rectangular iris in with "irisin" and a rectangular iris out with "irisout". Finally, "custom" lets the user define new transitions, if these ones are not enough.

time - The time that this transition will last for, in seconds.

mode - One of the modes given above.

The following parameters are only respected if the mode is "custom".

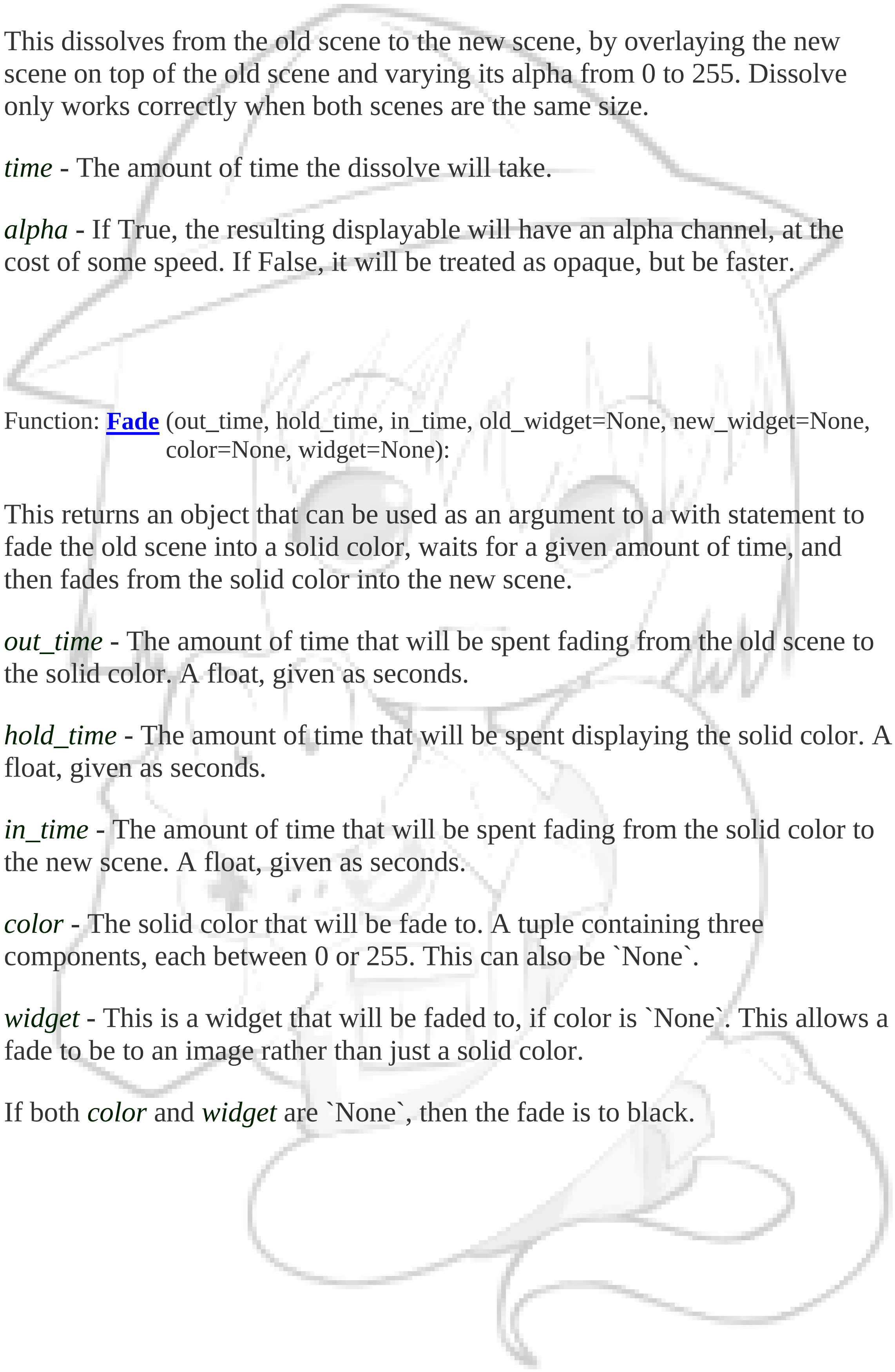
startcrop - The starting rectangle that is cropped out of the top image. A 4-element tuple containing x, y, width, and height.

startpos - The starting place that the top image is drawn to the screen at, a 2-element tuple containing x and y.

startcrop - The starting rectangle that is cropped out of the top image. A 4-element tuple containing x, y, width, and height.

startpos - The starting place that the top image is drawn to the screen at, a 2-element tuple containing x and y.

topnew - If True, the top layer contains the new image. Otherwise, the top layer contains the old image.



Function: **Dissolve** (time, old_widget=None, new_widget=None, alpha=False):

This dissolves from the old scene to the new scene, by overlaying the new scene on top of the old scene and varying its alpha from 0 to 255. Dissolve only works correctly when both scenes are the same size.

time - The amount of time the dissolve will take.

alpha - If True, the resulting displayable will have an alpha channel, at the cost of some speed. If False, it will be treated as opaque, but be faster.

Function: **Fade** (out_time, hold_time, in_time, old_widget=None, new_widget=None, color=None, widget=None):

This returns an object that can be used as an argument to a with statement to fade the old scene into a solid color, waits for a given amount of time, and then fades from the solid color into the new scene.

out_time - The amount of time that will be spent fading from the old scene to the solid color. A float, given as seconds.

hold_time - The amount of time that will be spent displaying the solid color. A float, given as seconds.

in_time - The amount of time that will be spent fading from the solid color to the new scene. A float, given as seconds.

color - The solid color that will be fade to. A tuple containing three components, each between 0 or 255. This can also be `None`.

widget - This is a widget that will be faded to, if color is `None`. This allows a fade to be to an image rather than just a solid color.

If both *color* and *widget* are `None`, then the fade is to black.

Function: **ImageDissolve** (image, time, reverse=False, old_widget=None, new_widget=None):

This dissolves the old scene into the new scene, using an image to control the dissolve process. Basically, this means that white pixels come in first and black last.

The two children should be the same size, or the behavior of ImageDissolve is undefined.

image - The image that will be used to control this transition. The image should be the same size as the scene being dissolved.

time - The amount of time the dissolve will take.

reverse - This reverses the ramp and the direction of the window slide. When True, black pixels dissolve in first, and white pixels come in last.

alpha - If True, the resulting displayable will have an alpha channel, at the cost of some speed. If False, it will be treated as opaque, but be faster.

Function: **MoveTransition** (delay, factory=None, enter_factory=None, leave_factory=None, old=False, layers=['master']):

This transition attempts to find images that have changed position, and moves them from the old position to the new transition, taking *delay* seconds to complete the move.

If *factory* is given, it is expected to be a function that takes as arguments: an old position, a new position, the delay, and a displayable, and to return a displayable as an argument. If not given, the default behavior is to move the displayable from the starting to the ending positions. Positions are always given as (xpos, ypos, xanchor, yanchor) tuples.

If *enter_factory* or *leave_factory* are given, they are expected to be functions that take as arguments a position, a delay, and a displayable, and return a displayable. They are applied to displayables that are entering or leaving the scene, respectively. The default is to show in place displayables that are entering, and not to show those that are leaving.

If *old* is True, then *factory* moves the old displayable with the given tag. Otherwise, it moves the new displayable with that tag. *(new in 6.6.1)*

layers is a list of layers that the transition will be applied to. *(new in 6.9.0)*

Images are considered to be the same if they have the same tag, in the same way that the tag is used to determine which image to replace or to hide. They are also considered to be the same if they are the same displayable.

If you use this transition to slide an image off the side of the screen, remember to hide it when you are done. (Or just use it with a leave_factory.)

There are several constructors that create functions for use with enter_factory and leave_factory:

Function: **MoveFactory** (**kwargs):

Can be used with factory to supply arguments to Move.

Function: **MoveIn** (pos, **kwargs):

Can be used with enter_factory to move in entering elements.

pos - An (xpos, ypos, xanchor, yanchor) tuple, giving the position to move from. Any component can be None, to take the corresponding component of the final position.

Keyword arguments are passed to Move.

Function: **MoveOut** (pos, **kwargs):

Can be used with leave_factory to move in entering elements.

pos - An (xpos, ypos, xanchor, yanchor) tuple, giving the position to move to. Any component can be None, to take the corresponding component of the original position.

Keyword arguments are passed to Move.

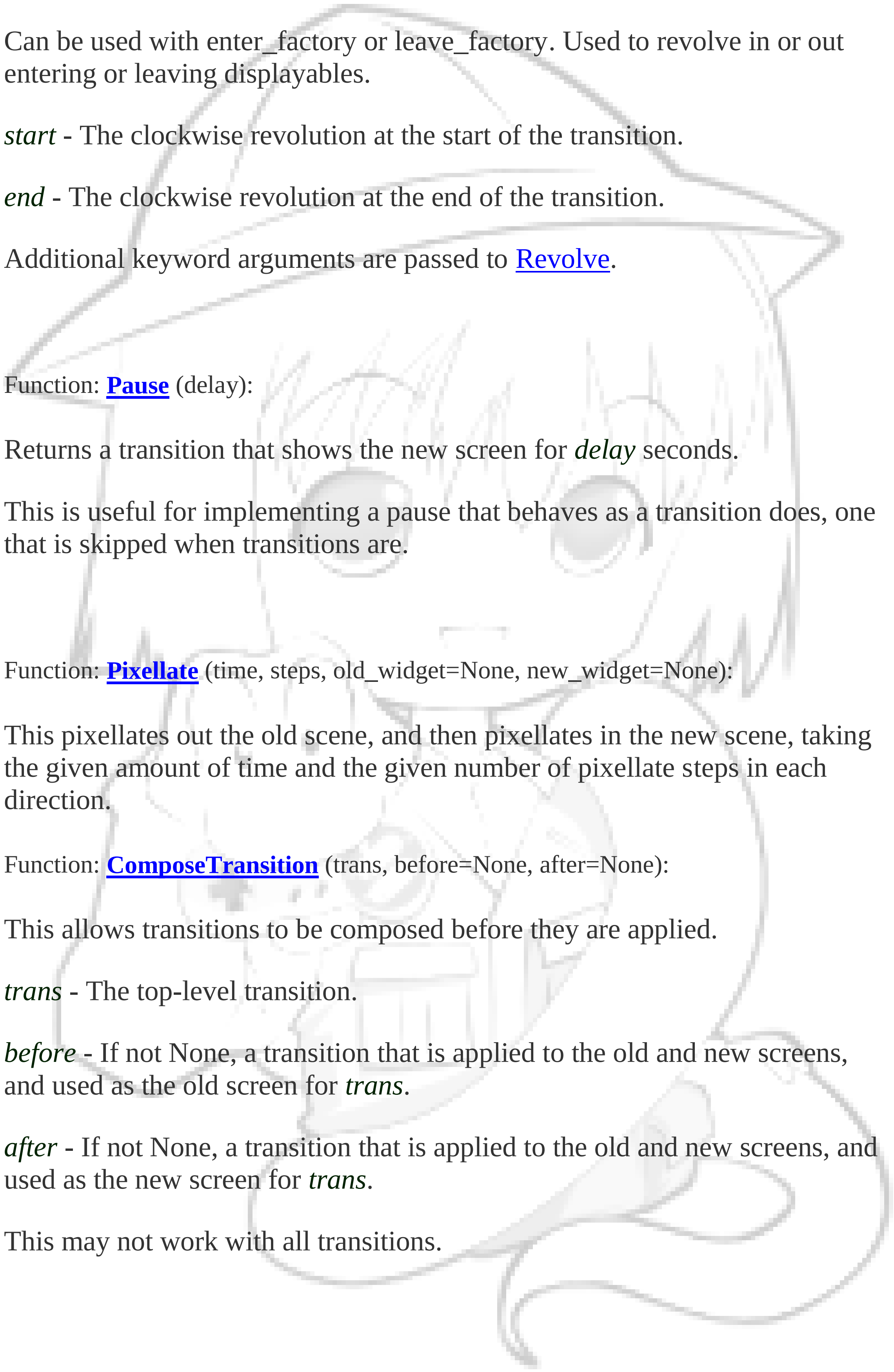
Function: **ZoomInOut** (start, end, **kwargs):

Can be used with enter_factory or leave_factory. Used to zoom in or out entering or leaving displayables.

start - The zoom factor at the start of the transition.

end - The zoom factor at the end of the transition.

Keyword arguments are passed to FactorZoom.



Function: **RevolveInOut** (start, end, **kwargs):

Can be used with enter_factory or leave_factory. Used to revolve in or out entering or leaving displayables.

start - The clockwise revolution at the start of the transition.

end - The clockwise revolution at the end of the transition.

Additional keyword arguments are passed to Revolve.

Function: **Pause** (delay):

Returns a transition that shows the new screen for *delay* seconds.

This is useful for implementing a pause that behaves as a transition does, one that is skipped when transitions are.

Function: **Pixellate** (time, steps, old_widget=None, new_widget=None):

This pixellates out the old scene, and then pixellates in the new scene, taking the given amount of time and the given number of pixellate steps in each direction.

Function: **ComposeTransition** (trans, before=None, after=None):

This allows transitions to be composed before they are applied.

trans - The top-level transition.

before - If not None, a transition that is applied to the old and new screens, and used as the old screen for *trans*.

after - If not None, a transition that is applied to the old and new screens, and used as the new screen for *trans*.

This may not work with all transitions.

Function: **MultipleTransition** (args):

This creates a transition that consists of one or more transitions. *args* must be a list of odd length, containing at least 3 elements. Odd elements of this list are considered to be displayables, while even elements are transitions. When used, this causes the first element (a displayable) to transition to the third element (a displayable), using the second element as the transition. If the fourth and fifth elements are present, a transition will occur from the third element (a displayable) to the fifth element (a displayable) using the fourth element (a transition).

There are two special values that will be recognized as displayables. *False* will be replaced with a displayable containing the scene before the transition, while *True* will be replaced with a displayable containing the scene after the transition.

Some transitions can also be applied to specific layers, using the renpy.transition function. Only transitions that are not completely opaque can be used in this way.

Transition Families

This function defines a family of similar transitions.

Function: **define.move_transitions** (prefix, delay, time_warp=None, in_time_warp=None, out_time_warp=None, layers=['master'], **kwargs):

This defines a family of move transitions, similar to the move and ease transitions. For a given prefix, this defines the transitions:

- prefix - A transition that takes *delay* seconds to move images that changed positions to their new locations.
- prefixinleft, prefixinright, prefixintop, prefixinbottom - A transition that takes *delay* seconds to move images that changed positions to their new locations, with newly shown images coming in from the appropriate side.
- prefixoutleft, prefixoutright, prefixoutop, prefixoutbottom - A transition that takes *delay* seconds to move images that changed

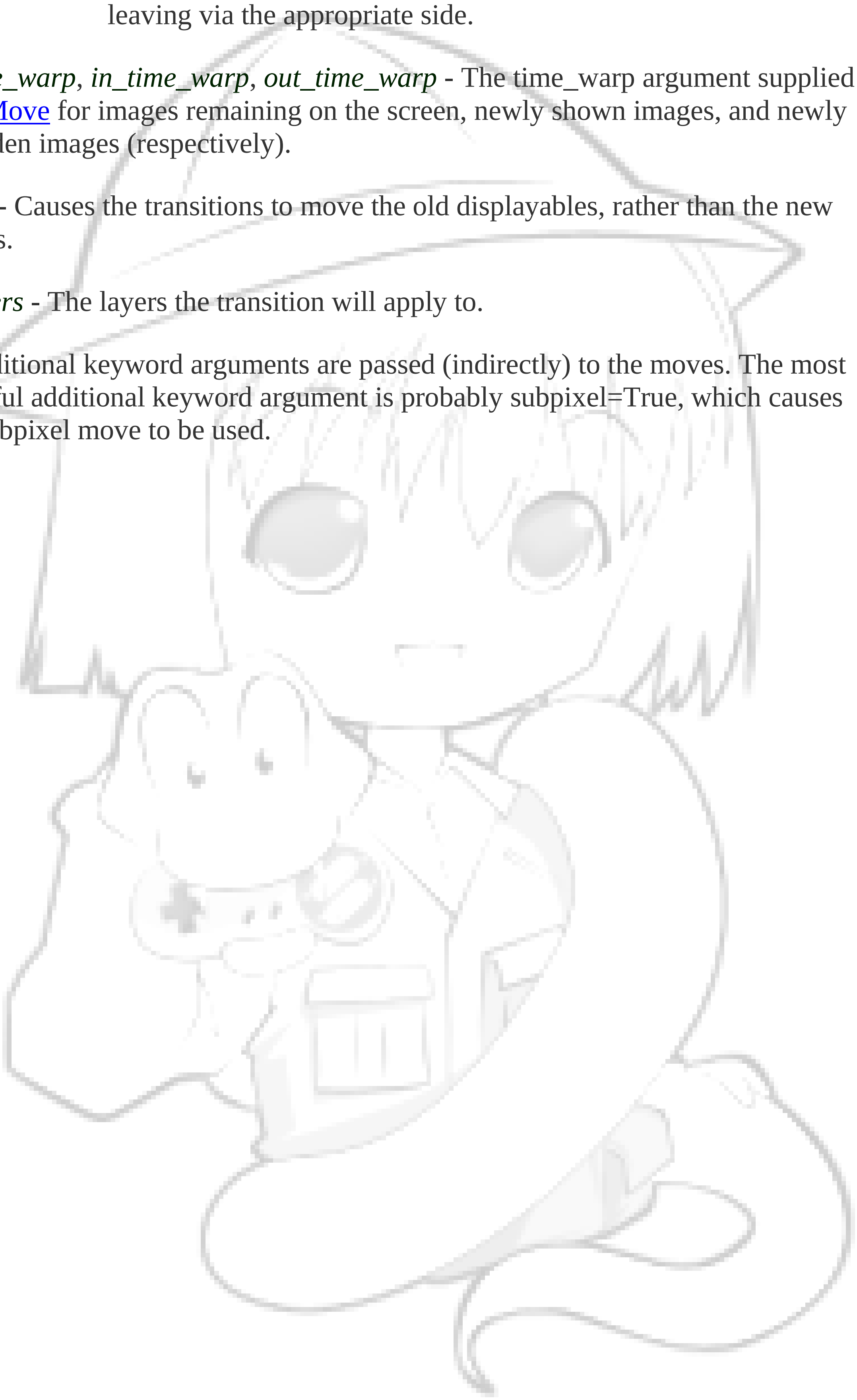
positions to their new locations, with newly hidden images leaving via the appropriate side.

time_warp, *in_time_warp*, *out_time_warp* - The time_warp argument supplied to Move for images remaining on the screen, newly shown images, and newly hidden images (respectively).

old - Causes the transitions to move the old displayables, rather than the new ones.

layers - The layers the transition will apply to.

Additional keyword arguments are passed (indirectly) to the moves. The most useful additional keyword argument is probably subpixel=True, which causes a subpixel move to be used.



Interaction Functions

The Ren'Py interaction functions are available so that you can customize interactions with the user taking advantage of the same code which the normal interactions use.

Statement Helper Functions

The following functions either implement new game behavior that didn't merit its own statement, or complement the behavior of statements.

Function: **renpy.block_rollback** ():

Prevents the game from rolling back to before the current statement.

Function: **renpy.checkpoint** (data=None):

This creates a checkpoint that the user can rollback to. The checkpoint is placed at the statement after the last statement that interacted with the user. Once this function has been called, there should be no more interaction with the user in the current Python block.

data - If not None, this contains data that can be accessed using renpy.roll_forward_info if and when the current statement is re-executed due to the user rolling back and then rolling forward again.

renpy.checkpoint should be called after ui.interact.

Function: **renpy.choice_for_skipping** ():

This is called to indicate to the skipping code that we have reached a choice. If we're skipping, and if the skip after choices preference is not True, then this disables skipping. This should only be needed in a custom interaction function.

Function: **renpy.clear_game_runtime** ():

Resets the game runtime timer down to 0.

The game runtime counter counts the number of seconds that have elapsed while waiting for user input in the current context. (So it doesn't count time spent in the game menu.)

Function: **renpy.current_interact_type** ():

Returns the value of the *type* parameter supplied to ui.interact() during the current interaction. See renpy.last_interact_type for possible interaction types.

Function: **renpy.display_menu** (items, window_style='menu_window', interact=True, with_none=None):

Displays a menu containing the given items, returning the value of the item the user selects.

items - A list of tuples that are the items to be added to this menu. The first element of a tuple is a string that is used for this menuitem. The second element is the value to be returned if this item is selected, or None if this item is a non-selectable caption.

interact - If True, then an interaction occurs. If False, no such interaction occurs, and the user should call ui.interact manually.

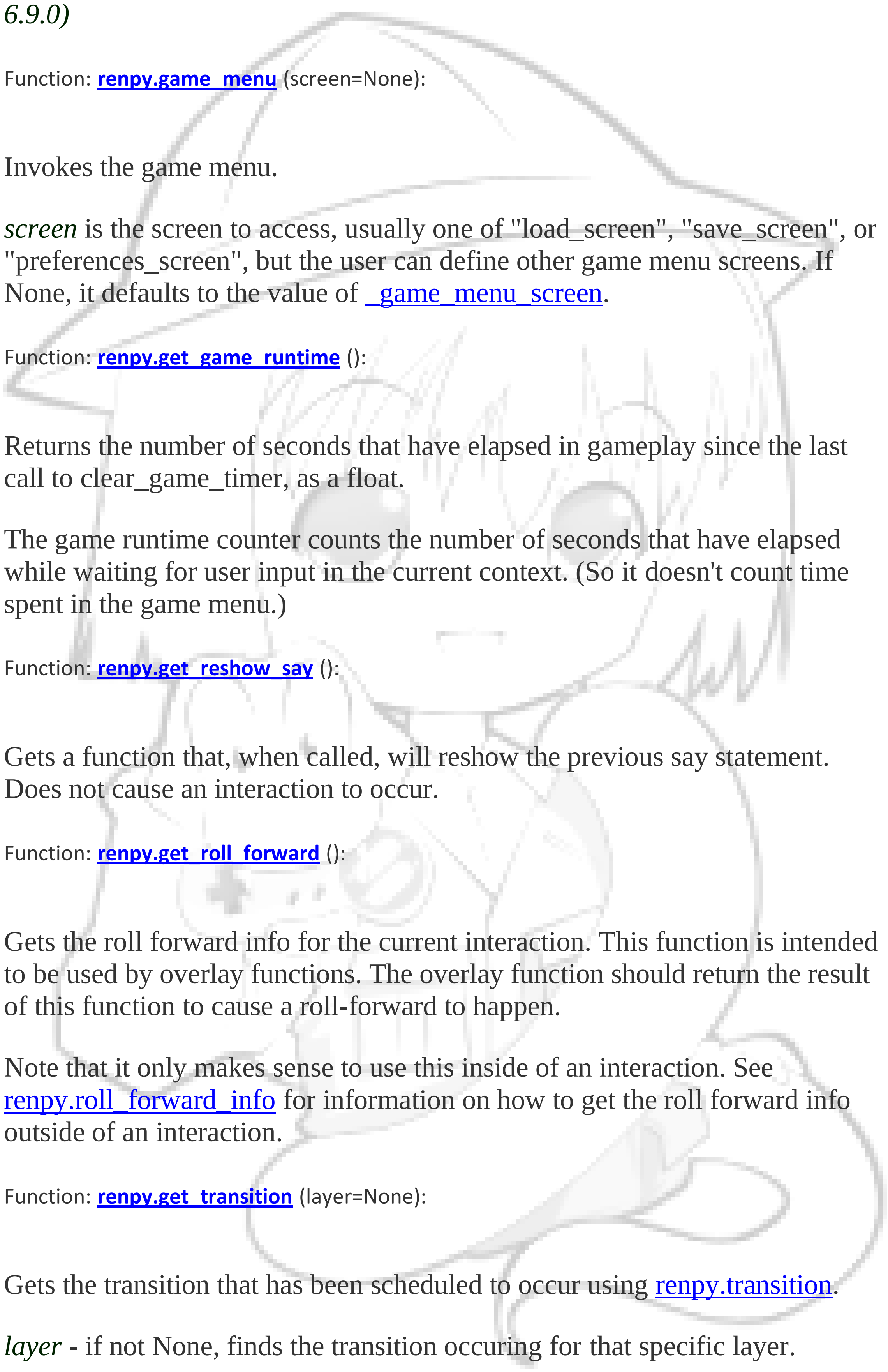
with_none - If True, causes a "with None" statement to be run after each interaction. If None (the default), checks config.implicit_with_none to determine if a "with None" should be run.

Function: **renpy.display_say** (who, what, ...):

Called by Character to implement the say statement.

Function: **renpy.full_restart** (transition=config.end_game_transition, label="_invoke_main_menu", target="_main_menu"):

This causes a full restart of Ren'Py. This resets the state of Ren'Py to what it was when the init blocks finished running, and then restarts the game. After some init code runs, *transition* will be set to run on the next interaction, and control will be transferred to *label*. The default label initializes the main menu context, and then invokes *target*. *target* can be a screen, like "preferences_screen" or "load_screen".



It generally doesn't make sense to specify both *label* and *target*. *(changed in 6.9.0)*

Function: **renpy.game_menu** (screen=None):

Invokes the game menu.

screen is the screen to access, usually one of "load_screen", "save_screen", or "preferences_screen", but the user can define other game menu screens. If None, it defaults to the value of _game_menu_screen.

Function: **renpy.get_game_runtime** ():

Returns the number of seconds that have elapsed in gameplay since the last call to clear_game_timer, as a float.

The game runtime counter counts the number of seconds that have elapsed while waiting for user input in the current context. (So it doesn't count time spent in the game menu.)

Function: **renpy.get_reshow_say** ():

Gets a function that, when called, will reshow the previous say statement. Does not cause an interaction to occur.

Function: **renpy.get_roll_forward** ():

Gets the roll forward info for the current interaction. This function is intended to be used by overlay functions. The overlay function should return the result of this function to cause a roll-forward to happen.

Note that it only makes sense to use this inside of an interaction. See renpy.roll_forward_info for information on how to get the roll forward info outside of an interaction.

Function: **renpy.get_transition** (layer=None):

Gets the transition that has been scheduled to occur using renpy.transition.

layer - if not None, finds the transition occuring for that specific layer.

Function: **renpy.imagemap** (ground, selected, hotspots, unselected=None, overlays=False, style='imagemap', with_none=None, **properties):

Displays an imagemap. An image map consists of two images and a list of hotspots that are defined on that image. When the user clicks on a hotspot, the value associated with that hotspot is returned. Despite the name, this takes arbitrary displayables, not just image manipulators.

ground - The name of the file containing the ground image. The ground image is displayed for areas that are not part of any hotspots.

selected - The name of the file containing the selected image. This image is displayed in hotspots when the mouse is over them.

hotspots - A list of tuples defining the hotspots in this image map. Each tuple has the format (x0, y0, x1, y1, result). (x0, y0) gives the coordinates of the upper-left corner of the hotspot, (x1, y1) gives the lower-right corner, and result gives the value returned from this function if the mouse is clicked in the hotspot.

unselected - If provided, then it is the name of a file containing the image that's used to fill in hotspots that are not selected as part of any image. If not provided, the ground image is used instead.

overlays - If True, overlays are displayed when this imagemap is active. If False, the overlays are suppressed.

with_none - If True, causes a "with None" statement to be run after each interaction. If None (the default), checks config.implicit_with_none to determine if a "with None" should be run.

Function: **renpy.input** (prompt, default=*, allow=None, exclude='{}', length=None, with_none=None):*

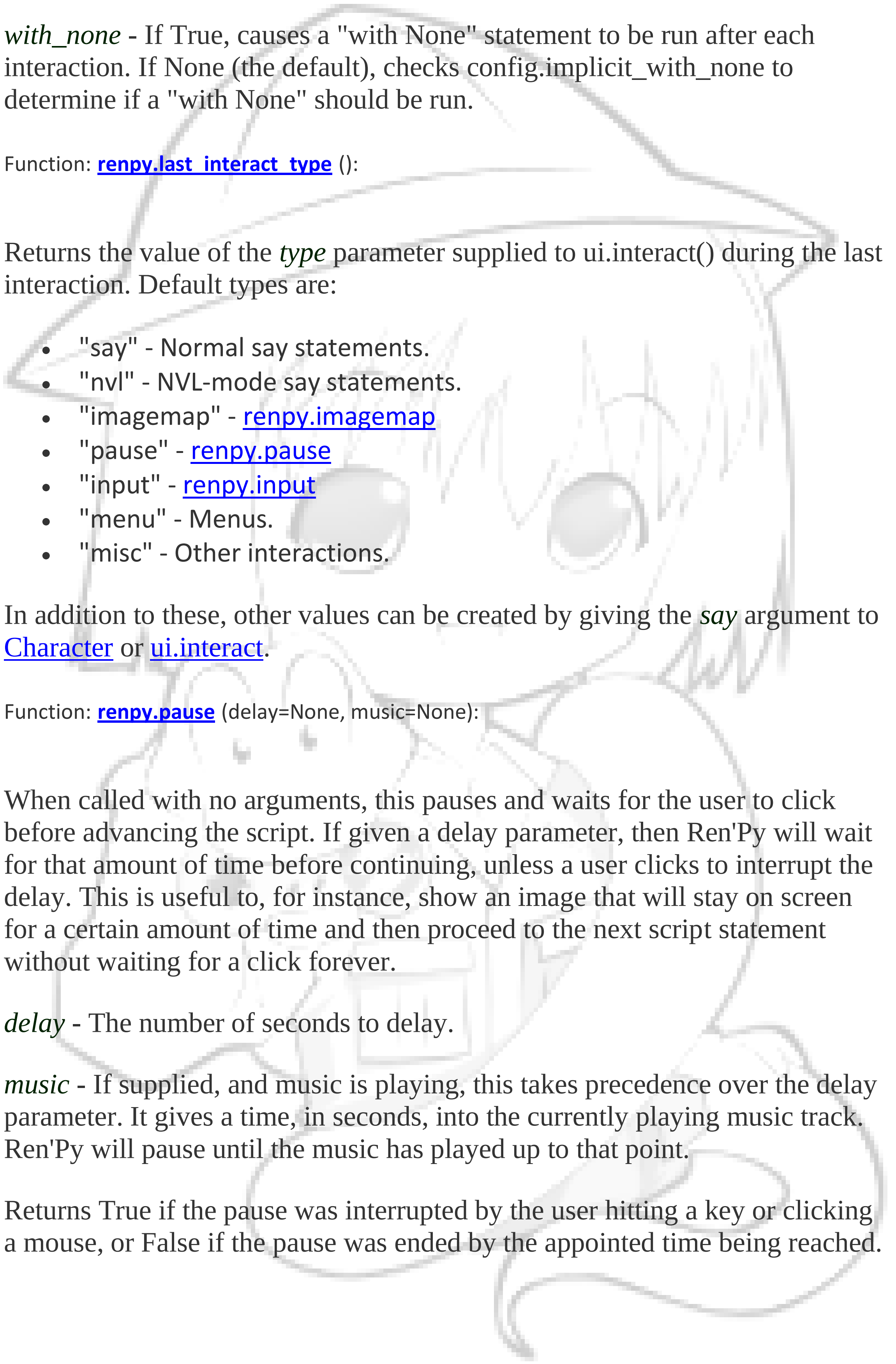
This pops up a window requesting that the user enter in some text. It returns the entered text.

prompt - A prompt that is used to ask the user for the text.

default - A default for the text that this input can return.

length - If given, a limit to the amount of text that this function will return.

allow - If not None, then if an input character is not in this string, it is ignored.



exclude - If not None, then if an input character is in this set, it is ignored.

with_none - If True, causes a "with None" statement to be run after each interaction. If None (the default), checks config.implicit_with_none to determine if a "with None" should be run.

Function: **renpy.last_interact_type** ():

Returns the value of the *type* parameter supplied to ui.interact() during the last interaction. Default types are:

- "say" - Normal say statements.
- "nvl" - NVL-mode say statements.
- "imagemap" - renpy.imagemap
- "pause" - renpy.pause
- "input" - renpy.input
- "menu" - Menus.
- "misc" - Other interactions.

In addition to these, other values can be created by giving the *say* argument to Character or ui.interact.

Function: **renpy.pause** (delay=None, music=None):

When called with no arguments, this pauses and waits for the user to click before advancing the script. If given a delay parameter, then Ren'Py will wait for that amount of time before continuing, unless a user clicks to interrupt the delay. This is useful to, for instance, show an image that will stay on screen for a certain amount of time and then proceed to the next script statement without waiting for a click forever.

delay - The number of seconds to delay.

music - If supplied, and music is playing, this takes precedence over the delay parameter. It gives a time, in seconds, into the currently playing music track. Ren'Py will pause until the music has played up to that point.

Returns True if the pause was interrupted by the user hitting a key or clicking a mouse, or False if the pause was ended by the appointed time being reached.

Function: **renpy.predict** (img):

Forces prediction of the supplied image or displayable. This will cause it to be loaded into the image cache during the next (and only the next) interaction, if there's any free time.

Function: **renpy.predict_display_say** (who, what, ...):

This is the default function used by Character to predict images that will be used by renpy.display_say.

Function: **renpy.quit** ():

This causes Ren'Py to exit entirely.

Function: **renpy.show_display_say** (who, what, who_args={}, what_args={}, window_args={}, image=False, no_ctc_what=..., **kwargs):

This is called (by default) by renpy.display_say to add the widgets corresponding to a screen of dialogue to the user. It is not expected to be called by the user, but instead to be called by renpy.display_say, or by a function passed as the *show_function argument* to Character or renpy.display_say.

who - The name of the character that is speaking, or None to not show this name to the user.

what - What that character is saying. Please note that this may not be a string, as it can also be a list containing both text and displayables, suitable for use as the first argument of ui.text.

who_args - Additional keyword arguments intended to be supplied to the ui.text that creates the who widget of this dialogue.

what_args - Additional keyword arguments intended to be supplied to the ui.text that creates the what widget of this dialogue.

window_args - Additional keyword arguments intended to be supplied to the ui.window that creates the who widget of this dialogue.

image - If True, then who should be interpreted as an image or displayable rather than a text string.

callback - If not None, a function that should be called with no arguments before each time something is shown.

no_ctc_what - The contents of *what* before the click-to-continue indicator was appended.

kwargs - Additional keyword arguments should be ignored.

This function is required to return the ui.text widget displaying the what text.

Function: **renpy.shown_window** ():

This should be called when a window is shown on the screen. Calling this prevents config.empty_window from being called when _window is true. (And hence, prevents the empty window from being shown.)

Function: **renpy.transition** (trans, layer=None, always=False):

Sets the transition that will be used for the next interaction. This is useful when the next interaction doesn't take a with clause, as is the case with interactions in the renpy.pause, renpy.input, and renpy.imagemap functions.

trans - The desired transition.

layer - If the layer setting is not None, then the transition will be applied only to the layer named. Please note that only some transitions can be applied to specific layers.

always - If True, forces the transition to be used, potentially overriding the game preference setting for showing all or no transitions.

Function: **renpy.restart_interaction** ():

Calling this restarts the current interaction, while keeping any ongoing transitions.

This should be called whenever displayables are added or removed over the course of an interaction, or when the information used to construct the overlay changes.

Function: **renpy.reshow_say** ():

Reshows the last say statement. Does not cause an interaction to occur.

Function: **renpy.roll_forward_info** ():

When re-executing a statement after rollback, this returns the data stored using renpy.checkpoint the last time this statement was executed. If this statement is not being re-executed due to a rollback, this returns None.

Often, this data is passed as the roll_forward parameter of ui.interact. It's important to check to ensure that the value is valid before using it in this way.

Context Functions

Contexts store the current scene lists and execution location. Ren'Py supports a stack of contexts, but only the top-level context is saved to the save file.

Function: **renpy.context** ():

Returns an object that is unique to the current context, that participates in rollback and the like.

Function: **renpy.call_in_new_context** (label):

This code creates a new context, and starts executing code from that label in the new context. Rollback is disabled in the new context. (Actually, it will just bring you back to the real context.)

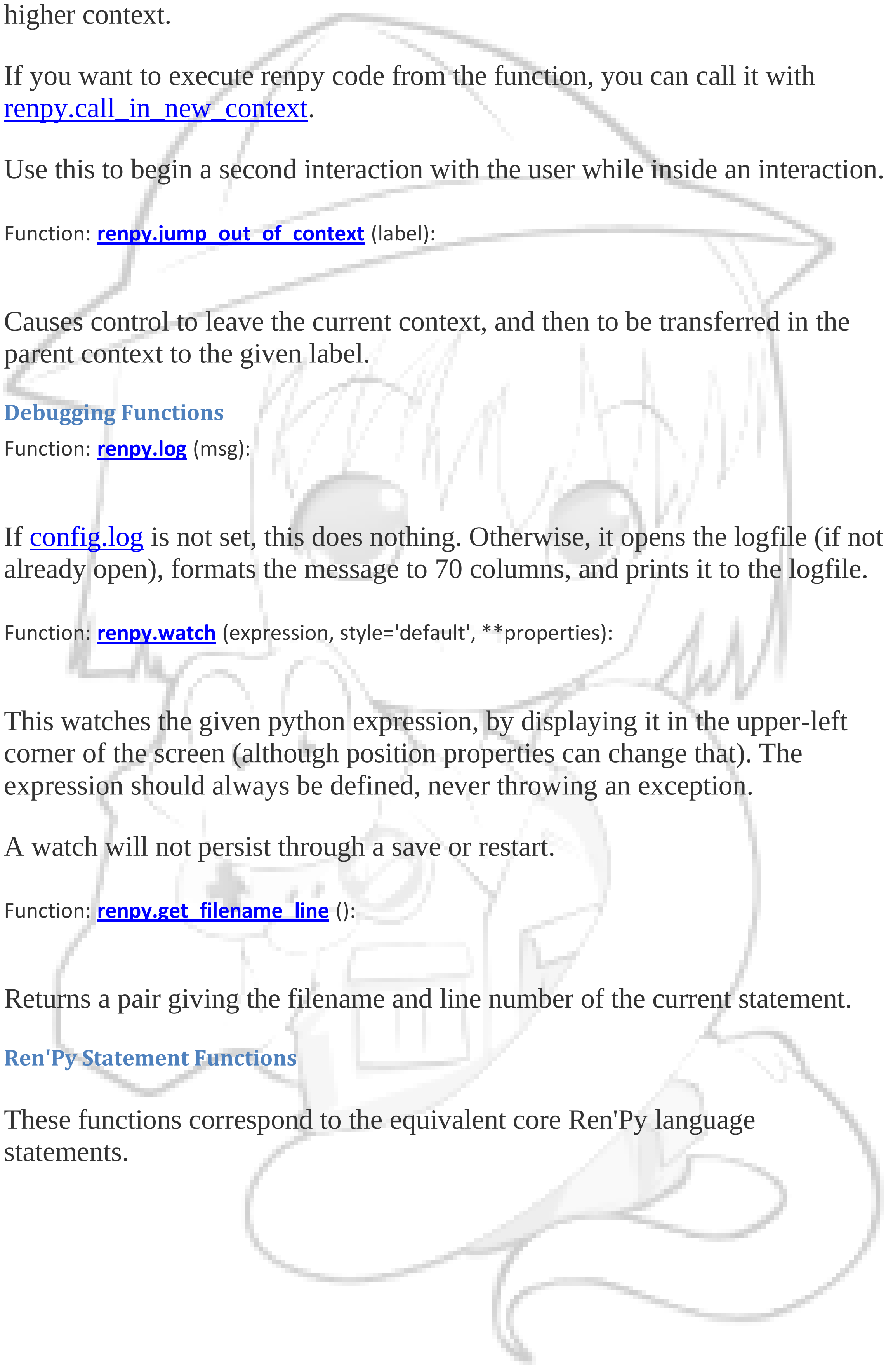
Use this to begin a second interaction with the user while inside an interaction.

Function: **renpy.invoke_in_new_context** (callable, *args, **kwargs):

This pushes the current context, and invokes the given python function in a new context. When that function returns or raises an exception, it removes the new context, and restores the current context.

Additional arguments and keyword arguments are passed to the callable.

Please note that the context so created cannot execute renpy code. So exceptions that change the flow of renpy code (like the one created by



renpy.jump) cause this context to terminate, and are handled by the next higher context.

If you want to execute renpy code from the function, you can call it with renpy.call_in_new_context.

Use this to begin a second interaction with the user while inside an interaction.

Function: **renpy.jump_out_of_context** (label):

Causes control to leave the current context, and then to be transferred in the parent context to the given label.

Debugging Functions

Function: **renpy.log** (msg):

If config.log is not set, this does nothing. Otherwise, it opens the logfile (if not already open), formats the message to 70 columns, and prints it to the logfile.

Function: **renpy.watch** (expression, style='default', **properties):

This watches the given python expression, by displaying it in the upper-left corner of the screen (although position properties can change that). The expression should always be defined, never throwing an exception.

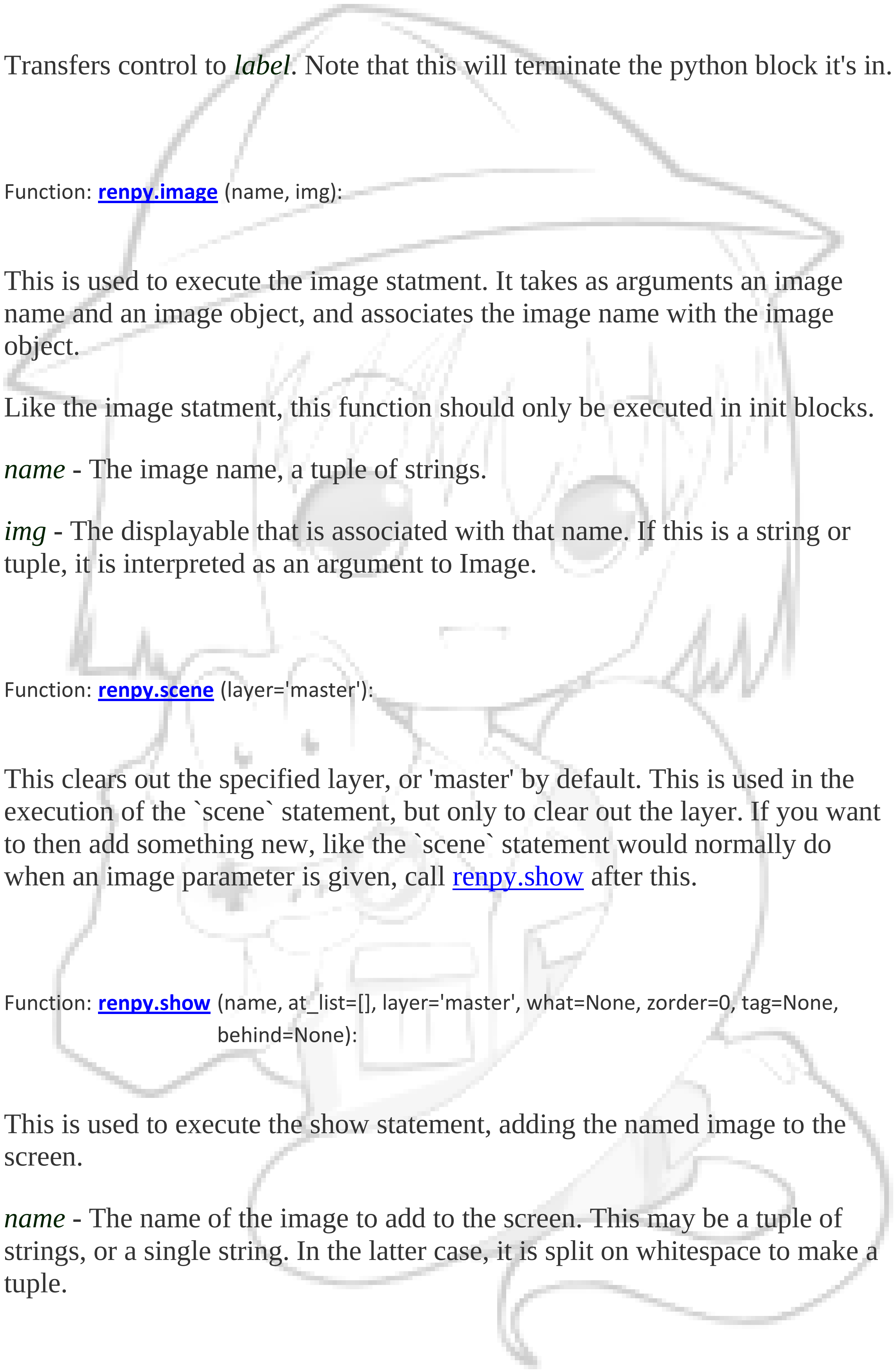
A watch will not persist through a save or restart.

Function: **renpy.get_filename_line** ():

Returns a pair giving the filename and line number of the current statement.

Ren'Py Statement Functions

These functions correspond to the equivalent core Ren'Py language statements.



Function: **renpy.jump** (label):

Transfers control to *label*. Note that this will terminate the python block it's in.

Function: **renpy.image** (name, img):

This is used to execute the image statment. It takes as arguments an image name and an image object, and associates the image name with the image object.

Like the image statment, this function should only be executed in init blocks.

name - The image name, a tuple of strings.

img - The displayable that is associated with that name. If this is a string or tuple, it is interpreted as an argument to Image.

Function: **renpy.scene** (layer='master'):

This clears out the specified layer, or 'master' by default. This is used in the execution of the `scene` statement, but only to clear out the layer. If you want to then add something new, like the `scene` statement would normally do when an image parameter is given, call renpy.show after this.

Function: **renpy.show** (name, at_list=[], layer='master', what=None, zorder=0, tag=None, behind=None):

This is used to execute the show statement, adding the named image to the screen.

name - The name of the image to add to the screen. This may be a tuple of strings, or a single string. In the latter case, it is split on whitespace to make a tuple.

at_list - The at list, a list of functions that are applied to the image when shown. The members of the at list need to be pickleable if sticky_positions is True.

layer - The layer the image will be shown on.

what - If not None, this is expected to be a displayable. That displayable is shown, instead of going through the normal lookup process.

zorder - The zorder of this image.

tag - The tag of this image. If None, the tag is taken from name.

behind - A list of tags this image will be shown behind, if they are present on the same layer at the same zorder, and an image with the same tag is not already present on the layer.

Function: **renpy.hide** (name, layer='master'):

This finds items in the given layer (or 'master' if no layer is given) that have the same name as the first component of the given name, and removes them from the layer. This is used to execute the hide statement.

name - The name of the image to hide from the screen. This may be a tuple of strings, or a single string. In the latter case, it is split on whitespace to make a tuple. Only the first element of the tuple is used.

layer - The layer this operates on.

Function: **renpy.with_statement** (trans, paired=None, always=False, clear=True):

The programmatic equivalent of a with statement.

trans - The transition caused by the with statement.

paired - The paired transition that is passed into config.with_callback.

always - Causes this transition to always occur, even if the user has transitions turned off.

clear - If True, the transient layer will be cleared at the end of the transition.

Scene List Access

These functions let you query the scene lists.

Function: **renpy.showing** (name, layer='master'):

Returns True if *name* is a prefix of an image that's being shown on *layer*. *name* may be a tuple of strings, or a single string. In the latter case, it is split on whitespace to make a tuple.

When called during image prediction, this uses the images that are predicted to be shown, rather than the images that are currently being shown.

Function: **renpy.get_at_list** (name, layer='master'):

This gets the at_list that was used to show a given image tag. *name* is parsed to get out the image tag, and the at_list corresponding to that tag is retrieved. If no image with that tag has been shown on the given layer, then None is returned.

Function: **renpy.layer_at_list** (at_list, layer='master'):

This sets the layer_at_list, an at_list that is applied to the given layer as a whole. This can be used to do things like moving the layer around, or zooming it in and out.

Note that this interacts poorly with MoveTransition.

Miscellaneous Utility Functions

Function: **color** (s):

This function converts hexcode string *s* into a color/alpha tuple. Leading # marks are ignored. Colors can be rgb or rgba, with each element having either one or two digits. (So the strings can be 3, 4, 6, or 8 digits long, not including the optional #.) A missing alpha is interpreted as 255, fully opaque.

For example, color('#123a') returns (17, 34, 51, 170), while color('c0c0c0') returns (192, 192, 192, 255).

Note that as of Ren'Py 5.6.0, functions requiring a color accept the hex string form of specification directly, with no need to use this color function.

Function: **renpy.cache_pin** (*image_manipulators):

When this is called with one or more image manipulators, it causes those image manipulators to be "pinned" into the image cache. This means that they will be loaded soon after the game begins, and will never be removed from the cache.

The usual use of this function is to preload images used in the game menu. Be careful with it, as it can increase Ren'Py's memory usage.

Function: **renpy.cache_unpin** (*image_manipulators):

This may be called with one or more image manipulators. It causes those manipulators to become unpinned and removed from the image cache.

Function: **renpy.curry** (func):

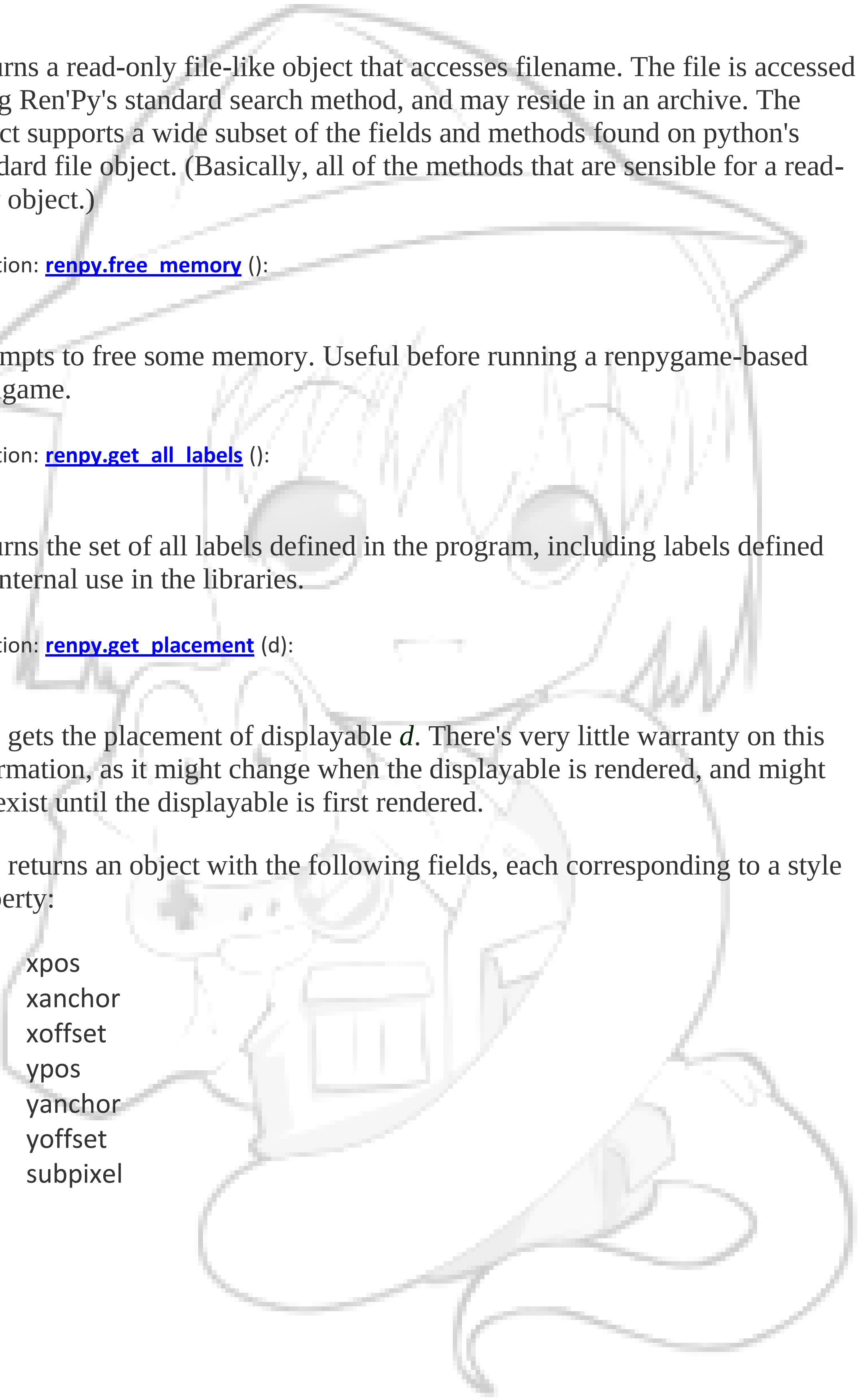
Currying is an operation that allows a function to be called using a chain of calls, each providing some of the arguments. In Ren'Py, calling renpy.curry(**func**) returns a callable object. When this object is called with arguments, a second callable object is returned that stores **func** and the supplied arguments. When this second callable object is called, the original function is called with the stored arguments and the arguments supplied to the second call.

Positional arguments from the first call are placed before positional arguments from the second call. If a keyword argument is given in both calls, the value from the second call takes priority.

Curry objects can be pickled provided the original function remains available at its original name.

Function: **renpy.exists** (filename):

Returns true if the given filename can be found in the searchpath. This only works if a physical file exists on disk. It won't find the file if it's inside of an archive.



Function: **renpy.file** (filename):

Returns a read-only file-like object that accesses filename. The file is accessed using Ren'Py's standard search method, and may reside in an archive. The object supports a wide subset of the fields and methods found on python's standard file object. (Basically, all of the methods that are sensible for a read-only object.)

Function: **renpy.free_memory** ():

Attempts to free some memory. Useful before running a renpygame-based minigame.

Function: **renpy.get_all_labels** ():

Returns the set of all labels defined in the program, including labels defined for internal use in the libraries.

Function: **renpy.get_placement** (d):

This gets the placement of displayable *d*. There's very little warranty on this information, as it might change when the displayable is rendered, and might not exist until the displayable is first rendered.

This returns an object with the following fields, each corresponding to a style property:

- xpos
- xanchor
- xoffset
- ypos
- yanchor
- yoffset
- subpixel

Function: **renpy.loadable** (filename):

Returns True if the given filename is loadable, meaning that it can be loaded from the disk or from inside an archive. Returns False if this is not the case.

Function: **renpy.load_module** (name):

This loads the Ren'Py module named *name*. A Ren'Py module consists of Ren'Py code that is loaded into the usual (store) namespace, contained in a file named *name*.rpym or *name*.rpymc. If a .rpym file exists, and is newer than the corresponding .rpymc file, it is loaded and a new .rpymc file is created.

All init code in the module is run before this function returns. An error is raised if the module *name* cannot be found, or is ambiguous.

Module loading may only occur from inside an init block.

Function: **renpy.seen_audio** (filename):

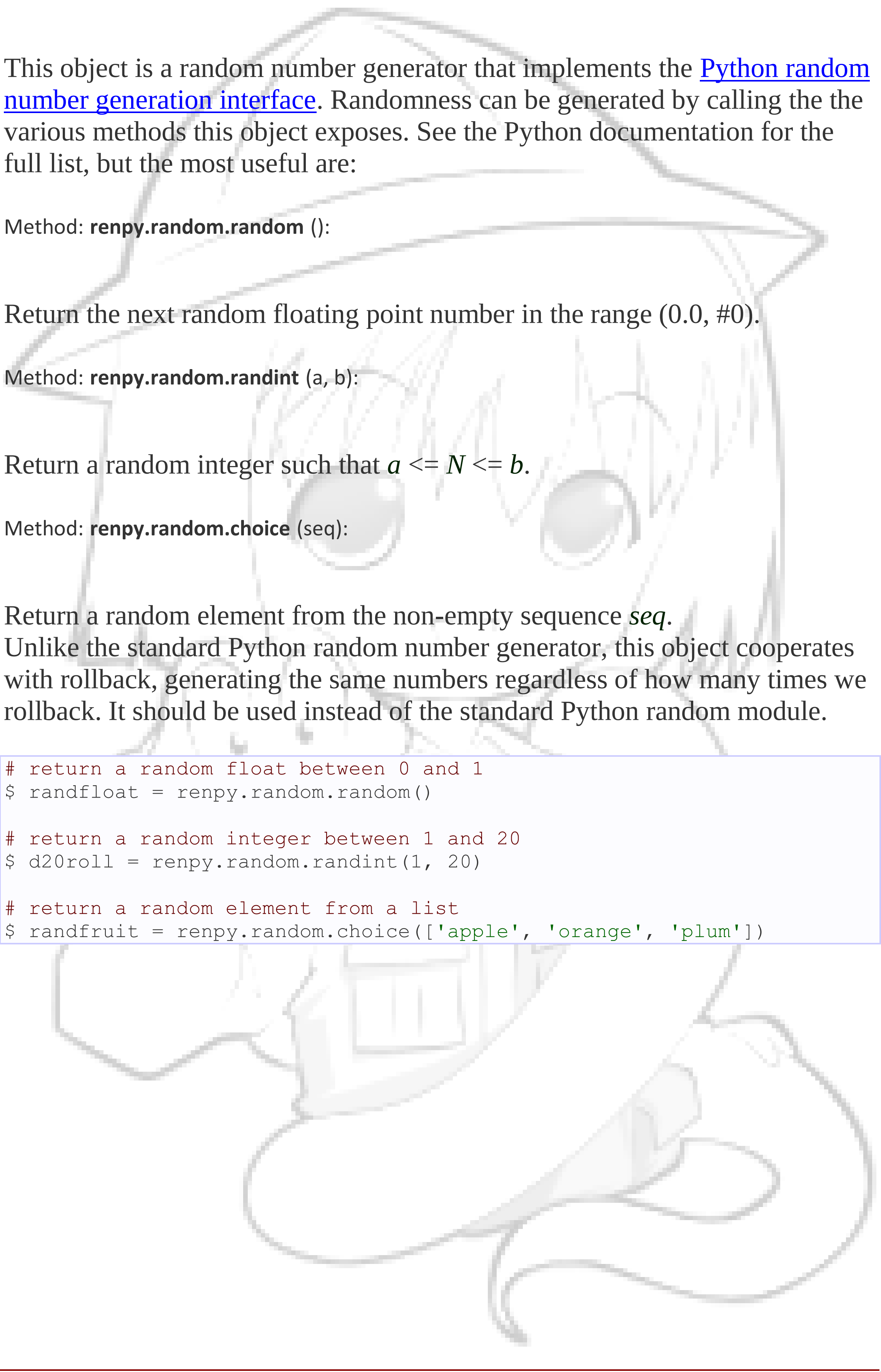
Returns True if the given filename has been played at least once on the current user's system.

Function: **renpy.seen_image** (name):

Returns True if the named image has been seen at least once on the user's system. An image has been seen if it's been displayed using the show statement, scene statement, or renpy.show function. (Note that there are cases where the user won't actually see the image, like a show immediately followed by a hide.)

Function: **renpy.seen_label** (label):

Returns true if the named label has executed at least once on the current user's system, and false otherwise. This can be used to unlock scene galleries, for example.



Variable: **renpy.random** = ...

This object is a random number generator that implements the Python random number generation interface. Randomness can be generated by calling the the various methods this object exposes. See the Python documentation for the full list, but the most useful are:

Method: **renpy.random.random** ():

Return the next random floating point number in the range (0.0, #0).

Method: **renpy.random.randint** (a, b):

Return a random integer such that *a* <= *N* <= *b*.

Method: **renpy.random.choice** (seq):

Return a random element from the non-empty sequence *seq*.
Unlike the standard Python random number generator, this object cooperates with rollback, generating the same numbers regardless of how many times we rollback. It should be used instead of the standard Python random module.

```
# return a random float between 0 and 1
$ randfloat = renpy.random.random()

# return a random integer between 1 and 20
$ d20roll = renpy.random.randint(1, 20)

# return a random element from a list
$ randfruit = renpy.random.choice(['apple', 'orange', 'plum'])
```

Audio

Ren'Py supports playing music and sound effects in the background, using the following audio file formats:

- OGG Vorbis
- MP3
- WAV (uncompressed PCM only)

Ren'Py supports an arbitrary number of audio channels. Three are defined by default:

- music - A channel for music playback.
- sound - A channel for sound effects.
- voice - A channel for voice.

The 'Music Volume', 'Sound Volume', and 'Voice Volume' settings of the in-game preferences menu are used to set individual volumes for these channels.

Sounds can also be set to play when buttons, menu choices, or imagemaps enter their hovered or activated states. See Sound Properties.

Audio Statements

The usual way to play music and sound in Ren'Py is using the three music/sound statements:

- The Play Statement.

play *channelname audiofile(s)* [*fadein*] [*fadeout*]

is used to play sound and music. If a file is currently playing, it is interrupted and replaced with the new file.

channelname is expected to be the name of a channel. (Usually, this is either "sound", "music", or "voice".)

audiofile(s) can be one file or list of files.

fadein and *fadeout* clauses are all optional. Fadeout gives the fadeout time for currently playing music, in seconds, while fadein gives the time it takes to fade in the new music.

```
play music "mozart.ogg"
play sound "woof.mp3"
play myChannel "punch.wav" # 'myChannel' needs to be defined with
renpy.music.register_channel().

"We can also play a list of sounds, or music."
play music [ "a.ogg", "b.ogg" ] fadeout 1.0 fadein 1.0
```

- The Stop Statement.

```
stop sound
stop music fadeout 1.0
```

- The Queue Statement.

```
queue sound "woof.ogg"
queue music [ "a.ogg", "b.ogg" ]
```

The advantage of using these statements is that your program will be checked for missing sound and music files when lint is run. The functions below exist to allow access to allow music and sound to be controlled from python, and to expose advanced (rarely-used) features.

Two configuration variables, config.main_menu_music and config.game_menu_music allow for the given music files to be played as the main and game menu music, respectively.

Defining Channels

It's possible to define your own channel by calling renpy.music.register_channel in init code.

Function: **renpy.music.register_channel** (channel, mixer, loop, tight=False, file_prefix="", file_suffix=""):

This registers a new audio channel named *channel*.

mixer - The name of the mixer the channel uses. The three mixers Ren'Py knows about by default are "music", "sfx", and "voice".

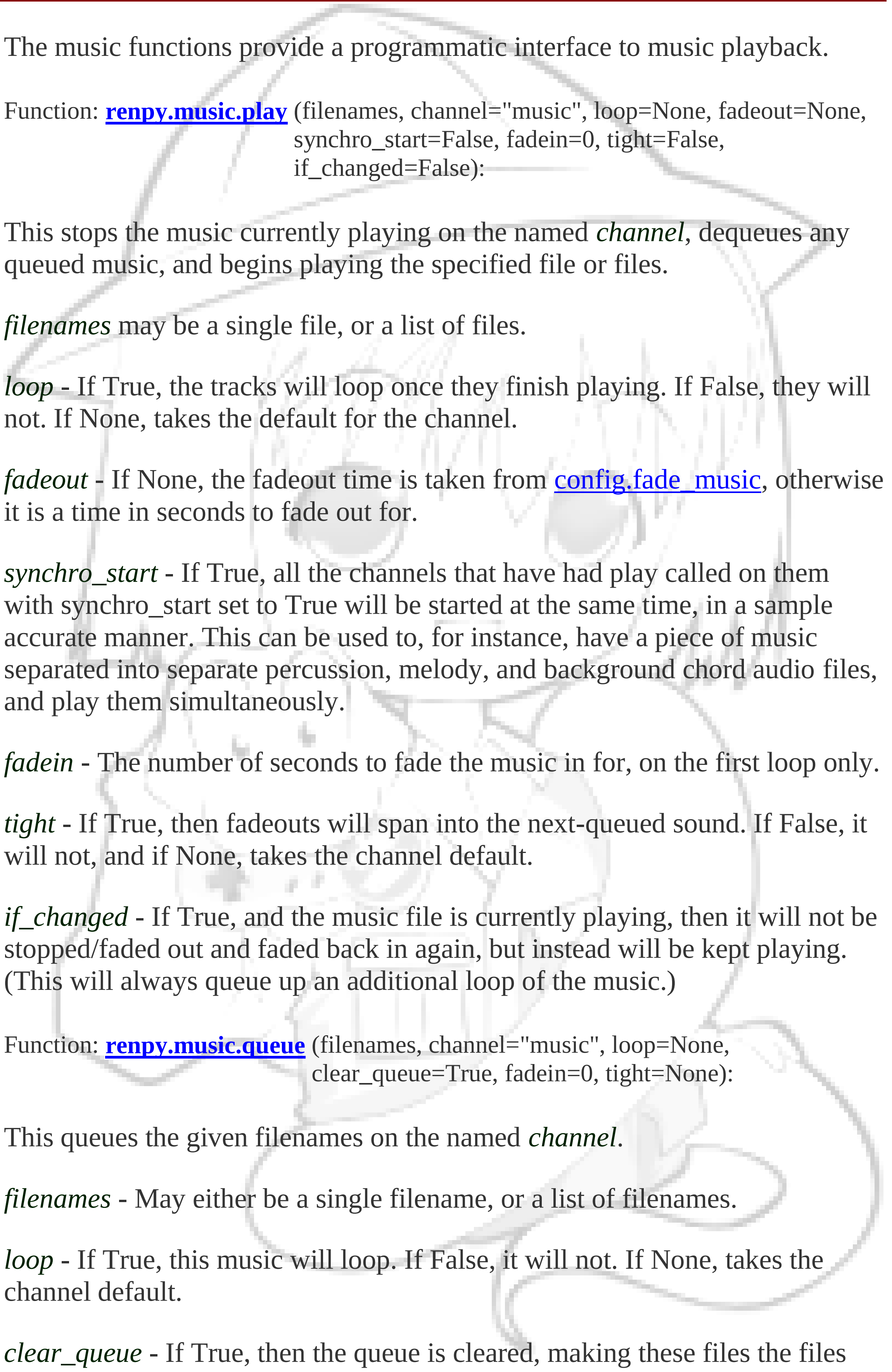
loop - Determines if sounds on this channel loop by default.

tight - Determines if sounds loop even during fadeout.

file_prefix - A prefix that is prepended to filenames provided to this channel before they are played.

file_suffix - A suffix that is appended to filenames provided to this channel before they are played.

Music Functions



The music functions provide a programmatic interface to music playback.

Function: **renpy.music.play** (filenames, channel="music", loop=None, fadeout=None, synchro_start=False, fadein=0, tight=False, if_changed=False):

This stops the music currently playing on the named *channel*, dequeues any queued music, and begins playing the specified file or files.

filenames may be a single file, or a list of files.

loop - If True, the tracks will loop once they finish playing. If False, they will not. If None, takes the default for the channel.

fadeout - If None, the fadeout time is taken from config.fade_music, otherwise it is a time in seconds to fade out for.

synchro_start - If True, all the channels that have had play called on them with synchro_start set to True will be started at the same time, in a sample accurate manner. This can be used to, for instance, have a piece of music separated into separate percussion, melody, and background chord audio files, and play them simultaneously.

fadein - The number of seconds to fade the music in for, on the first loop only.

tight - If True, then fadeouts will span into the next-queued sound. If False, it will not, and if None, takes the channel default.

if_changed - If True, and the music file is currently playing, then it will not be stopped/faded out and faded back in again, but instead will be kept playing. (This will always queue up an additional loop of the music.)

Function: **renpy.music.queue** (filenames, channel="music", loop=None, clear_queue=True, fadein=0, tight=None):

This queues the given filenames on the named *channel*.

filenames - May either be a single filename, or a list of filenames.

loop - If True, this music will loop. If False, it will not. If None, takes the channel default.

clear_queue - If True, then the queue is cleared, making these files the files that are played when the currently playing file finishes. If it is False, then these files are placed at the back of the queue. In either case, if no music is playing these files begin playing immediately.

fadein - The number of seconds to fade the music in for, on the first loop only.

tight - If True, then fadeouts will span into the next-queued sound. If False, it will not, and if None, takes the channel default.

Function: **renpy.music.stop** (channel="music", fadeout=None):

This stops the music that is currently playing on the named *channel*, dequeues all queued music, and sets the last queued file to None.

fadeout - If None, the music is faded out for the time given in config.fade_music, otherwise it is faded for the given number of seconds.

Function: **renpy.music.set_volume** (volume, delay=0, channel="music"):

This sets the volume of the named *channel*. The *volume* is a number between 0.0 and 1.0, and is interpreted as a fraction of the mixer volume for the channel.

It takes *delay* seconds to change/fade the volume from the old to the new value. This value is persisted into saves, and participates in rollback.

Function: **renpy.music.set_pan** (pan, delay, channel="music"):

This function allows sound and music to be panned between the two stereo channels.

pan - A number between -1 and 1 that control the placement of the audio. If this is -1, then all audio is sent to the left channel. If it's 0, then the two channels are equally balanced. If it's 1, then all audio is sent to the right ear.

delay - The amount of time it takes for the panning to occur.

channel - The channel the panning takes place on. This can be a sound or a music channel. Often, this is channel 7, the default music channel.

Function: **renpy.music.get_playing** (channel="music"):

Returns the filename of the music playing on the given *channel*, or None if no music is playing on that channel. Note that None may be returned when the user sets the music volume to zero, even if the game script requested that music be played on that channel.

Function: **renpy.music.set_volume** (volume, delay=0, channel="music"):

This sets the volume of the named *channel*. The *volume* is a number between 0.0 and 1.0, and is interpreted as a fraction of the mixer volume for the channel.

It takes *delay* seconds to change/fade the volume from the old to the new value. This value is persisted into saves, and participates in rollback.

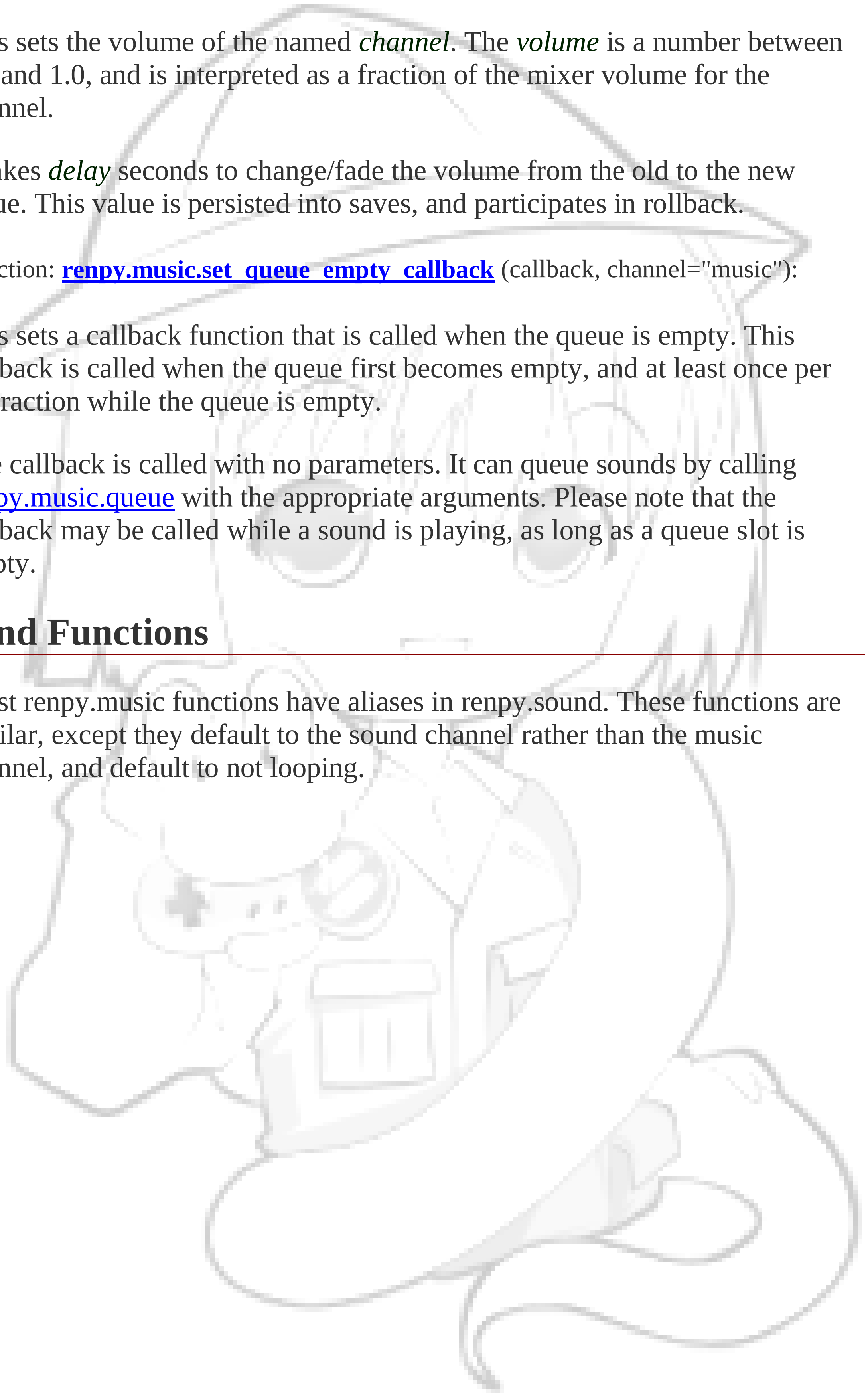
Function: **renpy.music.set_queue_empty_callback** (callback, channel="music"):

This sets a callback function that is called when the queue is empty. This callback is called when the queue first becomes empty, and at least once per interaction while the queue is empty.

The callback is called with no parameters. It can queue sounds by calling renpy.music.queue with the appropriate arguments. Please note that the callback may be called while a sound is playing, as long as a queue slot is empty.

Sound Functions

Most renpy.music functions have aliases in renpy.sound. These functions are similar, except they default to the sound channel rather than the music channel, and default to not looping.



Movies

Ren'Py is capable of using ffmpeg (included) to play movies using the video codecs:

- Theora
- MPEG 4 part 2 (including Xvid and DivX)
- MPEG 2
- MPEG 1

and the following audio codecs:

- Vorbis
- MP3
- MP2
- PCM

inside the following container formats:

- Matroska
- Ogg
- Avi
- Various kinds of MPEG stream.

(Note that using some of these formats may require patent licenses. When in doubt, and especially for commercial games, we recommend using Theora, Vorbis, and Matroska or Ogg.)

Ren'Py expects that every movie will have an audio track associated with it, even if that audio track consists of nothing but silence. This is because the audio track is used for synchronization purposes.

Movies can be displayed fullscreen, or in a displayable. Fullscreen movies are the more efficient.

Fullscreen Movies

The easiest way to display a movie fullscreen is to display it using the renpy.movie_cutscene function. This function displays a movie for a specified length of time. When that time has elapsed, or when the user clicks to dismiss the movie, the movie ends and the function returns.

Function: **renpy.movie_cutscene** (filename, delay=None, loops=0, stop_music=True):

This displays a fullscreen movie cutscene.

filename - The name of a file containing a movie.

delay - The number of seconds to wait before ending the cutscene. Normally the length of the movie, in seconds. If None, the length of the movie will be automatically determined. If -1, the cutscene will not automatically terminate, and will continue until the user clicks.

loops - The number of extra loops to show, -1 to loop forever.

stop_music - If True, stops the music channel while the cutscene is playing, and starts it again when the cutscene is over.

Returns True if the movie was terminated by the user, or False if the given delay elapsed uninterrupted.

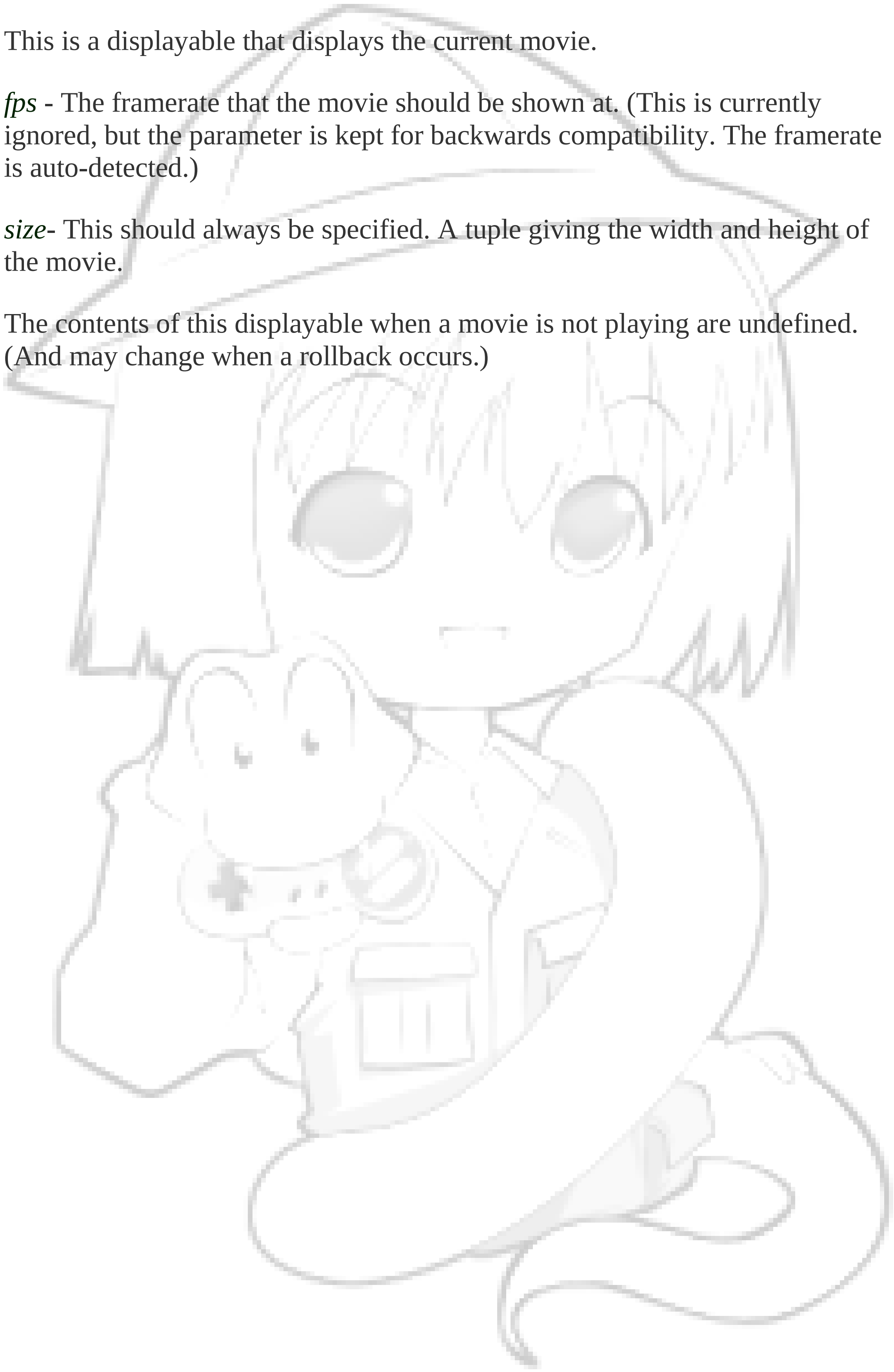
```
   $ renpy.movie_cutscene("On_Your_Mark.mpg")
```

Movies Inside Displayables

A movie can also be displayed inside a displayable, allowing it to be combined with other things on the screen. To do this, one must first show a Movie displayable, and then play the movie on an audio channel. (We recommend using the movie channel for this purpose.)

```
init:
    image movie = Movie(size=(400, 300), xalign=0.5, yalign=0.5)

label movie_sign:
    scene black
    show movie

    play movie "incubus.mkv"

    "Wow, this movie is really terrible."

    "I mean, it stars William Shatner..."

    "... speaking Esperanto."

    "MAKE IT STOP!"

    stop movie
    hide movie

    "Thats... better."
```

Function: **Movie** (fps=24, size=None, **properties):



This is a displayable that displays the current movie.

fps - The framerate that the movie should be shown at. (This is currently ignored, but the parameter is kept for backwards compatibility. The framerate is auto-detected.)

size- This should always be specified. A tuple giving the width and height of the movie.

The contents of this displayable when a movie is not playing are undefined. (And may change when a rollback occurs.)

UI Functions

While the say and menu statements are often enough for many games, there are times when more complex user interfaces are desired. For example, dating sim games can require more complex scheduling screens. These screens can be built using the UI functions.

Most of the UI functions create widgets and add them to the screen. The UI functions manage the complexity of nesting widgets. The screen can then be shown to the user with ui.interact, and a value returned and used by the game script.

To start, let's give a script snippet that uses the UI functions. The following displays a window containing three buttons, arraigned horizontally. The string assigned to `choice` varies depending on which of the three is picked.

```
$ ui.window()
$ ui.hbox()

$ ui.textbutton("A", clicked=ui.returns("A"))
$ ui.textbutton("B", clicked=ui.returns("B"))
$ ui.textbutton("C", clicked=ui.returns("C"))

$ ui.close()
$ choice = ui.interact(suppress_overlay=True)
```

There are three kinds of widgets that can be created by the UI functions.

- The first kind of widget (of which ui.window is one), takes a single child widget. When this kind of widget is open, the next widget to be created is added to it, and it then closes automatically.

- The second kind of widget can take an arbitrary number of child widgets. This kind of widget is exemplified by the layout widgets: vbox, hbox, and fixed. New widgets are added to this kind of widget until a matching call to ui.close occurs, unless they are instead added to an open child of this widget.

- The third kind of widget cannot take any children. This kind of widget is exemplified by ui.textbutton.

There is also a set of functions that are used to manage the interaction, and a set of functions that can be used to perform various actions when buttons are clicked.

Single-Child Widgets

Function: **ui.button** (clicked=None, **properties):

This creates a button that can be clicked by the user. When this button is clicked or otherwise selected, the function supplied as the clicked argument is called. If it returns a value, that value is returned from ui.interact.

Buttons created with this function contain another widget, specifically the next widget to be added. As a convenience, one can use ui.textbutton to create a button with a text label. See also ui.imagebutton, which creates a button in the form of an image.

clicked - A function that is called with no arguments when this button is clicked. If it returns a non-None value, then that value is returned from ui.interact. If clicked itself is None, the button is displayed in the "insensitive" (disabled) style.

hovered - A function that is called with no arguments when this button is focused. If it returns a non-None value, then that value is returned from ui.interact.

unhovered - A function that is called with no arguments when the button loses focus. It's also called at least once per interaction, when a button is not focused at the start of the interaction. The return value is currently ignored, but should be None.

role - The role this button undertakes. This can be the empty string, or "selected_".

keymap - A keymap that is used when this button is focused.

Note that code in the clicked, hovered, and unhovered methods is run inside the current interaction. This means that the screen is not cleared while this code is run. Displayables may be added or removed from the current interaction, provided renpy.restart_interaction is called to let Ren'Py know that the interaction has been changed. You should not run code that causes a new interaction from inside these functions, except inside a new context using renpy.call_in_new_context, renpy.invoke_in_new_context, or ui.callsinnewcontext.

Function: **ui.frame** (**properties):

A frame contains a single widget. This is similar to a window, but may be styled differently.

Function: **ui.sizer** (maxwidth=None, maxheight=None, **properties):

This is a widget that can shrink the size allocated to the next widget added. If maxwidth or maxheight is not None, then the space allocated to the child in the appropriate direction is limited to the given amount.

Please note that this only works with child widgets that can have a limited area allocated to them (like text), and not with ones that use a fixed area (like images).

maxwidth - The maximum width of the child widget, or None to not affect width.

maxheight - The maximum height of the child widget, or None ot not affect height.

Function: **ui.transform** (function=None, alpha=1, rotate=None, zoom=1, xzoom=1, yzoom=1, style='transform', **properties):

Wraps its child in a Transform, allowing the child to be rotated, zoomed, and alpha-modified. Parameters are as for Transform.

Function: **ui.viewport** (child_size=(None, None), xadjustment=None, yadjustment=None, set_adjustments=True, mousewheel=False, draggable=False, style='viewport', **properties):

Displays a viewport on the screen. A viewport restricts the size of its child, and allows the child to be displayed at an offset.

child_size - The x and y size of the area the child is asked to render. If either is None, defaults to the size of this viewport.

xadjustment - A ui.adjustment that's used for the x-axis of the viewpoert.

yadjustment - A ui.adjustment that's used for the y-axis of the viewport.

set_adjustments - If true, the range and page size of the adjustments will be set by this viewport.

mousewheel - If true, the mouse wheel can be used to scroll the viewport.

draggable - If true, the mouse can be used to drag around the viewport.

If xadjustment or yadjustment are None, adjustments are created automatically. These adjustments are available through the xadjustment and yadjustment properies of a viewport.

In general, viewports are only useful when the xmaximum and ymaximum properties are specified. You'll also want to set clipping=True on the style, although this is part of the default viewport style.

Function: **ui.window** (**properties):

A window contains a single widget. It draws that window atop a background and with appropriate amounts of margin and padding, taken from the window properties supplied to this call. The next widget created is added to this window.

Multiple-Child Widgets

Function: **ui.fixed** (**properties):

This creates a layout that places widgets at fixed locations relative to the origin of the enclosing widget. The layout takes up the entire area allocated to it. New widgets are added to this widget until the next call to ui.close.

Function: **ui.grid** (cols, rows, padding=0, transpose=False, **properties):

This creates a layout that places widgets in an evenly spaced grid. New widgets are added to this grid until ui.close is called. Widgets are added by going from left to right within a single row, and down to the start of the next row when a row is full. All cells must be filled (that is, exactly col * rows widgets must be added to the grid.)

The children of this widget should have a fixed size that does not vary based on the space allocated to them. Failure to observe this restriction could lead to really odd layouts, or things being rendered off screen. This condition is relaxed in the appropriate dimension if xfill or yfill is set.

Each cell of the grid is exactly the same size. By default, the grid is the smallest size that can accommodate all of its children, but it can be expanded to consume all available space in a given dimension by setting xfill or yfill to True, as appropriate. (Otherwise, xfill and yfill are inherited from the style.)

cols - The number of columns in this grid.

rows - The number of rows in this grid.

padding - The amount of space to leave between rows and columns.

xfill - True if the grid should consume all available width.

yfill - True if the grid should consume all available height.

transpose - If True, grid will fill down columns before filling across rows.

Function: **ui.hbox** (spacing=None, style='hbox', **properties):

This creates a layout that places widgets next to each other, from left to right. New widgets are added to this hbox until ui.close is called.

spacing - The number of pixels to leave between widgets. If None, take the amount of spacing from the style.

Function: **ui.vbox** (spacing=None, style='vbox', **properties):

This creates a layout that places widgets next to each other, from top to bottom. New widgets are added to this vbox until ui.close is called.

spacing - The number of pixels to leave between widgets. If None, take the amount of spacing from the style.

Function: **ui.side** (places, style='default', **properties):

This positions child widgets so that they surround a parent widget. It takes a single argument, *places*, which controls where each child widget is placed. Places is expected to be iterable, and each element should be in the list:

'c', 't', 'b', 'l', 'r', 'tl', 'tr', 'bl', 'br'

'c' means center, 't' top, 'tl' top left, 'br' bottom right, and so on. A side should be given the same number of children as the number of entries in the places list.

The top and bottom children are rendered with a 0 requested height, while the left and right children are rendered with a 0 requested width. It is therefore suggested that you enforce a minimum width on the children that are placed in these slots.

The center widget is rendered before the others. Other than that, the order widgets are rendered in is undefined.

No-Child Widgets

Function: **ui.autobar** (range, start, end, time, **properties):

Creates a bar (with a range of *range*) that automatically moves from *start* to *end* in *time* seconds.

Function: **ui.bar** (range=None, value=None, changed=None, step=None, page=None, adjustment=None, style='bar', **properties):

This creates a bar widget. The bar widget can be used to display data in a bar graph format, and optionally to report when the user clicks on a location in that bar.

adjustment - This should always be given as a keyword argument. If not none, it should be a ui.adjustment that is to used by this bar, and the following arguments are ignored. If None, a new ui.adjustment is created from the *range*, *value*, *changed*, *step*, and *page* arguments.

style - The style of this bar. As of 6.2.0, there are four styles that you can use:

- **bar** - A horizontal bar, with the largest value to the right.
- **scrollbar** - A horizontal scrollbar, with the largest value to the right.
- **vbar** - A vertical bar, with the largest value to the top.
- **vscrollbar** - A vertical scrollbar, with the largest value to the bottom.

The width and height should be set with the xmaximum and ymaximum properties. For best results, if clicked is set then width should be at least twice as big as range.

Function: **ui.image** (im, **properties):

This loads an image, and displays it as a widget. The image may be the name of a file containing the image, or an arbitrary displayable.

Function: **ui.imagebutton** (idle_image, hover_image, clicked=None, image_style='image_button_image', **properties):

This creates a button that contains two images. The first is the idle image, which is used when the mouse is not over the image, while the second is the hover image, which is used when the mouse is over the image. If the button is clicked or otherwise selected, then the clicked argument is called. If it returns a value, that value is returned from ui.interact.

idle_image - The file name of the image used when this button is idle (doesn't have the mouse pointer over it, or focus from keyboard or joystick). Must be specified.

hover_image - The file name of the image used when this button is hovered (the mouse pointer is over it, or the user has focused it with keyboard or joystick). Must be specified.

clicked - The function that is called when this button is clicked.

hovered - A function that is called with no arguments when this button is hovered. If it returns a non-None value, then that value is returned from ui.interact.

unhovered - A function that is called with no arguments when this button loses focus.

image_style - The style that is applied to the images that are used as part of the imagebutton.

role - The role this button undertakes. This can be the empty string, or "selected_".

insensitive_image - If specified, the file name of the image used when this button is disabled (clicked is None).

activate_image - If specified, the file name of the image used when this button is activated (the mouse button is being held down over it, or equivalents for keyboard and joystick).

selected_idle_image, *selected_hover_image*, *selected_insensitive_image* and *selected_activate_image* also exist. They are the equivalents of *idle_image*, *hover_image*, *insensitive_image* and *activate_image* which are used when this button is selected (represents the current state, like the "Save Game" button on the save game screen).

Function: **ui.imagemap** (ground, selected, hotspots, unselected=None, style='imagemap', button_style='imagemap_button', **properties):

This is called to create imagemaps. Parameters are roughly the same as renpy.imagemap. The value of the hotspot is returned when ui.interact returns.

Function: **ui.input** (default, length=None, allow=None, exclude='{}', button=None, changed=None, prefix="", suffix="", **properties):

This displays a text area that accepts input from the user. Only ASCII is input reliably, although some non-ASCII languages may also work on some platforms.

default - The default value for the text area.

length - If not None, a limit on the number of characters that can be input.

allow - If not None, a string containing the characters that are allowed.

exclude - A string containing characters that are no allowed. The brace characters should always be disallowed, since they may accidentally cause a text tag to be shown.

button - If not None, then this should be a button. The input is only active when this button is focused.

changed - If not None, then this function is called when the text is changed.

prefix - A prefix included before the text to be edited.

suffix - A suffix included after the text to be edited.

If changed is None, then the text is returned as the result of ui.interact() when enter is pressed. Otherwise, this cannot cause ui.interact() to terminate.

Function: **ui.keymap** (**keymap):

This is a pseudo-widget that adds a keymap to the screen. This function takes as keyword arguments the names of bindings or keysyms. These keywords should be given as their arguments a function, which is called with zero arguments when an appropriate keysym is pressed.

Function: **ui.menu** (menuitems, style='menu', caption_style='menu_caption', choice_style='menu_choice', choice_chosen_style='menu_choice_chosen', choice_button_style='menu_choice_button', choice_chosen_button_style='menu_choice_chosen_button', location=None, focus=None, default=False, **properties):

This creates a new menu widget. Unlike the `menu` statement or renpy.display_menu function, this menu widget is not enclosed in any sort of window. You'd have to do that yourself, if it is desired.

menuitems - A list of tuples that are the items to be added to this menu. The first element of a tuple is a string that is used for this menuitem. The second element is the value to be returned from ui.interact if this item is selected, or None if this item is a non-selectable caption.

location - Some serializable and hashable object that represents the location of this menu in the game. (Normally, this is the name of the statement executing the menu.) If provided, then this logs which menuitems have been chosen, and changes the style of those menuitems to the choice_seen_style.

Function: **ui.null** (width=0, height=0, **properties):

This widget displays nothing on the screen. Why would one want to do this? If a widget requires contents, but you don't have any contents to provide it. It can also be used to create empty space, by giving non-zero *width* and *heigh* arguments.

Function: **ui.pausebehavior** (delay, result=False):

This is a pseudo-widget that adds the pause behavior to the screen. The pause behavior is to return the supplied result when the given number of seconds elapses. This widget should not be added to any other widget, but should instead be only added to the screen itself.

Please note that this widget will always pause for the given amount of time. If you want a pause that can be interrupted by the user, add in a saybehavior.

delay - The amount of time to pause, in seconds.

result - The result that will be retuned after the delay time elapses.

Function: **ui.saybehavior** (afm=None, dismiss=['dismiss'], allow_dismiss=None):

This is a pseudo-widget that adds the say behavior to the screen. The say behavior is to return True if the left mouse is clicked or enter is pressed. It also returns True in various other cases, such as if the current statement has already been seen. This widget should not be added to any other widget, but should instead be only added to the screen itself.

If *afm* is present, it is a block of text, that's given to the auto forwarding mode algorithm to determine the auto-forwarding timeout.

If *dismiss* is present, it is a list of names of keybindings that are used to dismiss this saybehavior.

If *allow_dismiss* is present, it should be a function. This function is called without any arguments each time the user initiates a dismiss request. If the function returns True, the dismiss request is allowed, otherwise it is ignored. *Added in 5.6.6.*

There should only be one saybehavior shown at a time, otherwise they will fight for focus.

Changes

The dismiss parameter was changed in 5.6.1 from a single keybinding to a list of keybindings.

Function: **ui.soundstopbehavior** (channel, result=False):

Adding this behavior to the screen causes ui.interact to return *result* when there is no sound playing on the named music channel.

Function: **ui.text** (label, **properties):

This creates a widget displaying a text label.

label - The text that will be displayed on the screen. It uses font properties. The label can also be a list containing both text strings in widgets, in the case of a widget, the widget is displayed as if it was text of the enclosing font. The height of the area allocated is that of the font, with the width being taked from a render of the widget.

slow - If True, the text will be typed at the screen at a rate determined by the slow_cps property, if set, or the "Text Speed" preference. If None (the default), then it will be typed at a speed determined by the slow_cps property. If False, then it will appear instantly.

slow_speed - If slow is True, then this is the number of cps the text is displayed at, overriding the preference.

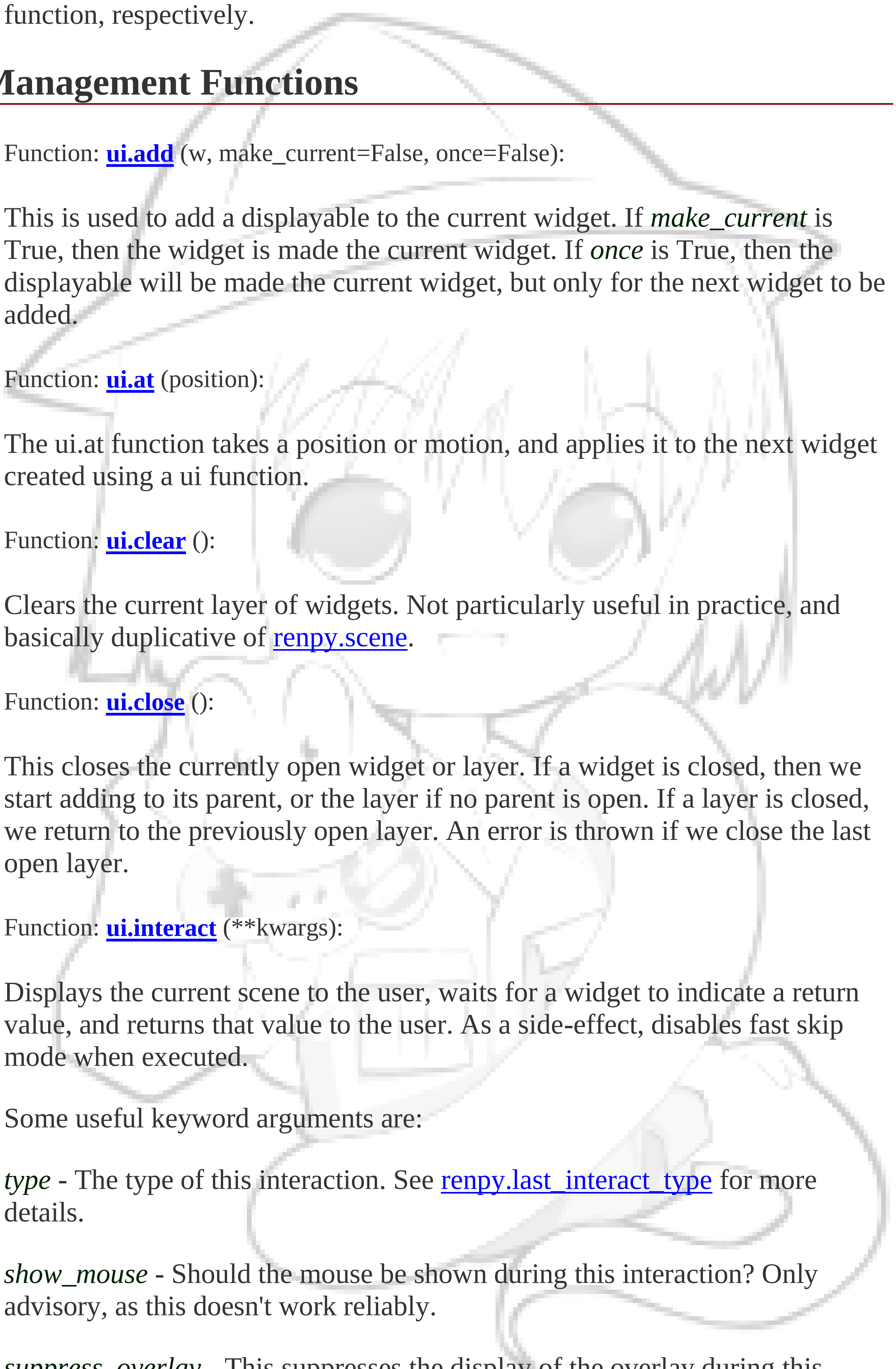
slow_done - If not None and slow is True, this is a callback that is called when we're done displaying text on the screen.

Function: **ui.timer** (seconds, function, repeat=False, args=(), kwargs={}):

This sets up a timer that will call *function* after *seconds* seconds have elapsed. If *repeat* is true, then the function is called every *seconds* seconds thereafter.

args and *kwargs* are the positional and keyword arguments supplied to the function, respectively.

Management Functions



Function: **ui.add** (w, make_current=False, once=False):

This is used to add a displayable to the current widget. If *make_current* is True, then the widget is made the current widget. If *once* is True, then the displayable will be made the current widget, but only for the next widget to be added.

Function: **ui.at** (position):

The ui.at function takes a position or motion, and applies it to the next widget created using a ui function.

Function: **ui.clear** ():

Clears the current layer of widgets. Not particularly useful in practice, and basically duplicative of renpy.scene.

Function: **ui.close** ():

This closes the currently open widget or layer. If a widget is closed, then we start adding to its parent, or the layer if no parent is open. If a layer is closed, we return to the previously open layer. An error is thrown if we close the last open layer.

Function: **ui.interact** (**kwargs):

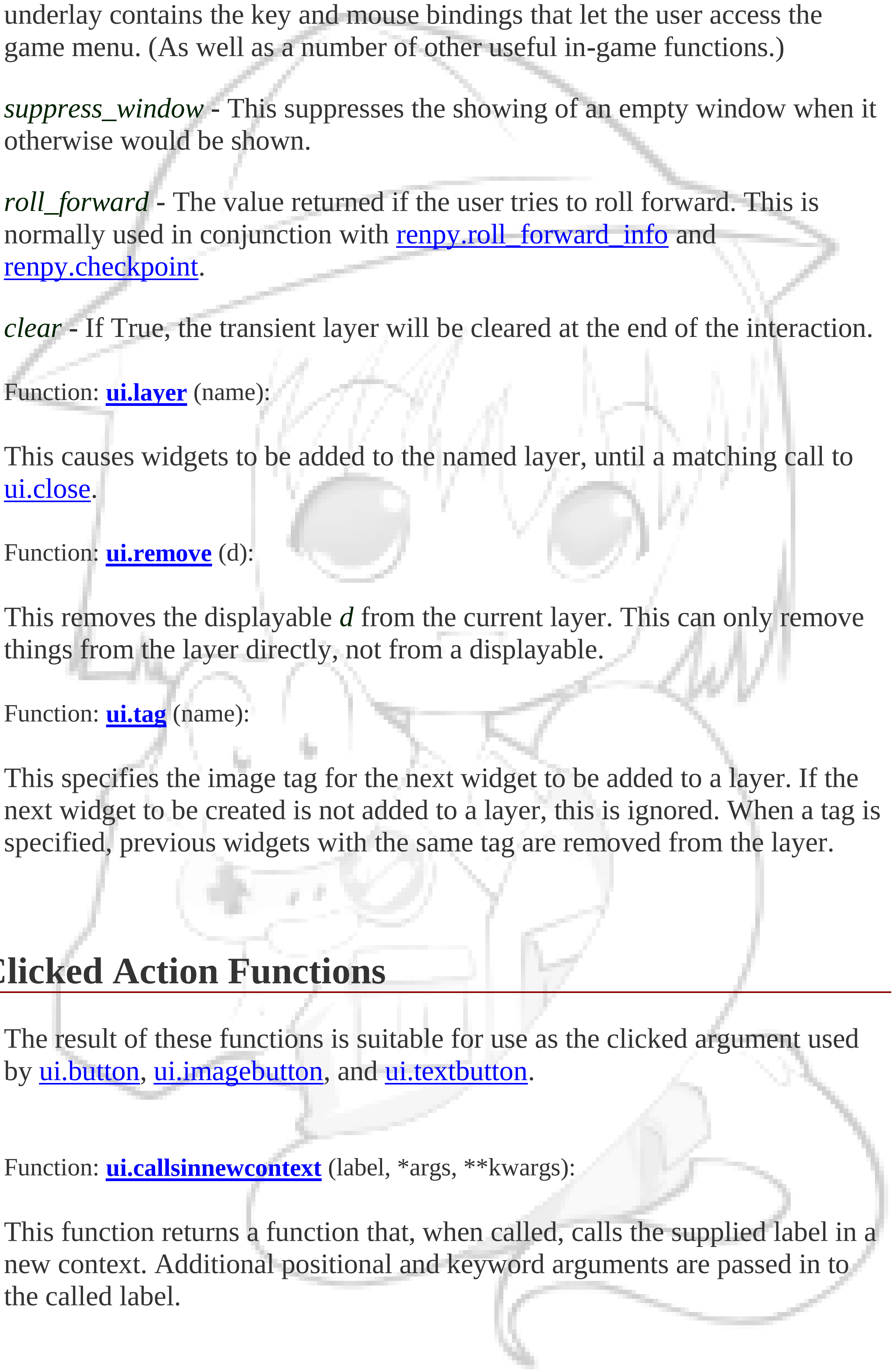
Displays the current scene to the user, waits for a widget to indicate a return value, and returns that value to the user. As a side-effect, disables fast skip mode when executed.

Some useful keyword arguments are:

type - The type of this interaction. See renpy.last_interact_type for more details.

show_mouse - Should the mouse be shown during this interaction? Only advisory, as this doesn't work reliably.

suppress_overlay - This suppresses the display of the overlay during this interaction.



suppress_underlay - This suppresses the underlay during this interaction. The underlay contains the key and mouse bindings that let the user access the game menu. (As well as a number of other useful in-game functions.)

suppress_window - This suppresses the showing of an empty window when it otherwise would be shown.

roll_forward - The value returned if the user tries to roll forward. This is normally used in conjunction with renpy.roll_forward_info and renpy.checkpoint.

clear - If True, the transient layer will be cleared at the end of the interaction.

Function: **ui.layer** (name):

This causes widgets to be added to the named layer, until a matching call to ui.close.

Function: **ui.remove** (d):

This removes the displayable *d* from the current layer. This can only remove things from the layer directly, not from a displayable.

Function: **ui.tag** (name):

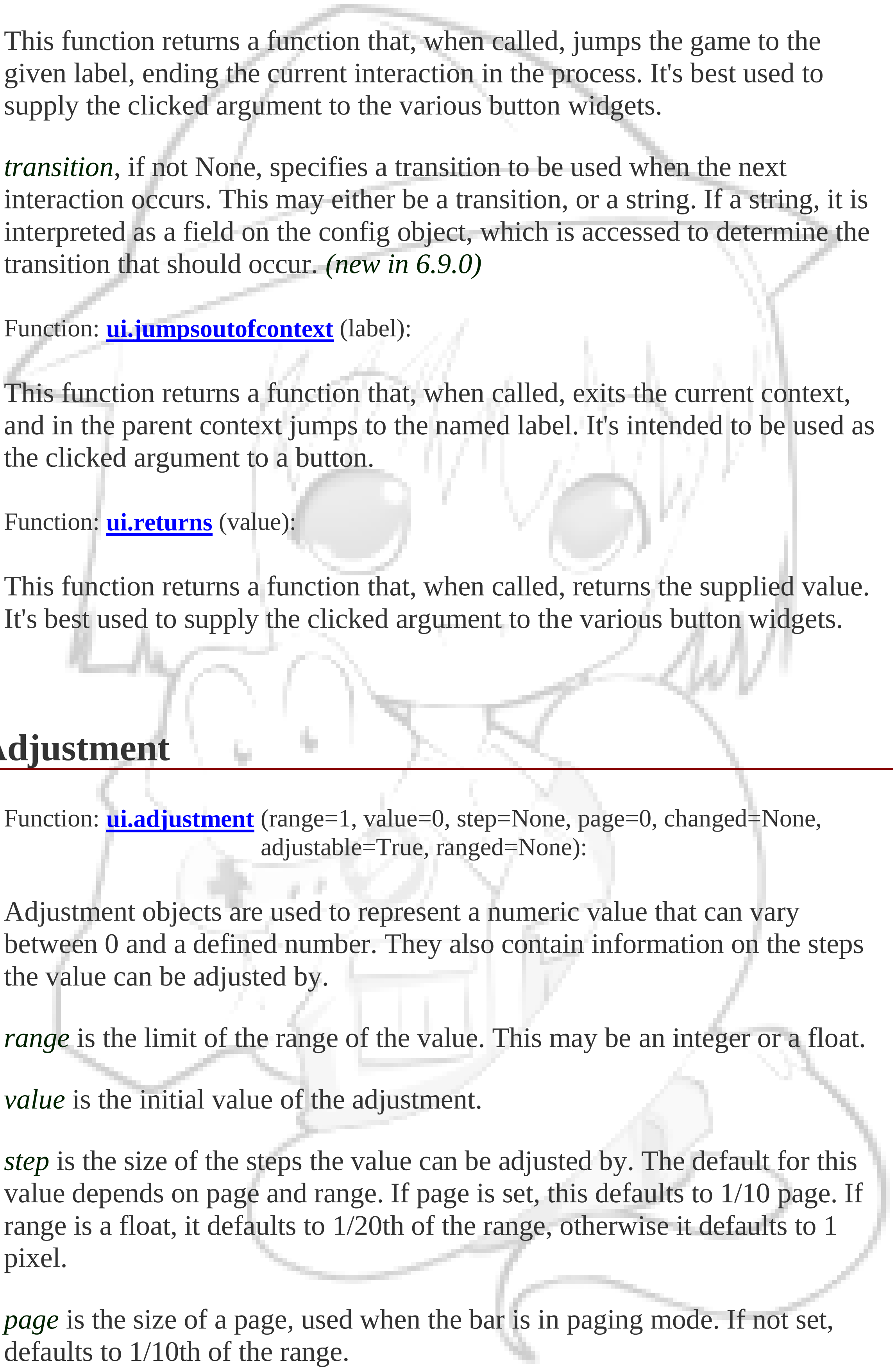
This specifies the image tag for the next widget to be added to a layer. If the next widget to be created is not added to a layer, this is ignored. When a tag is specified, previous widgets with the same tag are removed from the layer.

Clicked Action Functions

The result of these functions is suitable for use as the clicked argument used by ui.button, ui.imagebutton, and ui.textbutton.

Function: **ui.callsinnewcontext** (label, *args, **kwargs):

This function returns a function that, when called, calls the supplied label in a new context. Additional positional and keyword arguments are passed in to the called label.



Function: **ui.jumps** (label, transition=None):

This function returns a function that, when called, jumps the game to the given label, ending the current interaction in the process. It's best used to supply the clicked argument to the various button widgets.

transition, if not None, specifies a transition to be used when the next interaction occurs. This may either be a transition, or a string. If a string, it is interpreted as a field on the config object, which is accessed to determine the transition that should occur. *(new in 6.9.0)*

Function: **ui.jumpsoutofcontext** (label):

This function returns a function that, when called, exits the current context, and in the parent context jumps to the named label. It's intended to be used as the clicked argument to a button.

Function: **ui.returns** (value):

This function returns a function that, when called, returns the supplied value. It's best used to supply the clicked argument to the various button widgets.

Adjustment

Function: **ui.adjustment** (range=1, value=0, step=None, page=0, changed=None, adjustable=True, ranged=None):

Adjustment objects are used to represent a numeric value that can vary between 0 and a defined number. They also contain information on the steps the value can be adjusted by.

range is the limit of the range of the value. This may be an integer or a float.

value is the initial value of the adjustment.

step is the size of the steps the value can be adjusted by. The default for this value depends on page and range. If page is set, this defaults to 1/10 page. If range is a float, it defaults to 1/20th of the range, otherwise it defaults to 1 pixel.

page is the size of a page, used when the bar is in paging mode. If not set, defaults to 1/10th of the range.

These four arguments correspond to fields on the adjustment object. The fields can be safely set before anything using the adjustment has been rendered. After that, they should be treated as read-only.

changed is a function that's called when the adjustment is changed. The function is called with one argument, the new value of the adjustment. This should be present for the bar to be adjustable.

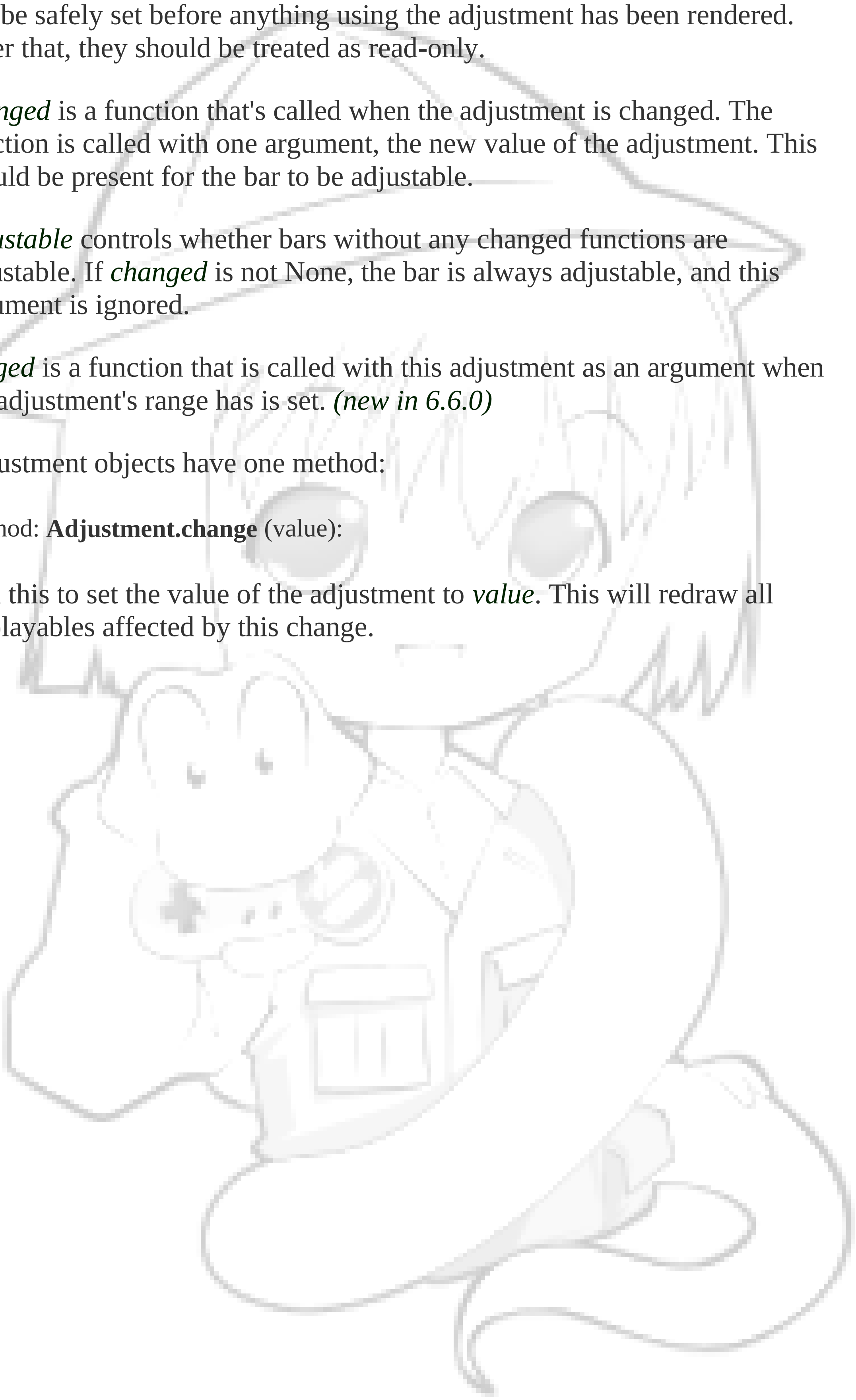
adjustable controls whether bars without any changed functions are adjustable. If *changed* is not None, the bar is always adjustable, and this argument is ignored.

ranged is a function that is called with this adjustment as an argument when the adjustment's range has is set. *(new in 6.6.0)*

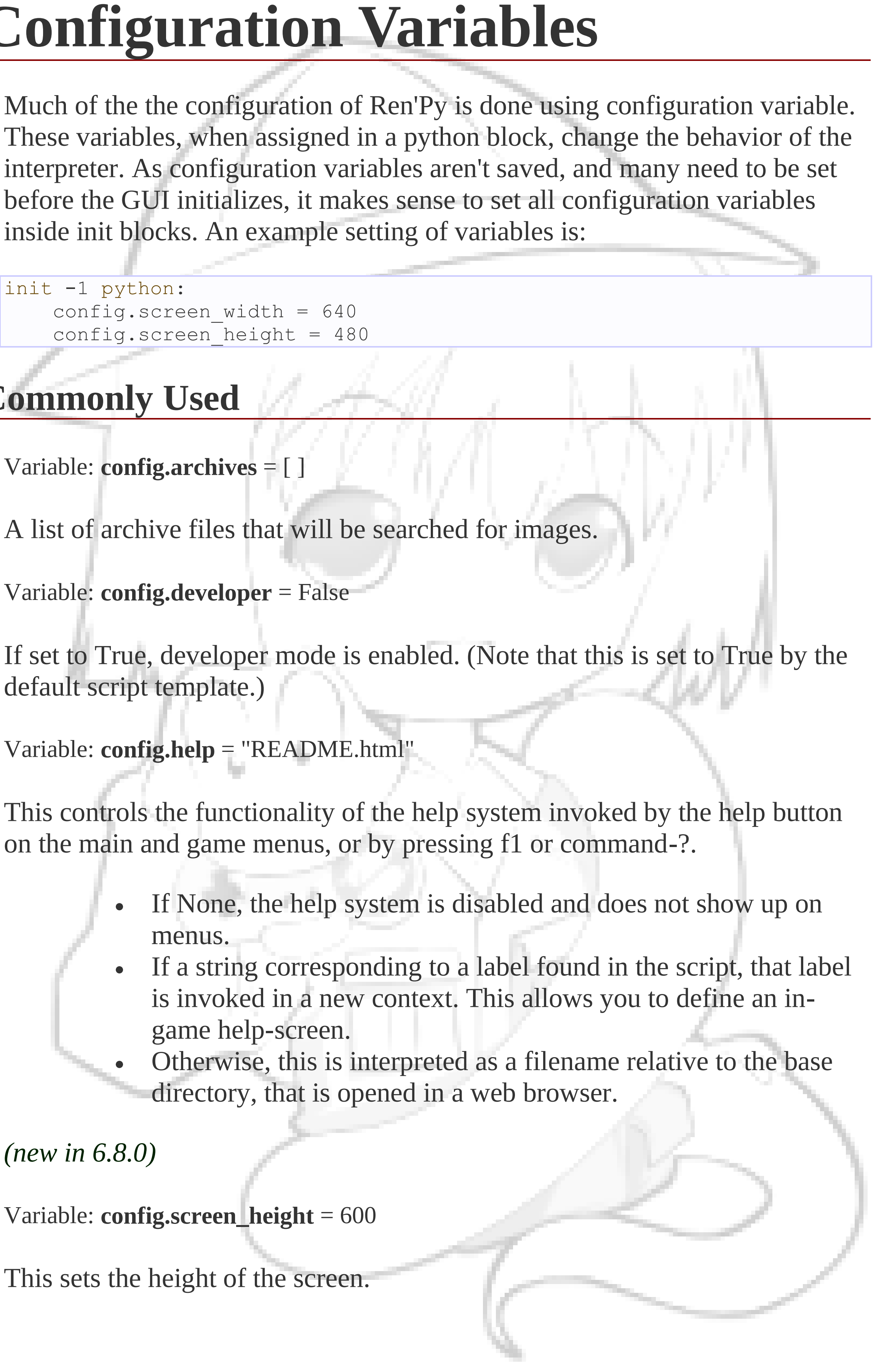
Adjustment objects have one method:

Method: **Adjustment.change** (value):

Call this to set the value of the adjustment to *value*. This will redraw all displayables affected by this change.



Configuration Variables



Much of the the configuration of Ren'Py is done using configuration variable. These variables, when assigned in a python block, change the behavior of the interpreter. As configuration variables aren't saved, and many need to be set before the GUI initializes, it makes sense to set all configuration variables inside init blocks. An example setting of variables is:

```
init -1 python:
    config.screen_width = 640
    config.screen_height = 480
```

Commonly Used

Variable: **config.archives** = [ ]

A list of archive files that will be searched for images.

Variable: **config.developer** = False

If set to True, developer mode is enabled. (Note that this is set to True by the default script template.)

Variable: **config.help** = "README.html"

This controls the functionality of the help system invoked by the help button on the main and game menus, or by pressing f1 or command-?.

- If None, the help system is disabled and does not show up on menus.
- If a string corresponding to a label found in the script, that label is invoked in a new context. This allows you to define an in-game help-screen.
- Otherwise, this is interpreted as a filename relative to the base directory, that is opened in a web browser.

(new in 6.8.0)

Variable: **config.screen_height** = 600

This sets the height of the screen.

Variable: **config.screen_width** = 800

This sets the width of the screen.

Variable: **config.translations** = dict(...)

This is a map used to translate text in the game menu into your language. See Localizing Ren'Py for how to use it, and here for a list of available translations.

Variable: **config.window_icon** = None

If not None, this is expected to be the filename of an image giving an icon that is used for the window on Linux and Mac OS X. This should be a large image, with 8-bit alpha.

Variable: **config.windows_icon** = None

If not None, this is expected to be the filename of an image giving an icon that is used for the window on Windows. This should be a 32x32 image with 1-bit alpha. (Opaque images work the best.)

Variable: **config.window_title** = "A Ren'Py Game"

This is prepended to _window_subtitle to produce the caption for the window.

Variable: **config.save_directory** = "..."

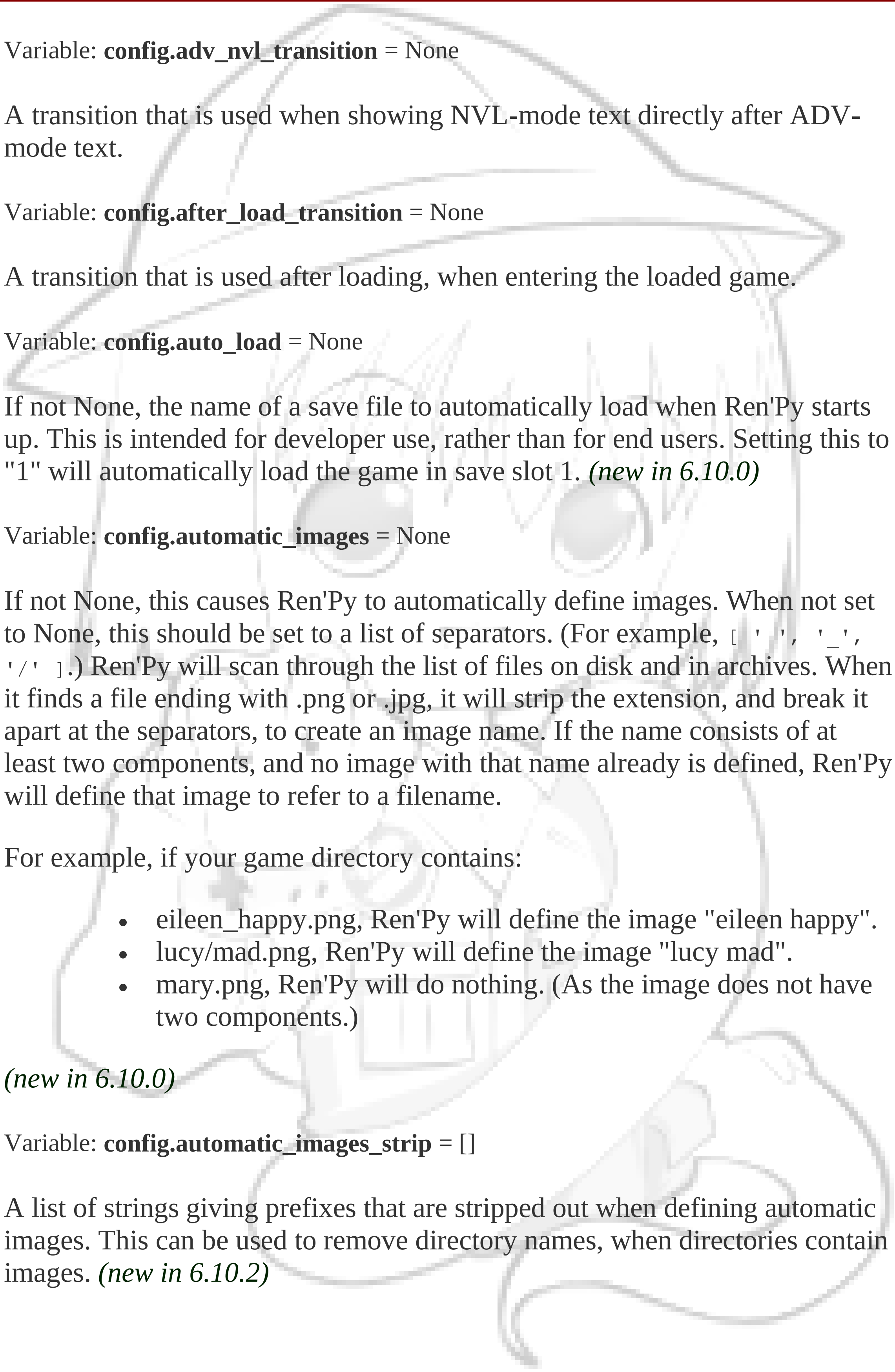
This is used to generate the default directory in which games and persistent information are saved. The directory name generated depends on the platform:

- Windows: %APPDATA%/RenPy/*save_directory*
- Mac OS X: ~/Library/RenPy/*save_directory*
- Linux/Other: ~/.renpy/*save_directory*

This should be set in a python early block, as the save directory name must be known before init code runs, as persistent information is made available to init code.

This directory may be overridden by user invocation. Code that needs to know the save directory should read config.savedir instead. *(new in 6.7.0)*

Occasionally Used



Variable: **config.adv_nvl_transition** = None

A transition that is used when showing NVL-mode text directly after ADV-mode text.

Variable: **config.after_load_transition** = None

A transition that is used after loading, when entering the loaded game.

Variable: **config.auto_load** = None

If not None, the name of a save file to automatically load when Ren'Py starts up. This is intended for developer use, rather than for end users. Setting this to "1" will automatically load the game in save slot 1. *(new in 6.10.0)*

Variable: **config.automatic_images** = None

If not None, this causes Ren'Py to automatically define images. When not set to None, this should be set to a list of separators. (For example, `[ ' ', '_', '/' ]`.) Ren'Py will scan through the list of files on disk and in archives. When it finds a file ending with .png or .jpg, it will strip the extension, and break it apart at the separators, to create an image name. If the name consists of at least two components, and no image with that name already is defined, Ren'Py will define that image to refer to a filename.

For example, if your game directory contains:

- eileen_happy.png, Ren'Py will define the image "eileen happy".
- lucy/mad.png, Ren'Py will define the image "lucy mad".
- mary.png, Ren'Py will do nothing. (As the image does not have two components.)

(new in 6.10.0)

Variable: **config.automatic_images_strip** = []

A list of strings giving prefixes that are stripped out when defining automatic images. This can be used to remove directory names, when directories contain images. *(new in 6.10.2)*

Variable: **config.debug** = False

Enables some minor debugging functionality (mostly by turning some missing files into errors.) This should always be turned off in a release.

Variable: **config.debug_image_cache** = False

If True, Ren'Py will print the contents of the image cache to standard output (wherever that goes) whenever the contents of the image cache change.

Variable: **config.debug_sound** = False

Enables debugging of sound functionality. This disables the supression of errors when generating sound. However, if a sound card is missing or flawed, then such errors are normal, and enabling this may prevent Ren'Py from functioning normally. This should always be False in a released game.

Variable: **config.default_afm_time** = None

If not None, this sets the default auto-forward-mode timeout. If not None, then this is the time in seconds we should delay when showing 250 characters. 0 is special-cased to be infinite time, disabling auto-forward mode. *(new in 6.11.0)*

Variable: **config.default_afm_enable** = None

If not None, this should be a boolean that controls if auto-forward-mode is enabled by default. When it's False, auto-forwarding will not occur. Set this to False with caution, as the default Ren'Py UI does not provide a way of changing it's setting. (But one can use Preference action in a screen to create such a UI.) *(new in 6.11.0)*

Variable: **config.default_fullscreen** = None

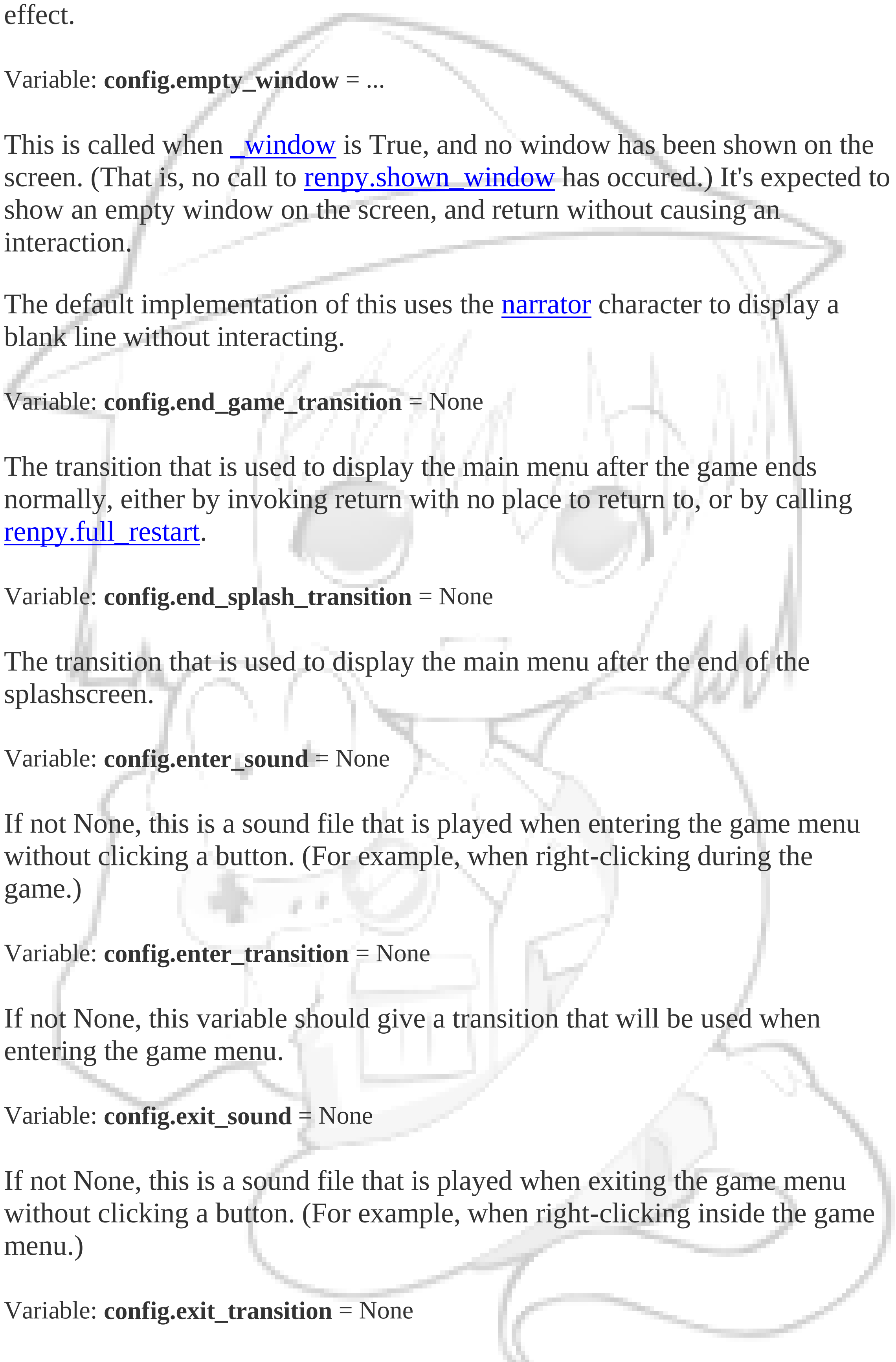
This sets the default value of the fullscreen preference. This should be True or False. If None, this is ignored, allowing other code to set the default value. (It's usually set to False in options.rpy.)

The file saves/persistent in the game directory must be deleted for this to take effect.

Variable: **config.default_text_cps** = None

If not None, this sets the default number of characters per second to show. 0 is special cased to mean an infinite number of characters per second. (It's usually set to 0 in options.rpy.)

The file saves/persistent in the game directory must be deleted for this to take effect.



Variable: **config.empty_window** = ...

This is called when _window is True, and no window has been shown on the screen. (That is, no call to renpy.shown_window has occured.) It's expected to show an empty window on the screen, and return without causing an interaction.

The default implementation of this uses the narrator character to display a blank line without interacting.

Variable: **config.end_game_transition** = None

The transition that is used to display the main menu after the game ends normally, either by invoking return with no place to return to, or by calling renpy.full_restart.

Variable: **config.end_splash_transition** = None

The transition that is used to display the main menu after the end of the splashscreen.

Variable: **config.enter_sound** = None

If not None, this is a sound file that is played when entering the game menu without clicking a button. (For example, when right-clicking during the game.)

Variable: **config.enter_transition** = None

If not None, this variable should give a transition that will be used when entering the game menu.

Variable: **config.exit_sound** = None

If not None, this is a sound file that is played when exiting the game menu without clicking a button. (For example, when right-clicking inside the game menu.)

Variable: **config.exit_transition** = None

If not None, this variable should give a transition that will be performed when exiting the game menu.

Variable: **config.font_replacement_map** = dict()

This is a map from (font, bold, italics) to (font, bold, italics), used to replace a font with one that's specialized as having bold and/or italics. For example, if you wanted to have everything using an italic version of "Vera.ttf" use "VeraIt.ttf" instead, you could write config.font_replacement_map["Vera.ttf", False, True] = ("VeraIt.ttf", False, False). Please note that these mappings only apply to specific variants of a font. In this case, requests for a bold italic version of vera will get a bold italic version of vera, rather than a bold version of the italic vera.

Variable: **config.framerate** = 100

If not None, this is the upper limit on the number of frames Ren'Py will attempt to display per second.

Variable: **config.game_main_transition** = None

The transition that is used to display the main menu after leaving the game menu. This is used when the load and preferences screens are invoked from the main menu, and it's also used when the user picks "Main Menu" from the game menu.

Variable: **config.game_menu** = [...]

This is used to customize the choices on the game menu. Please read Main and Game Menus for more details on the contents of this variable.

Variable: **config.game_menu_music** = None

If not None, a music file to play when at the game menu.

Variable: **config.has_autosave** = True

If true, the game will autosave. If false, no autosaving will occur.

Variable: **config.hyperlink_callback** = ...

A function that is called with the argument to a hyperlink when that hyperlink is clicked. Defaults to a function that inteprets the argument as a label, which is called in a new context when the hyperlink is clicked. The default function also dispatches arguments beginning with "http:" as urls, opening a web browser for them.

Variable: **config.hyperlink_focus** = None

If not None, a function that is called when a hyperlink gains or loses focus. When focus is gained, this function is called with a single argument, the argument of the hyperlink. When focus is lost, the function is called with None as its argument.

Variable: **config.hyperlink_styler** = ...

A function that is called to determine the style of hyperlinks. The function is called with a single argument, the argument of the hyperlink. It should return a style object, not a string. *(new in 6.6.2)*

Variable: **config.image_cache_size** = 8

This is used to set the size of the image cache, as a multiple of the screen size. This is mostly a suggestion, as if Ren'Py ever needs to display more than this amount of images at once, it will cache them all. But it will stop predictively loading images when the cache is filled beyond this, and it will try to drop images from the cache until it shrinks below this size. If this is too small, there will be more disk access as images are loaded and re-loaded. If this is too large, memory usage will be increased as images are kept in memory.

Variable: **config.main_game_transition** = None

The transition used when entering the game menu from the main menu, as is done when clicking "Load Game" or "Preferences".

Variable: **config.main_menu** = [("Start Game", "start", "True"), ("Continue Game", ...), ("Preferences", ...), ("Quit Game", ...)]

This is used to give the main menu that is shown to the user when the game first starts. It is a list of tuples, where the first element of each tuple is the title of the menu button, and the second element is a label that we jump to when that button is selected. (This jump exits the context in which the start menu executes.) The second element may also be a function, in which case it is called when the item is selected, in the menu context. The third element in each tuple is an expression, which is evaluated each time the main menu is displayed. If the expression evaluates to false, the menu choice is disabled.

For more information, see Main_and_Game_Menus

Variable: **config.main_menu_music** = None

If not None, a music file to play when at the main menu.

Variable: **config.menu_clear_layers** = []

A list of layer names (as strings) that are cleared when entering the game menu. *(new in 6.10.2)*

Variable: **config.menu_window_subtitle** = ""

The _window_subtitle variable is set to this value when entering the main or game menus. *(new in 6.7.1)*

Variable: **config.misssing_background** = "black"

This is the background that is used when config.developer is True and an undefined image is used in a scene statement. This is an image name, not a displayable. The image must be defined. *(new in 6.8.0)*

Variable: **config.mouse** = None

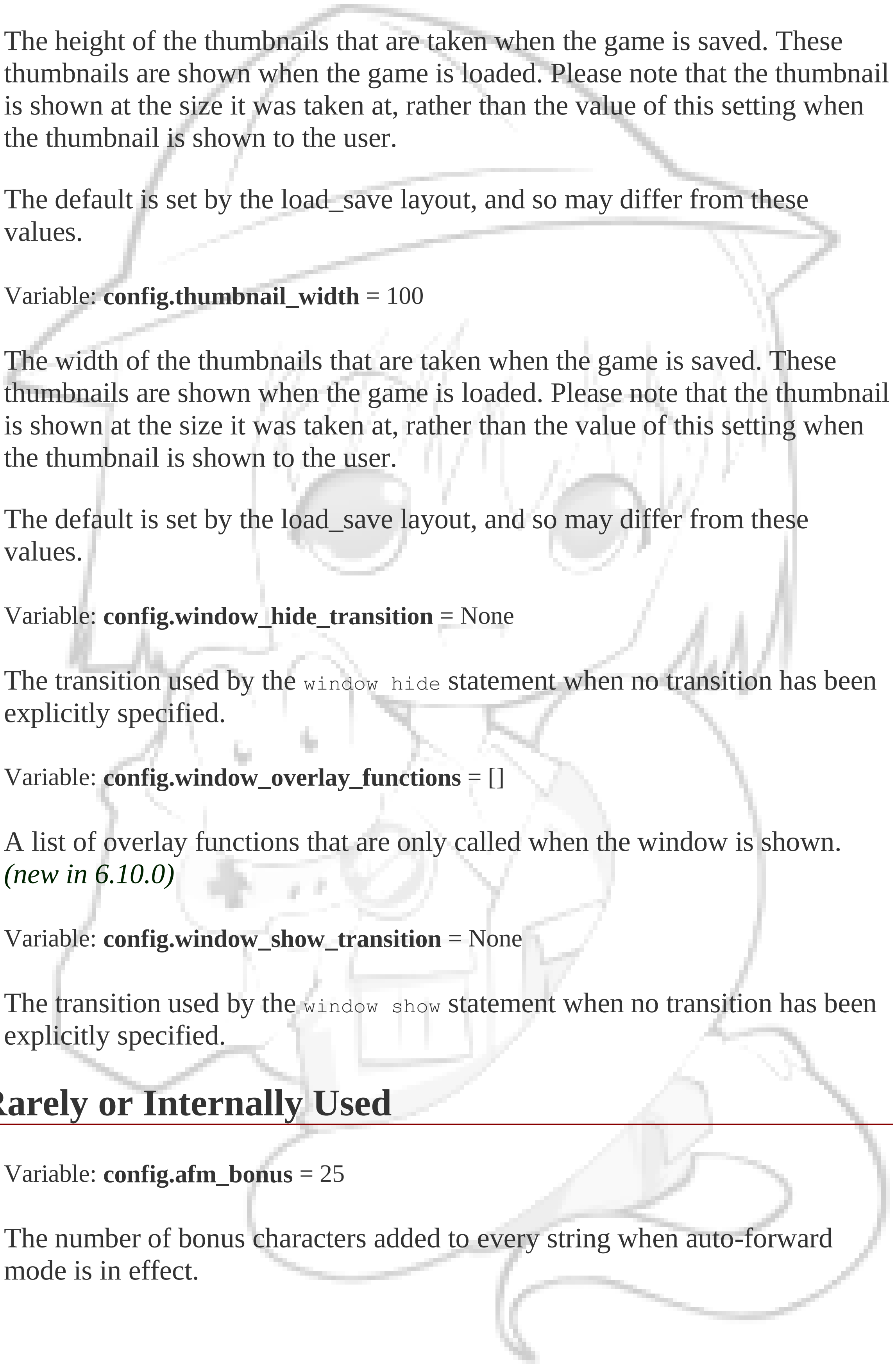
This variable controls the use of user-defined mouse cursors. If None, the system mouse is used, which is usually a black-and-white mouse cursor. Otherwise, this should be a dictionary giving the mouse animations for various mouse types. Keys used by the default library include "default", "say", "with", "menu", "prompt", "imagemap", "pause", "mainmenu", and "gamemenu". The "default" key should always be present. The values in the dictionary should be list of frames making up the mouse cursor. Each frame is an (image, x-offset, y-offset) tuple. Image is a filename of an image, while x- and y- offset are the offsets of the hotspot within the image. The frames are played back at 20hz, and the animation loops after all frames have been shown. Please note that to show a frame more than once, it must be repeated in a frame list. That is implemented efficiently.

Variable: **config.nvl_adv_transition** = None

A transition that is used when showing ADV-mode text directly after NVL-mode text.

Variable: **config.overlay_functions** = [ ]

A list of functions. When called, each function is expected to return a list of displayables, which are added to the overlay list. See the section on overlays for more.



Variable: **config.thumbnail_height** = 75

The height of the thumbnails that are taken when the game is saved. These thumbnails are shown when the game is loaded. Please note that the thumbnail is shown at the size it was taken at, rather than the value of this setting when the thumbnail is shown to the user.

The default is set by the load_save layout, and so may differ from these values.

Variable: **config.thumbnail_width** = 100

The width of the thumbnails that are taken when the game is saved. These thumbnails are shown when the game is loaded. Please note that the thumbnail is shown at the size it was taken at, rather than the value of this setting when the thumbnail is shown to the user.

The default is set by the load_save layout, and so may differ from these values.

Variable: **config.window_hide_transition** = None

The transition used by the `window hide` statement when no transition has been explicitly specified.

Variable: **config.window_overlay_functions** = []

A list of overlay functions that are only called when the window is shown. *(new in 6.10.0)*

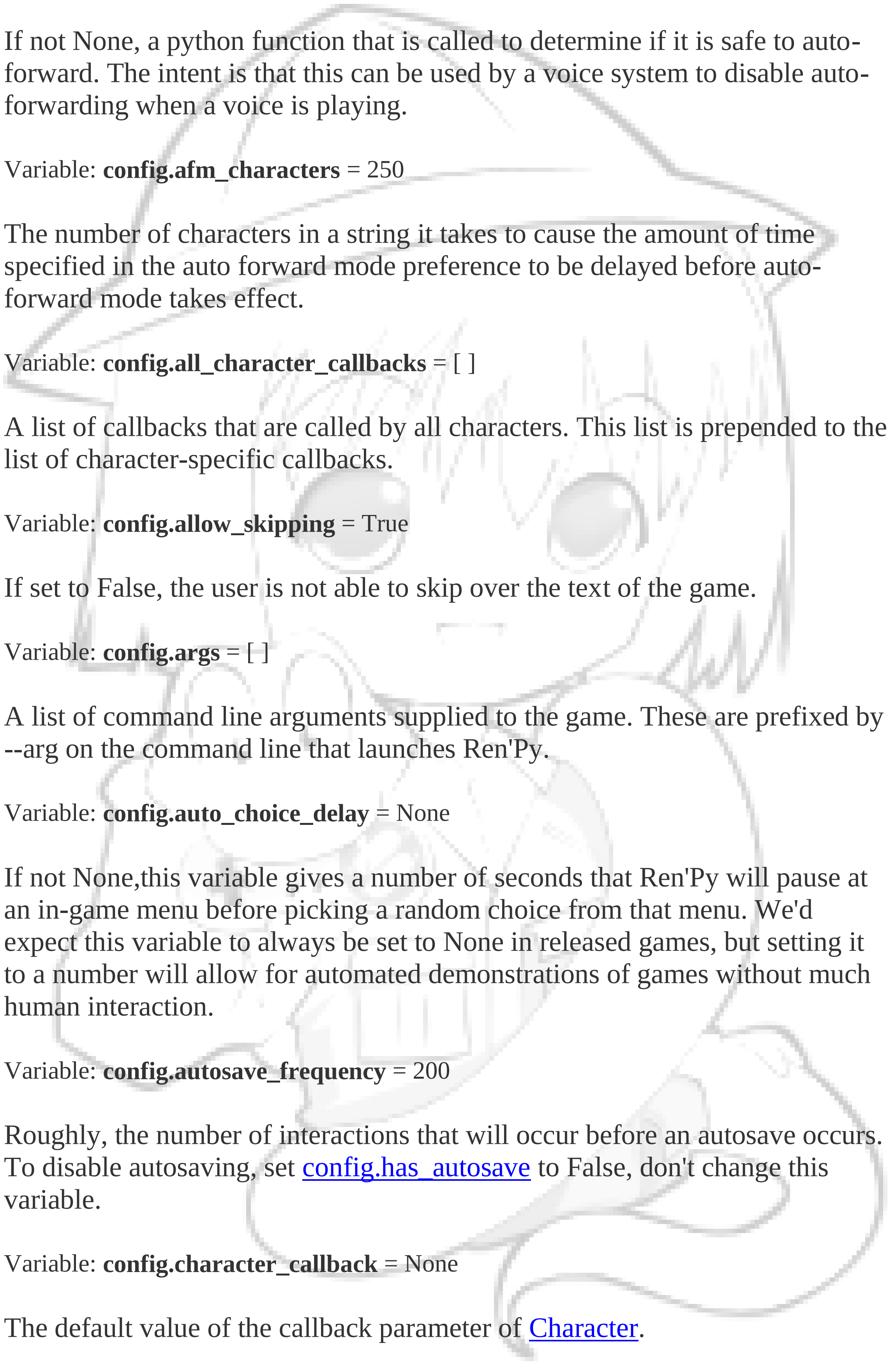
Variable: **config.window_show_transition** = None

The transition used by the `window show` statement when no transition has been explicitly specified.

Rarely or Internally Used

Variable: **config.afm_bonus** = 25

The number of bonus characters added to every string when auto-forward mode is in effect.



Variable: **config.afm_callback** = None

If not None, a python function that is called to determine if it is safe to auto-forward. The intent is that this can be used by a voice system to disable auto-forwarding when a voice is playing.

Variable: **config.afm_characters** = 250

The number of characters in a string it takes to cause the amount of time specified in the auto forward mode preference to be delayed before auto-forward mode takes effect.

Variable: **config.all_character_callbacks** = [ ]

A list of callbacks that are called by all characters. This list is prepended to the list of character-specific callbacks.

Variable: **config.allow_skipping** = True

If set to False, the user is not able to skip over the text of the game.

Variable: **config.args** = [ ]

A list of command line arguments supplied to the game. These are prefixed by --arg on the command line that launches Ren'Py.

Variable: **config.auto_choice_delay** = None

If not None,this variable gives a number of seconds that Ren'Py will pause at an in-game menu before picking a random choice from that menu. We'd expect this variable to always be set to None in released games, but setting it to a number will allow for automated demonstrations of games without much human interaction.

Variable: **config.autosave_frequency** = 200

Roughly, the number of interactions that will occur before an autosave occurs. To disable autosaving, set config.has_autosave to False, don't change this variable.

Variable: **config.character_callback** = None

The default value of the callback parameter of Character.

Variable: **config.clear_layers** = []

A list of names of layers to clear when entering the main and game menus. *(new in 6.7.0)*

Variable: **config.editor** = None

If not None, this is expected to be a command line for an editor that is invoked when the launch_editor (normally shift-E) key is pressed. The following substitutions make sense here:

- %(filename)s - The filename of the most interesting file to be edited. This is the file that should be shown to the user.
- %(line)d - The line number of the most interesting file to show to the user.
- %(otherfiles)s - Other, less-interesting files to show to the user.
- %(allfiles)s - All the files.

filename, *otherfiles*, and *allfiles* have shell-relevant characters escaped with backslashes. *otherfiles* and *allfiles* separate files with config.editor_file_separator (by default '" "', a double-quote, a space, and a quote). Since all filenames should be enclosed in double-quotes, this means that *otherfiles* and *allfiles* will create several quoted files. If two double-quotes occur in a row the string, then they are both removed. (This allows an empty allfiles to be used.) *(changed in 6.7.1)*

A reasonable example is:

```
init python:
    config.editor = 'myeditor "%(filename)s" +line:%(line)d
"%(otherfiles)s"'
```

This defaults to the value of the RENPY_EDITOR environment variable. If not defined by that variable or user code, this is set automatically by the Ren'Py launcher.

Variable: **config.editor_file_separator** = '" "'

The separator used between filenames when lists of files are provided to the editor.

Variable: **config.editor_transient** = None

If not None, this is expected to be a command line for an editor that is invoked on transient files, such as lint results, parse errors, and tracebacks. Substitutions are as for config.editor.

This defaults to the value of the RENPY_EDITOR_TRANSIENT environment variable. If not defined by that variable or user code, this is set automatically by the Ren'Py launcher.

Variable: **config.fade_music** = 0.0

This is the amount of time in seconds to spend fading the old track out before a new music track starts. This should probably be fairly short, so the wrong music doesn't play for too long.

Variable: **config.fast_skipping** = False

Set this to True to allow fast skipping outside of developer mode.

Variable: **config.file_open_callback** = None

If not None, this is a function that is called with the file name when a file needs to be opened. It should return a file-like object, or None to load the file using the usual Ren'Py mechanisms. Your file-like object must implement at least the read, seek, tell, and close methods. *(new in 6.10.0)*

Variable: **config.focus_crossrange_penalty** = 1024

This is the amount of penalty to apply to moves perpendicular to the selected direction of motion, when moving focus with the keyboard.

Variable: **config.hard_rollback_limit** = 100

This is the number of steps that Ren'Py will let the user interactively rollback. Set this to 0 to disable rollback entirely, although we don't recommend that, as rollback is useful to let the user see text he skipped by mistake.

Variable: **config.hide** = renpy.hide

A function that is called when the hide statement is executed. This should take the same arguments as renpy.hide.

Variable: **config.implicit_with_none** = True

If True, then by default the equivalent of a "with None" statement will be performed after interactions caused by dialogue (through the Character object), menus (and renpy.display_menu), renpy.input, and renpy.imagemap. All of the above functions take a *with_none* parameter, which will override this variable.

Variable: **config.interact_callbacks** = ...

A list of functions that are called (without any arguments) when an interaction is started or restarted.

Variable: **config.joystick** = True

Governs if joystick support is enabled. If False, Joystick detection is disabled.

Variable: **config.keymap** = dict(...)

This variable contains a keymap giving the keys and mouse buttons assigned to each possible operation. Please see the section on Keymaps for more information.

Variable: **config.label_callback** = None

If not None, this is a function that is called whenever a label is reached. It is called with two parameters. The first is the name of the label. The second is true if the label was reached through jumping, calling, or creating a new context, and false otherwise. *(new in 6.9.0)*

Variable: **config.label_overrides** = {
}

This variable gives a way of causing jumps and calls of labels in Ren'Py code to be redirected to other labels. For example, if you add a mapping from "start" to "mystart", all jumps and calls to "start" will go to "mystart" instead.

Variable: **config.layer_clipping** = {
}

Controls layer clipping. This is a map from layer names to (x, y, height, width) tuples, where x and y are the coordinates of the upper-left corner of the layer, with height and width giving the layer size.

If a layer is not mentioned in config.layer_clipping, then it is assumed to take up the full screen.

Variable: **config.layers** = ['master', 'transient', 'overlay']

This variable gives a list of all of the layers that Ren'Py knows about, in the order that they will be displayed to the screen. (The lowest layer is the first entry in the list.) Ren'Py uses the layers "master", "transient", and "overlay" internally, so they should always be in this list.

Variable: **config.lint_hooks** = ...

This is a list of functions that are called, with no arguments, when lint is run. The functions are expected to check the script data for errors, and print any they find to standard output (using the python print statement is fine in this case).

Variable: **config.load_before_transition** = True

If True, the start of an interaction will be delayed until all images used by that interaction have loaded. (Yeah, it's a lousy name.)

Variable: **config.log** = None

If not None, this is expected to be a filename. Much of the text shown to the user by say or menu statements will be logged to this file.

Variable: **config.missing_image_callback** = None

If not None, this function is called when an attempt to load an image fails. It may return None, or it may return an image manipulator. If an image manipulator is returned, that image manipulator is loaded in the place of the missing image.

Variable: **config.mouse_hide_time** = 30

The mouse is hidden after this number of seconds has elapsed without any mouse input. This should be set to longer then the expected time it will take to read a single screen, so mouse users will not experience the mouse appearing then disappearing between clicks.

Variable: **config.overlay_during_with** = True

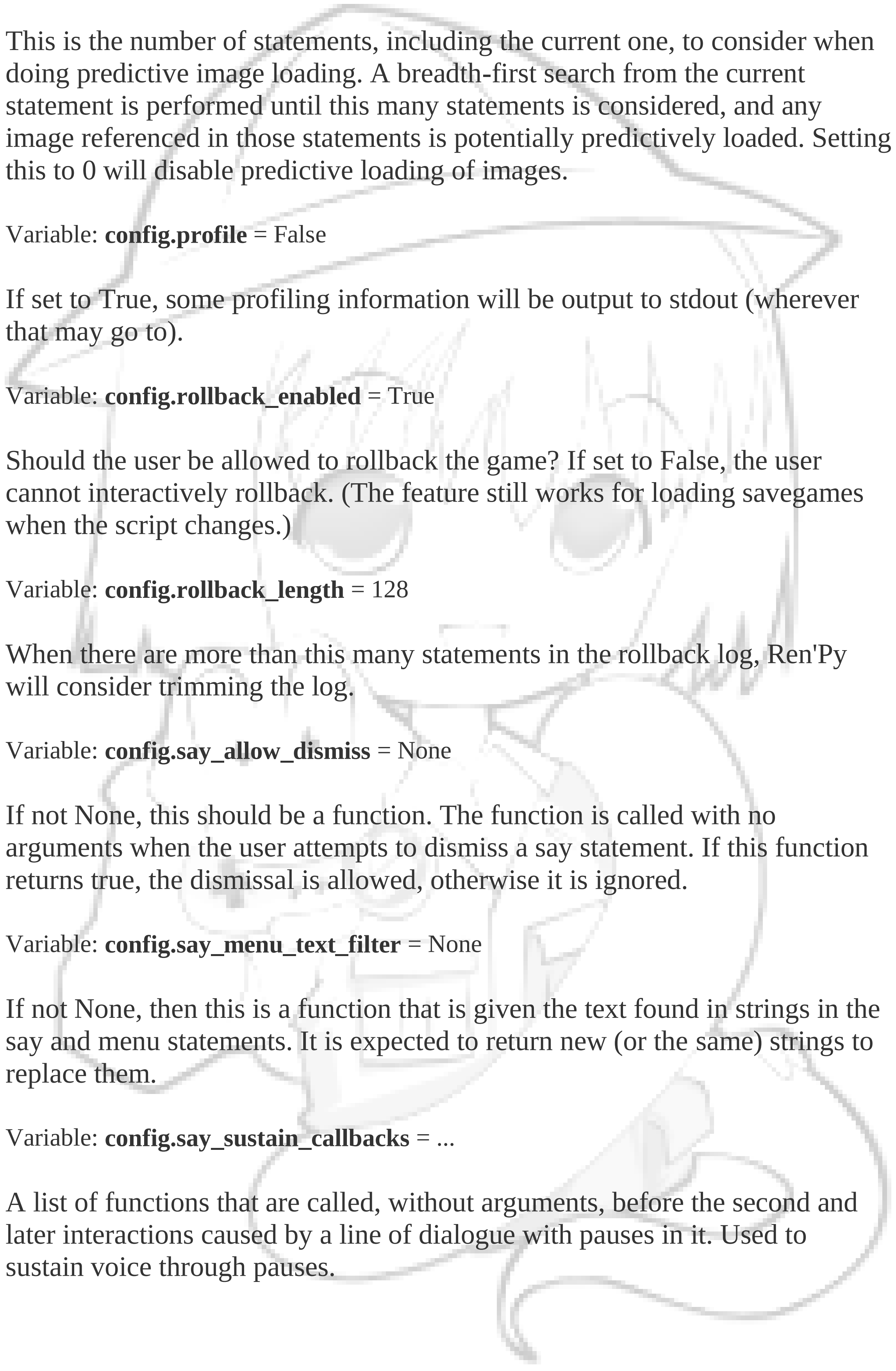
True if we want overlays to be shown during with statements, or False if we'd prefer that they be hidden during the with statements.

Variable: **config.overlay_layers** = ['overlay']

This is a list of all of the overlay layers. Overlay layers are cleared before the overlay functions are called. "overlay" should always be in this list.

Variable: **config.periodic_callback** = None

If not None, this should be a function. The function is called, with no arguments, at around 20hz.



Variable: **config.predict_statements** = 10

This is the number of statements, including the current one, to consider when doing predictive image loading. A breadth-first search from the current statement is performed until this many statements is considered, and any image referenced in those statements is potentially predictively loaded. Setting this to 0 will disable predictive loading of images.

Variable: **config.profile** = False

If set to True, some profiling information will be output to stdout (wherever that may go to).

Variable: **config.rollback_enabled** = True

Should the user be allowed to rollback the game? If set to False, the user cannot interactively rollback. (The feature still works for loading savegames when the script changes.)

Variable: **config.rollback_length** = 128

When there are more than this many statements in the rollback log, Ren'Py will consider trimming the log.

Variable: **config.say_allow_dismiss** = None

If not None, this should be a function. The function is called with no arguments when the user attempts to dismiss a say statement. If this function returns true, the dismissal is allowed, otherwise it is ignored.

Variable: **config.say_menu_text_filter** = None

If not None, then this is a function that is given the text found in strings in the say and menu statements. It is expected to return new (or the same) strings to replace them.

Variable: **config.say_sustain_callbacks** = ...

A list of functions that are called, without arguments, before the second and later interactions caused by a line of dialogue with pauses in it. Used to sustain voice through pauses.

Variable: **config.savedir** = ...

The complete path to the directory in which the game is saved. This should only be set in a python early block. See also config.save_directory, which generates the default value for this if it is not set during a python early block.

Variable: **config.scene** = renpy.scene

A function that's used in place of renpy.scene by the scene statement. Note that this is used to clear the screen, and config.show is used to show a new image. This should have the same signature as renpy.scene.

Variable: **config.script_version** = None

If not None, this is interpreted as a script version. The library will use this script version to enable some compatibility features, if necessary. If None, we assume this is a latest-version script.

This is normally set in a file added by the Ren'Py launcher when distributions are built.

Variable: **config.searchpath** = ['common', 'game']

A list of directories that are searched for images, music, archives, and other media, but not scripts. This is initialized to a list containing "common" and the name of the game directory, which changes depending on the name of the exe file. This variable is not used to load scripts, as scripts will be loaded before it can be set.

Variable: **config.show** = renpy.show

A function that is used in place of renpy.show by the show and scene statements. This should have the same signature as renpy.show.

Variable: **config.skip_delay** = 75

The amount of time that dialogue will be shown for, when skipping statements using ctrl, in milliseconds. (Although it's nowhere near that precise in practice.)

Variable: **config.skip_indicator** = True

If True, the library will display a skip indicator when skipping through the script.

Variable: **config.sound** = True

If True, sound works. If False, the sound/mixer subsystem is completely disabled.

Variable: **config.sound_sample_rate** = 44100

The sample rate that the sound card will be run at. If all of your wav files are of a lower rate, changing this to that rate may make things more efficent.

Variable: **config.start_interact_callbacks** = ...

A list of functions that are called (without any arguments) when an interaction is started. These callbacks are not called when an interaction is restarted.

Variable: **config.sticky_positions** = False

Sticky positions causes the at clause of a show or scene statement to be remembered for as long as that character is on the screen. Future show statements without at clauses will be given the remembered at clause. This lets us easily change the character's expression without changing their position on the screen. Sticky positions does not interact well with multiple layers, when they are used with the show, hide, and scene statements via an onlayer clause.

Variable: **config.top_layers** = [ ]

This is a list of names of layers that are displayed above all other layers, and do not participate in a transition that is applied to all layers. If a layer name is listed here, it should not be listed in config.layers.

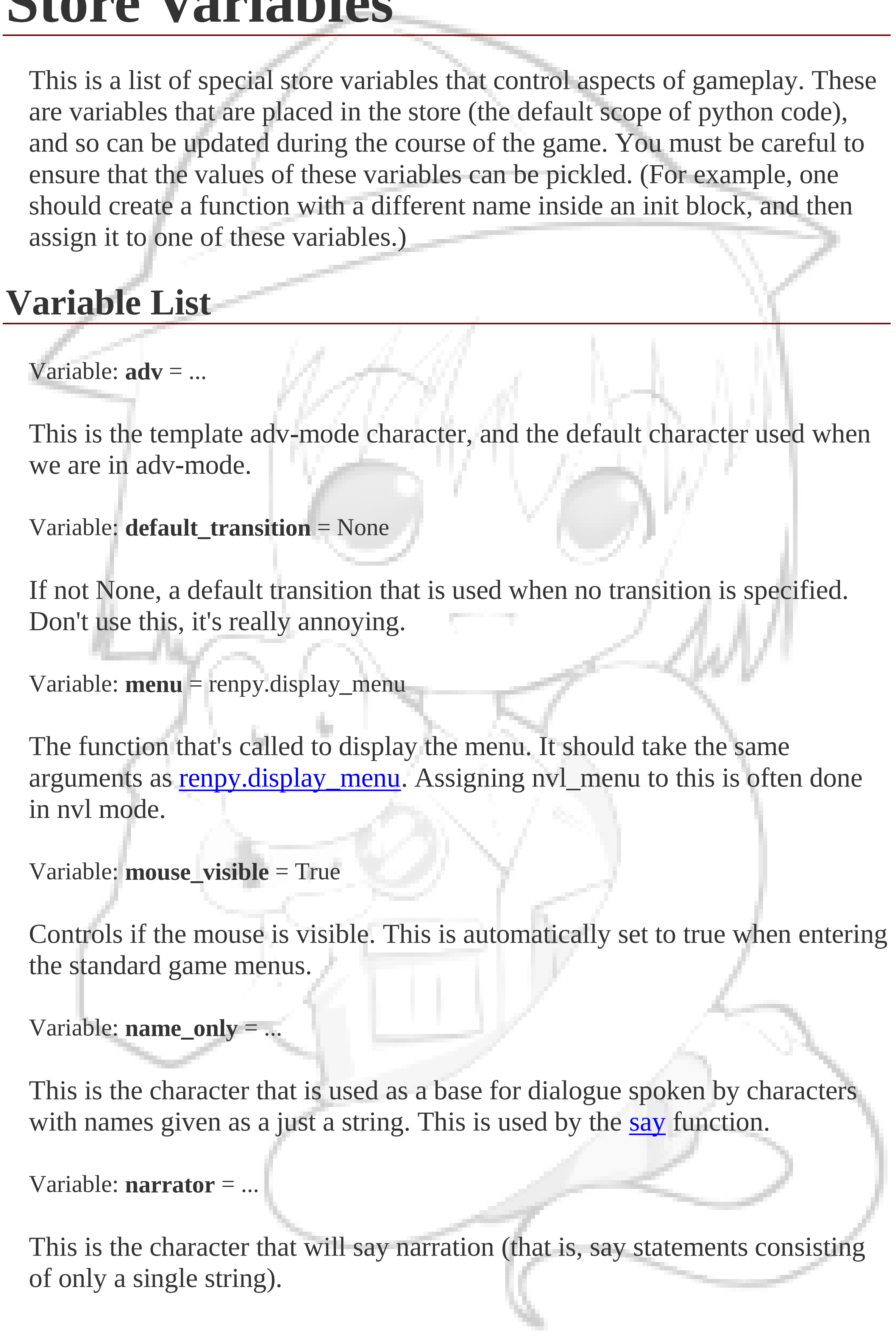
Variable: **config.transient_layers** = ['transient']

This variable gives a list of all of the transient layers. Transient layers are layers that are cleared after each interaction. "transient" should always be in this list.

Variable: **config.with_callback** = None

If not None, this should be a function that is called when a with statement occurs. This function can be responsible for putting up transient things on the screen during the transition. The function is called with a single argument, which is the transition that is occuring. It is expected to return a transition, which may or may not be the transition supplied as its argument.

Store Variables



This is a list of special store variables that control aspects of gameplay. These are variables that are placed in the store (the default scope of python code), and so can be updated during the course of the game. You must be careful to ensure that the values of these variables can be pickled. (For example, one should create a function with a different name inside an init block, and then assign it to one of these variables.)

Variable List

Variable: **adv** = ...

This is the template adv-mode character, and the default character used when we are in adv-mode.

Variable: **default_transition** = None

If not None, a default transition that is used when no transition is specified. Don't use this, it's really annoying.

Variable: **menu** = renpy.display_menu

The function that's called to display the menu. It should take the same arguments as renpy.display_menu. Assigning nvl_menu to this is often done in nvl mode.

Variable: **mouse_visible** = True

Controls if the mouse is visible. This is automatically set to true when entering the standard game menus.

Variable: **name_only** = ...

This is the character that is used as a base for dialogue spoken by characters with names given as a just a string. This is used by the say function.

Variable: **narrator** = ...

This is the character that will say narration (that is, say statements consisting of only a single string).

Variable: **nvl** = ...

This is the default kind of character for NVL-mode characters.

Variable: **predict_menu** = ...

This is called with no arguments, and is expected to return a list of displayables used by the menu function.

Variable: **predict_say** = ...

This is used to predict the images used by say. It's called with the same two arguments, and is expected to return a list of displayables used.

Variable: **_rollback** = True

Controls if rollback is allowed.

Variable: **say** = ...

A function that is called when the say statement is called with a string for a name. The default behavior of this function creates a copy of the name_only object with the supplied name, and then uses that to display the dialogue. The function takes two arguments, given the name of the character and the line of dialogue.

Variable: **save_name** = ""

This is a name that's used when data is saved.

Variable: **suppress_overlay** = False

When true, overlay functions will not be called and the overlay will not be displayed. Defaults to true, but is set to false upon entering the out-of-game menus.

Variable: **_window** = False

This controls if the window is shown automatically when a character is not speaking. Normally, this is set by the `window show` and `window hide` statements.

Variable: **_window_subtitle** =

This is appended to config.window_title to produce the caption for the game window. This is automatically set to config.menu_window_subtitle when entering the game menu. *(new in 6.7.1)*

Variables and functions specific to menu customization are documented at Customizing the Main and Game Menus.

Overlays

Overlays are used to display information above the scene currently displayed. The overlay is regenerated each time an interaction with the user begins, making it suitable for displaying to the user things like statistics or dates. The overlay is generally displayed whenever transient things (like dialogue, thoughts and menus) are.

Overlays are set up by adding to the config.overlay_functions list a Python function which, when called, uses the ui functions to add widgets to the screen. By default, such widgets are added to the 'overlay' layer, but a call to ui.layer can change the layer to any of the layers listed in config.overlay_layers. These functions are called for each interaction, which allows the overlay to change to reflect the status of game variables. If a variable affecting the overlay changes during an interaction, renpy.restart_interaction should be called to regenerate the overlay.

As an example, take the following code fragement. When added to a program, this displays a date image in the upper-right corner of the screen (as is done in Kanon). The image shown is based on the variable date. If date is None, then no date is shown. Otherwise, a png file beginning with the value of date is shown.

```
init:
    $ date = "mar25"

    python hide:
        def date_overlay():
            if date:
                ui.image(date + ".png",
                         xpos=1.0, xanchor="right",
                         ypos=0.0, yanchor="top")

        config.overlay_functions.append(date_overlay)
```

Like all config variables, config.overlay_functions should only be changed in an init block. If you need to toggle an overlay on and off, then the overlay function should be conditioned on some normal variable. This is done in the example above when date is None.

Saving, Loading, and Rollback

Ren'Py supports saving a game to disk, loading that game back in from disk, and rolling back the game to a previous state. As these actions share infrastructure, we will discuss them all in this section, paying attention to what is required to support loading of a saved game when a script changes between saving and loading.

Please note that save-game compatibility is not guaranteed between releases of Ren'Py, even between older and newer releases of Ren'Py. It may be necessary to take this into account when releasing a game with an updated version of Ren'Py.

Saving, Loading, and Rollback can all be seen as operations that affect the game state. Saving is an operation that persists the game state out to disk, Loading consists of restoring the game state from disk, and Rollback reverts the game state to what it was at a previous point in time. This game state consists of two parts: The user state consists of variables and objects that have been changed by the user, while the internal state consists of information used by Ren'Py directly.

Internal State. Specifically, the internal state of Ren'Py consists of the following:

- The name of the currently-executing statement.
- The name of each statement on the return stack.
- The layers containing images that are shown on the screen.
- The music that Ren'Py has been requested to play.
- The time the game has been played for.

These items are automatically saved, loaded, and rolled back when the appropriate commands are invoked by the user. You do not need do anything for this state to be handled, and there is no way to prevent these things from being saved.

User State. The other kind of state that Ren'Py can save and restore is user state. User state consists of all variables that have been changed after the end of the init phase, and all objects transitively reachable from such variables.

It's important to clarify what it means for a variable to be changed. In Python and Ren'Py, variables reference objects. A variable changes when it is updated to reference a new object. A variable does not change when the object it references changes.

For example, in the following code:

```
init:
      $ a = 0
      $ state = object()

$ a = 1
$ b = [ ]
$ state.love_love_points = a + 1
```

In this example, the variables `a` and `b` are changed by the code outside of the init block (assuming that code executes), while state is not changed. In the code outside of the init block, `a` is assigned a different integer object, while `b` is assigned a new empty list. While a field on the object `state` refers to has changed, `state` itself still refers to the same object, and not considered to have changed. (Hence, state and its associated object are not considered part of user state.)

User state is gathered by first finding all variables that have changed outside of init blocks. Ren'Py then finds all objects reachable from one of those variables through some combination of field access, iteration, or iteration over items (as in a dictionary). This combination of variable and object values comprises the user state.

It's important to ensure that every object in the user state can be pickled (serialized) by the python pickle module. Most python constructs can be pickled, including booleans, integers, floating-point numbers, strings, lists, tuples, dictionaries, and most objects. You can also refer to your own classes and functions, provided that they are defined in a python block (not a python hide block) inside an init block, and always exist with the same name in later versions of the script. There are some python objects that cannot be pickled, such as files, iterators, and generators. These objects should not be used outside of python hide blocks.

While these rules may seem to be complex, it's hoped that in practice they can be reduced to a simple heuristic: Any variable changed outside of an init block, and any object reachable from such a variable, will be saved, loaded, and rolled back properly.

State that isn't Saved. There is some state kept by Ren'Py that is not part of the interpreter state. This includes:

- Mappings of image names to displayables, created by the image statement.
- Configuration variables (config.varname).
- Styles (style.stylename).

As a result, config variables and styles should be set up in init blocks, and then left alone for the rest of the game. As the image statement can only be run from inside an init block, it is impossible to set up a mapping from image name to displayable outside of an image block. To ensure compatibility between script version, once an image name or style is present in a released version of the game, it should be present in all future released versions of the game.

Details. As the game is played, Ren'Py logs all changes to user and interpreter state. When the game is saved, it writes this log out to disk, alongside the current state. When the game is loaded back in, the variables are reset to what the were when the init code in the current version of the script finished running. The saved user state is then merged with this, with saved user state overriding any variable that was also assigned in the init code. Finally, a rollback is triggered.

The rollback that is triggered on load ends when it can find a statement that has the same name as it had when it was encountered in the log. When the script hasn't changed, all statements have the same name, so the effect of the rollback is to bring control back to the start of the statement that was executing when the user saved. When the script has changed, however, the only statements that retain their names are statements that have an explicit name specified. (These statements are labels, menu statements with an explicit name, and call ... from ... statements.) The game will rollback to the start of the most recent statement that exists in both the old and new games.

When a rollback occurs, both user and interpreter state are restored to what they were when the statement that is being rolled back to began executing. The statement is then executed again, and play continues normally.

Please note that we can only roll back the current statement, and not the return sites listed on the return stack. If the name of a return site changes, we will not be able to return from a procedure call, and the script will crash. If a return site has an explicit name, however, that name is returned to even if the script change. Because of this, it's important that every call site in a released game have a from clause associated with it, giving a name for the return site that can appear in both old scripts.

Finally, if allowed, rollback can be invoked explicitly by the user. When such a rollback occurs, we first look for a previous statement that is a checkpoint (checkpoints are say and menu statements, as well as python blocks that called renpy.checkpoint). Once a checkpoint is found, we look for a statement which has a name that exists in the current script (this is normally the same statement). We then rollback to that statement and begin executing again.

What this means is that when a rollback occurs, the game is usually reverted to the start of the say or menu statement that executed before the currently executing statement.

There is one variable that controls the behavior of loading and saving:

Variable: **save_name** =

This is a name that will be associated with save files. It's expected that the game will update this on a regular basis with the name of the section of the script that is currently executing. When a save file is shown to the user, this name will be shown with it. Alternatively, never change this and no save name will be shown.

Running code after a load. If the after_load label exists, then it is called after a load has occured. It is responsible for updating variables that may have changed between script versions, and performing any other action that may need to be taken in response to the load. When a return statement is executed, execution is transferred to the point where the game was saved.

Please note that when calling after_load, the return stack is not safe, and hence the user should not be allowed to save before control returns from after_load. To be safe, this means the after_load code should not be interactive.

Load/Save Functions

The following functions are useful for code that wants to manage loading and saving on its own.

Function: **renpy.can_load** (filename):

Returns true if filename can be loaded (by the load/save mechanism), or false otherwise. This is a very basic check, mostly ensuring that the file exists.

Function: **renpy.list_saved_games** (regexp=r'.'):

This scans the savegames that we know about and returns information about them. It returns a list of tuples, where each tuple represents one savegame and consists of:

- The filename of the save.
- The extra_info that was passed to renpy.save.
- A displayable, the screenshot used to show the game.
- The time the game was saved at, seconds since 1/1/1970 UTC.

The regexp matches at the start of the filename, and filters the list.

Function: **renpy.load** (filename):

Causes the load/save mechanism to load *filename*. This function never returns.

Function: **renpy.rename_save** (old, new):

Renames the save *old* to the save *new*, overwriting any existing save named *new*.

Function: **renpy.save** (filename, extra_info=""):

Saves the current game in *filename*.

filename - A string, giving the filename to save in. Note that this does not correspond to any particular file on disk, but should be treated as an arbitrary game identifier. Normal save slots are base-10 representations of integers, while other names will not show up in the file-picker.

extra_info - Additional information saved with the game. This is usually the value of save_name.

renpy.take_screenshot must be called before renpy.save, except when inside the game menu, where it has been automatically called.

Function: **renpy.scan_saved_game** (name):

This scans a single savegame, and returns information about it. If a savegame with the filename *name* does not exist, this function returns None. Otherwise, it returns a triple consisting of:

- The extra_info that was passed to renpy.save.
- A displayable, the screenshot used to show the game.
- The time the game was saved at, seconds since 1/1/1970 UTC.

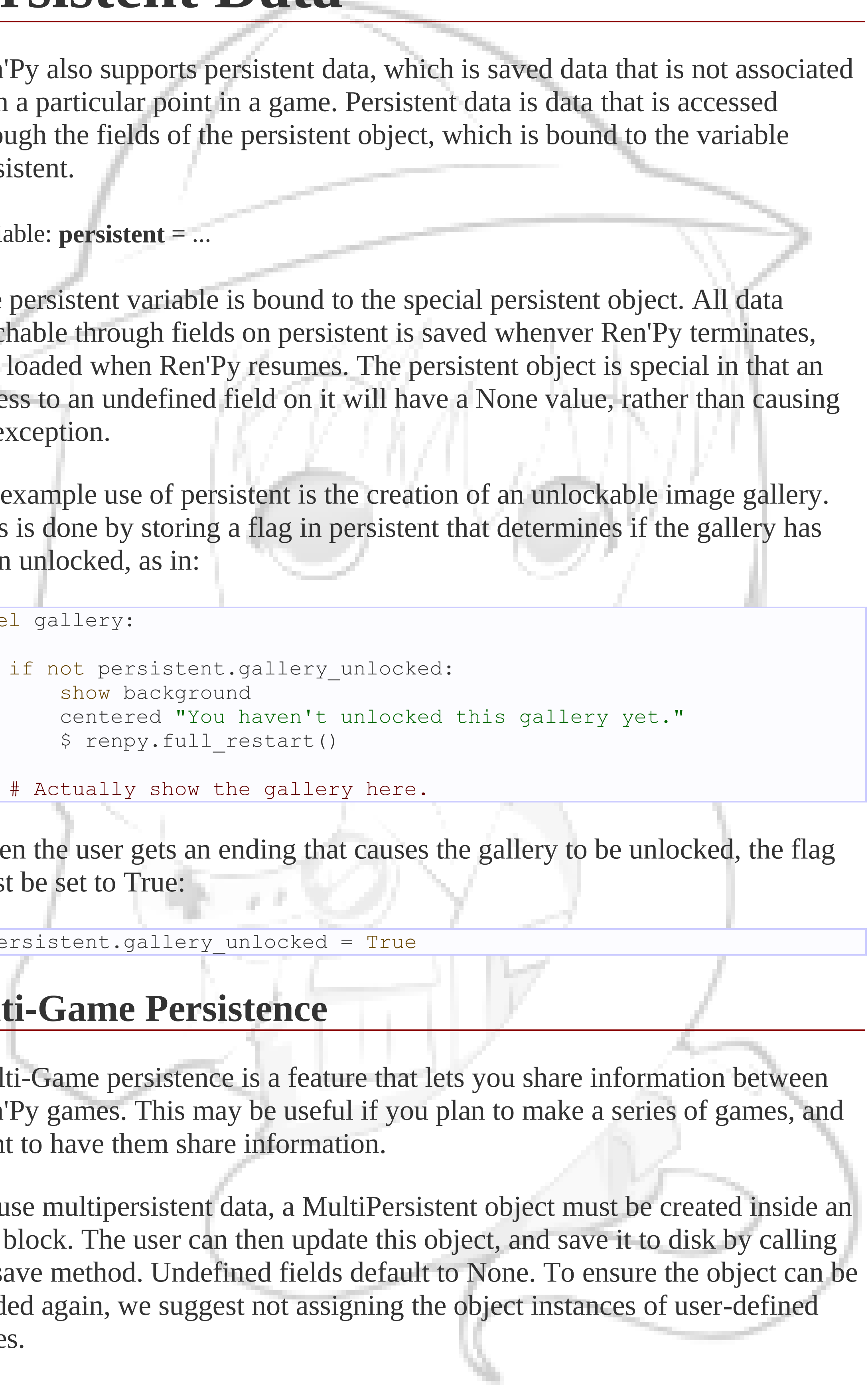
Function: **renpy.take_screenshot** (size=None):

This function should be called before renpy.save, to take the screenshot that will be used as part of a savegame. This is called automatically when entering the game menu, but it may make sense to call it manually if you force a save.

size -- The size of the screenshot. If None, defaults to (config.thumbnail_width, config.thumbnail_height).

Function: **renpy.unlink_save** (filename):

Unlinks (deletes) the save file with the given name. Throws an exception if the file can't be deleted.



Persistent Data

Ren'Py also supports persistent data, which is saved data that is not associated with a particular point in a game. Persistent data is data that is accessed through the fields of the persistent object, which is bound to the variable persistent.

Variable: **persistent** = ...

The persistent variable is bound to the special persistent object. All data reachable through fields on persistent is saved whenver Ren'Py terminates, and loaded when Ren'Py resumes. The persistent object is special in that an access to an undefined field on it will have a None value, rather than causing an exception.

An example use of persistent is the creation of an unlockable image gallery. This is done by storing a flag in persistent that determines if the gallery has been unlocked, as in:

```
label gallery:

    if not persistent.gallery_unlocked:
        show background
        centered "You haven't unlocked this gallery yet."
        $ renpy.full_restart()

    # Actually show the gallery here.
```

When the user gets an ending that causes the gallery to be unlocked, the flag must be set to True:

```
$ persistent.gallery_unlocked = True
```

Multi-Game Persistence

Multi-Game persistence is a feature that lets you share information between Ren'Py games. This may be useful if you plan to make a series of games, and want to have them share information.

To use multipersistent data, a MultiPersistent object must be created inside an init block. The user can then update this object, and save it to disk by calling its save method. Undefined fields default to None. To ensure the object can be loaded again, we suggest not assigning the object instances of user-defined types.

Function: **MultiPersistent** (key):

Creates a new MultiPersistent object. This should only be called inside an init block, and it returns a new MultiPersistent object corresponding to key.

key a key that is used to access the multipersistent data. To prevent conflicts, we suggest that the key be suffixed by a domain you own... so if you owned renpy.org, you could use the key "demo.renpy.org".

Method: **MultiPersistent.save** ():

Saves a MultiPersistent object to disk. Changes are not persisted until this method is called.

Example. Part 1 of a multi-part game:

```
init:
    $ mp = MultiPersistent("demo.renpy.org")

label start:

    # ...

    # Record the fact that the user beat part 1.

    $ mp.beat_part_1 = True
    $ mp.save()

    e "You beat part 1. See you in part 2!"
```

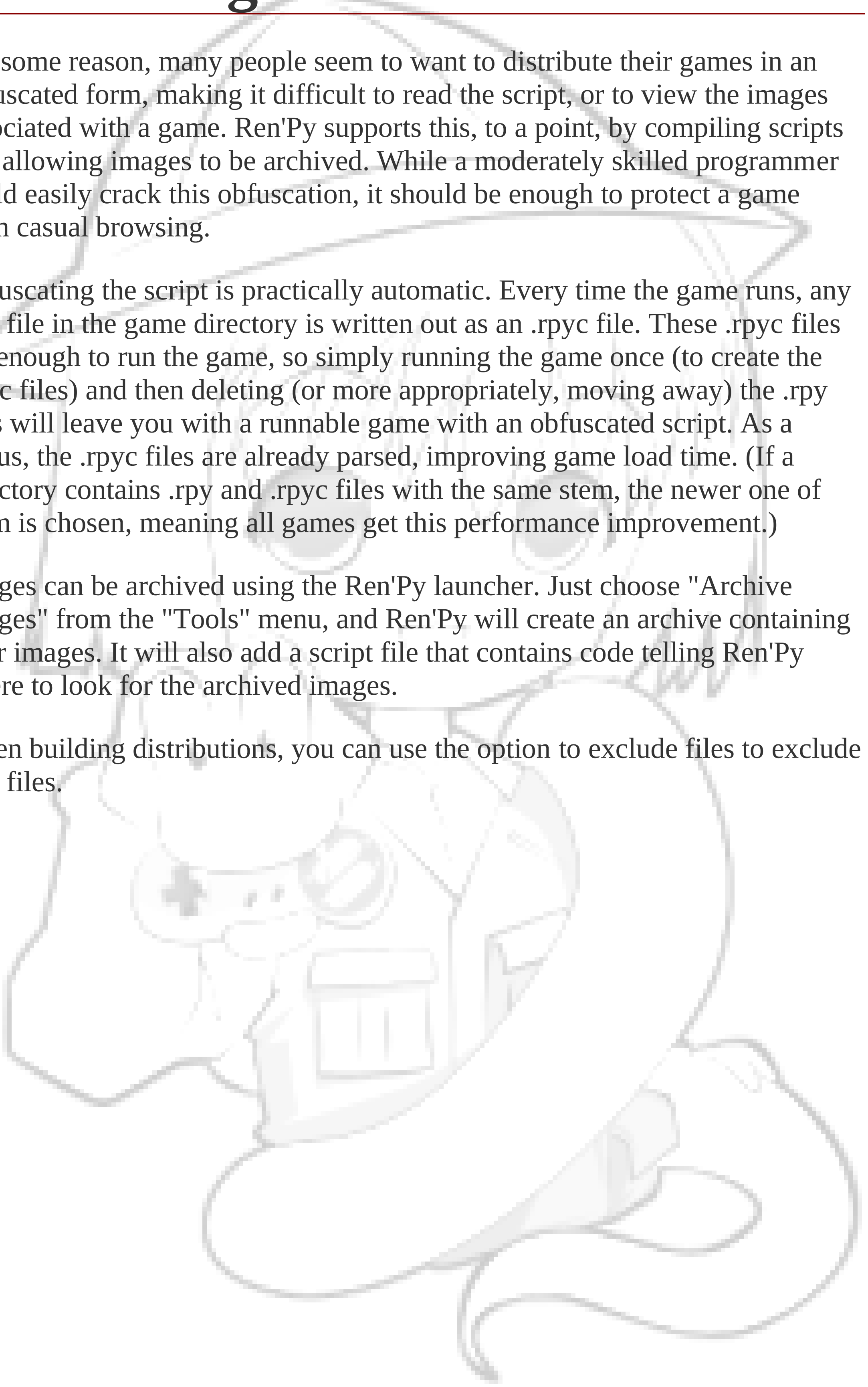
And the code for part 2:

```
init:
    $ mp = MultiPersistent("demo.renpy.org")

label start:

    if mp.beat_part_1:
         e "I see you've beaten part 1, so welcome back!"
    else:
         e "Hmm, you haven't played part 1, why not try it first?"
```

Data Location. The place where the data is stored varies based on the operating system being used:

- Windows: %APPDATA%/RenPy/persistent
- Mac OS X: ~/Library/RenPy/persistent
- Linux/Other: ~/.renpy/persistent

Obfuscating Your Game



For some reason, many people seem to want to distribute their games in an obfuscated form, making it difficult to read the script, or to view the images associated with a game. Ren'Py supports this, to a point, by compiling scripts and allowing images to be archived. While a moderately skilled programmer could easily crack this obfuscation, it should be enough to protect a game from casual browsing.

Obfuscating the script is practically automatic. Every time the game runs, any .rpy file in the game directory is written out as an .rpyc file. These .rpyc files are enough to run the game, so simply running the game once (to create the .rpyc files) and then deleting (or more appropriately, moving away) the .rpy files will leave you with a runnable game with an obfuscated script. As a bonus, the .rpyc files are already parsed, improving game load time. (If a directory contains .rpy and .rpyc files with the same stem, the newer one of them is chosen, meaning all games get this performance improvement.)

Images can be archived using the Ren'Py launcher. Just choose "Archive Images" from the "Tools" menu, and Ren'Py will create an archive containing your images. It will also add a script file that contains code telling Ren'Py where to look for the archived images.

When building distributions, you can use the option to exclude files to exclude .rpy files.

Localizing Ren'Py

While Ren'Py is by default set up to operate in an English speaking environment, it is not limited to such settings. Assuming a proper font is loaded, Ren'Py scripts can contain any language expressible in Unicode.

There are two things in the Ren'Py library that may need to be translated into a user's language. The first is the main menu. There is no explicit support for doing this, but as the config.main_menu variable supports changing the text of the main menu, it also supports translating said text.

The second thing that needs to be translated is the game menu. The config.translations dictionary is used to translate text in the game menu into your language. If a key in this map corresponds to the English text that would be displayed, the value corresponding to that key is displayed again. For example:

```
init:
    $ config.translations = {
        "Yes" : u"HIja'",
        "No" : u"ghobe'",
        # etc.
        }
```

The *u* characters prefixed to the strings on the right, while not strictly necessary in this case, are used to tell Python that the string is in Unicode rather than ASCII. This is useful if your language uses non-ascii characters.

The _ Function

Translation actually occurs using a function, named _, to look up the translation of a string. The default function uses config.translations as described above, but _ can be re-defined to use an alternate means of determining the translation.

Function: _ (s):

This function is called to translate s into the local language.

Properties and Styles

Ren'Py contains a style system, which is used to determine the look of displayables. This system allows the look of Ren'Py games to be customized to a great extent.

Any function that takes a **properties parameter can be given the properties described below, including prefixed properties, but not all properties are understood by all functions. These functions may also be given a *style* parameter. A style may be specified by supplying the style object directly, or by supplying a string giving the style name.

```
    $ ui.text("Hello, World", xalign=0.5, yalign=0.5, size=42,
style='default')
    $ ui.text('How are you doing?', style=style.centered_text)
    $ ui.saybehavior()
    $ ui.interact()
```

Styles

Each Displayable has a default style that is used when no style is explicitly given. Displayables used by Ren'Py are often given unique styles, which allow their look to be customized using the style system.

A style can be accessed as a field on the *style* object. Prefixed properties are fields on styles, which allow values to be assigned to them. Styles should only be created and changed inside init blocks.

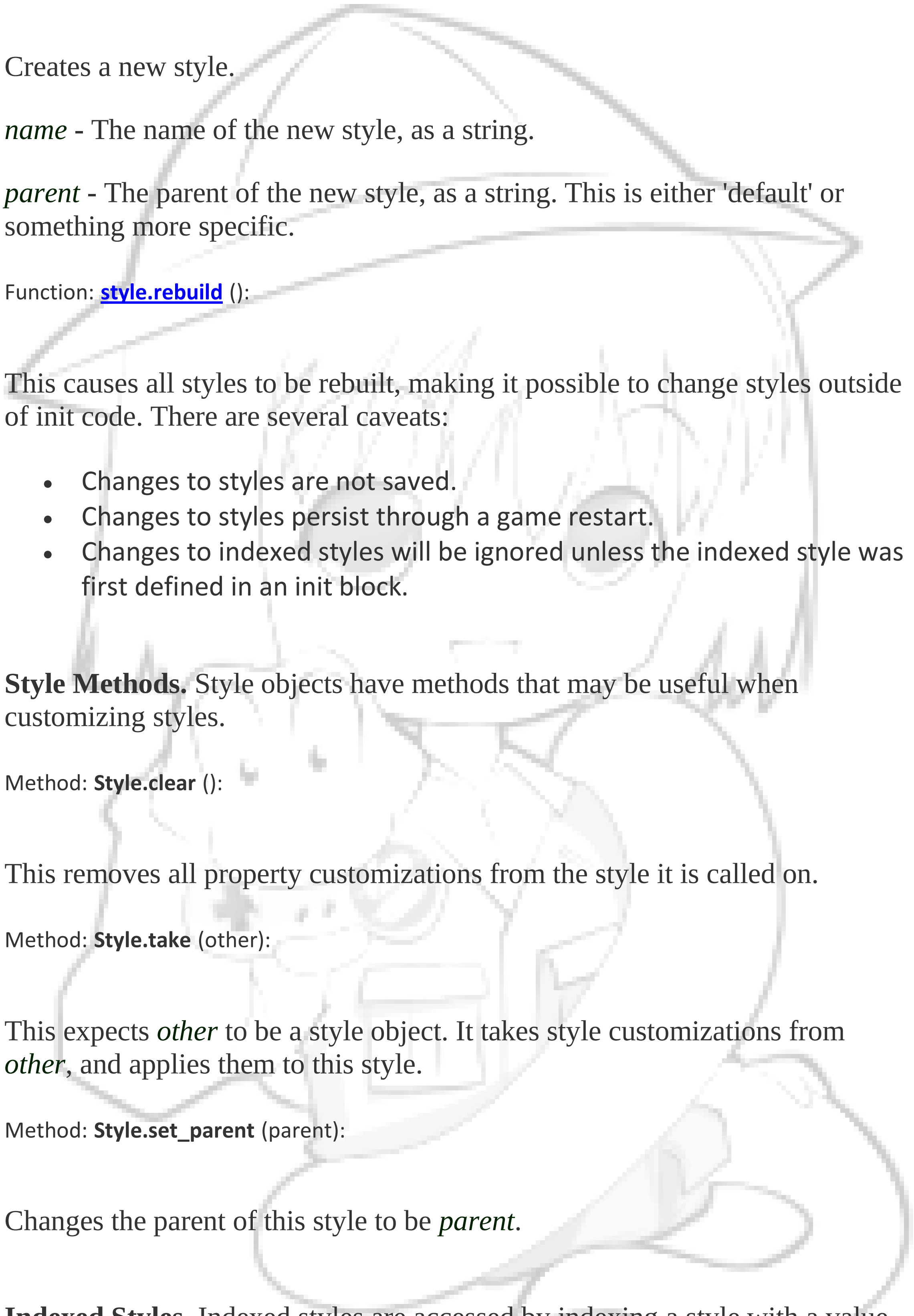
```
init:
    $ style.button_text.size = 22
    $ style.button_text.hover_size = 28
```

New styles can be created by using the Style function. These styles should be assigned to one or more fields on the style object at the time of creation.

Function: **Style** (parent):

Constructs a new style, with the given style as its parent. It's expected that the result of this call will be assigned to a field of the `style` object.

For compatibility purposes, styles may also be created using the style.create function.



Function: **style.create** (name, parent):

Creates a new style.

name - The name of the new style, as a string.

parent - The parent of the new style, as a string. This is either 'default' or something more specific.

Function: **style.rebuild** ():

This causes all styles to be rebuilt, making it possible to change styles outside of init code. There are several caveats:

- Changes to styles are not saved.
- Changes to styles persist through a game restart.
- Changes to indexed styles will be ignored unless the indexed style was first defined in an init block.

Style Methods. Style objects have methods that may be useful when customizing styles.

Method: **Style.clear** ():

This removes all property customizations from the style it is called on.

Method: **Style.take** (other):

This expects *other* to be a style object. It takes style customizations from *other*, and applies them to this style.

Method: **Style.set_parent** (parent):

Changes the parent of this style to be *parent*.

Indexed Styles. Indexed styles are accessed by indexing a style with a value. If an indexed style does not exist, indexing creates it. Indexed styles are lightweight, allowing there to be a large number of them. The parent of an indexed style is the style it was indexed off of.

```
init python:
    # Creates the new indexed style button['Foo']
    style.button['Foo'].background = "#f00"
    style.button['Foo'].xalign = 1.0

label example:
    python:
        # Accesses style button['Foo']
        ui.textbutton('Foo', style=style.button['Foo'],
clicked=ui.returns('Foo'))

        # Accesses style button['Bar'], creating it if necessary.
        ui.textbutton('Bar', style=style.button['Bar'],
clicked=ui.returns('Bar'))

        ui.interact()
```

Property Prefixes

The style system lets us specify new values for properties of a displayable depending on the situation that displayable is in. There are two components of the situation. The displayable may be selected or unselected, and it may be activated, focused, unfocused, or unfocusable. A displayable is activated when the button containing it is clicked, and returns to a previous state if the clicked function returns None. Prefixed properties let one set the value of properties for some subset of these situations.

In the following table, we give a list of prefixes and situations. We indicate which situations a prefix applies in with a *.

prefix	unselected				selected			
	activated	focused	unfocused	unfocusable	activated	focused	unfocused	unfocusable
(no prefix)	*	*	*	*	*	*	*	*
activate_	*				*			
hover_	*	*			*	*		
idle_			*				*	
insensitive_				*				*
selected_					*	*	*	*
selected_activate					*			

_								
selected_hover_					*	*		
selected_idle_							*	
selected_insensitive_								*

Resolving Conflicts

There are two kinds of conflicts that can occur.

If we have multiple prefixes that would assign to the same property in the same situation at the same time, then we take the more specific one, where a more specific is defined as being lower in the above chart. This is the case when more then one prefix is given for a property in the call to a function that takes properties as arguments.

When assignment occurs at distinct points in time, then the value of the property for a given situation is determined the last assignment performed. This is the case when we have assignment to style properties.

To avoid conflicts, assign values to prefixed properties in order from most to least specific.

Indexed Styles and Inheritance

Styles can be indexed with a string or number. This allows one to easily have styles that are specific to one button, without having to define all styles in advance. Indexed styles do not show up in the style hierarchy, but do show up when using the style inspector.

When styles inherit properties, inheritance is first through the indexed form of a style, then the unindexed form of that style, then the indexed form of that style's parent, then the unindexed form of the parent, and so on. Specifically, if style.mm_button inherits from style.button inherits from style.default, then the components of style.mm_button["Start Game"] are taken from:

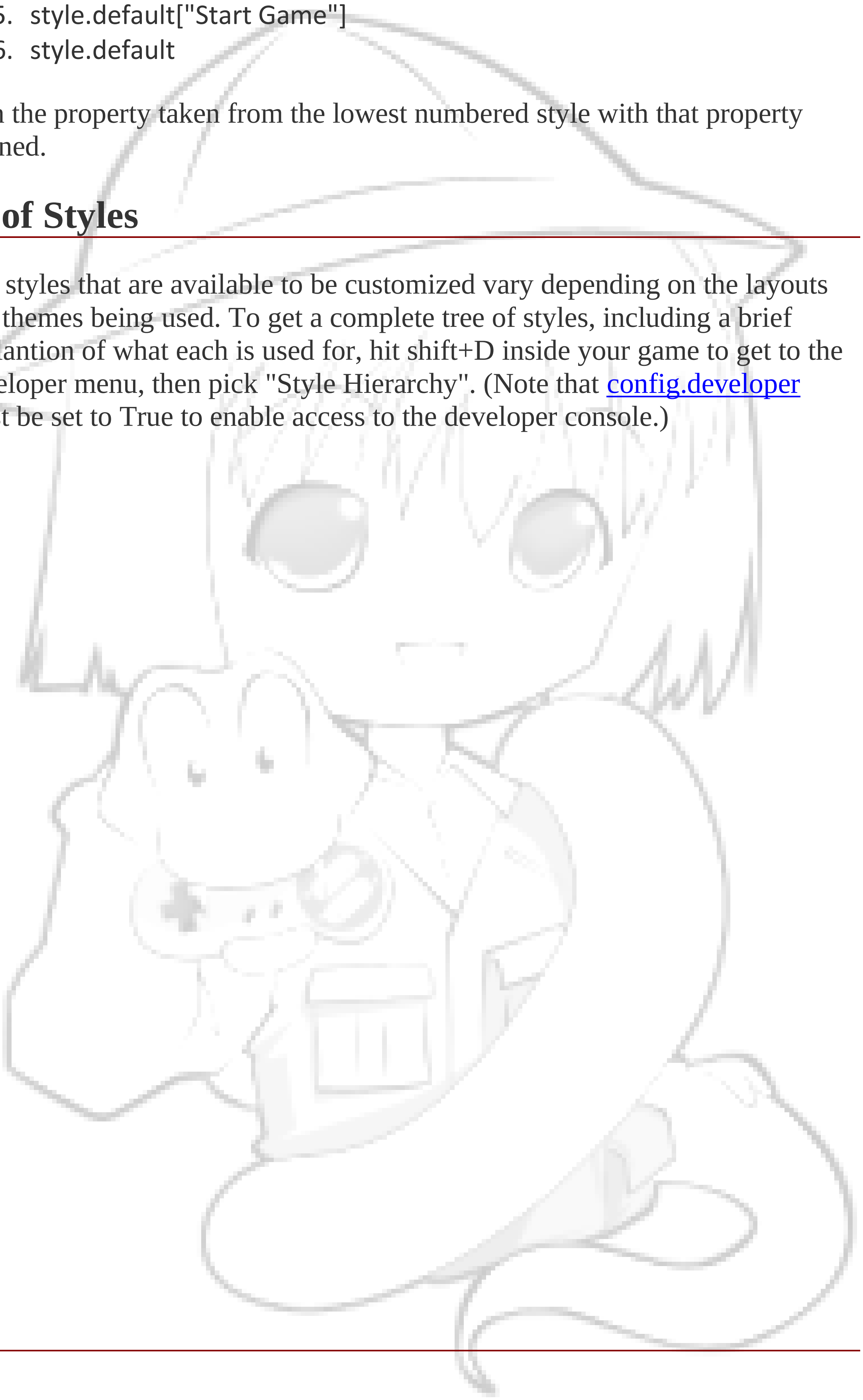
1. style.mm_button["Start Game"]
2. style.mm_button
3. style.button["Start Game"]

4. style.button
5. style.default["Start Game"]
6. style.default

with the property taken from the lowest numbered style with that property defined.

List of Styles

The styles that are available to be customized vary depending on the layouts and themes being used. To get a complete tree of styles, including a brief explantion of what each is used for, hit shift+D inside your game to get to the developer menu, then pick "Style Hierarchy". (Note that config.developer must be set to True to enable access to the developer console.)



List of Properties

This is a list of properties, loosely grouped by the use of the property.

Text Properties

The following properties are used by Text displayables.

antialias --- If True, the text will be antialiased. Otherwise, it will be left jagged.

black_color --- This should be an RGBA triple. When an SFont is used, this is the color the black will be mapped to. This property is ignored when a truetype font is used. The alpha channel in this color is ignored.

bold --- If True, then this text will be rendered in bold.

color --- The color in which the text will be displayed on the screen, as an RGBA tuple. When an SFont is used, this is the color that white in that Sfont will be displayed as.

drop_shadow --- This is used to control the generation of a drop shadow on text. It's either a 2-element tuple, a list of tuples, or None. If it's a tuple, then the 2 elements control the drop shadow offsets in X and Y, respectively. If it's a list, then it's expected to be a list of 2-element tuples, each of which is the drop shadow. Using a list like [(-1, -1), (1, -1), (-1, 1), (1, 1)] creates outlined text instead of a drop-shadow. If None, then no drop shadow is created for the text.

drop_shadow_color --- The color of the drop-shadow. Example: "#000"

first_indent --- This is used to give, in pixels, the indentation of the first line of text in the text widget. It can be used to indent the first line of a paragraph of text. (Not that that's a good idea on a computer monitor, better to leave a blank line between paragraphs.)

font --- The font that will be used to render text. This is either the name of a file containing a truetype font, or the name of a font. In the former case, the font is loaded from the file. In the latter case, the filename is used to search SFonts registered with register_sfont, and truetype fonts found on the system font directory.

size --- The size of the font that is used to display the text on the screen. Please note that the meaning of this can vary from font to font, and bears only a weak relationship with the number of pixels high that the font will be on the screen.

italic --- If True, then this text will be rendered in italics.

justify --- If True, additional whitespace is inserted between words so that the left and right margins of each line are even. This is not performed on the last line of a paragraph.

language --- The language (really, language family) that's used to display the text. Right now, this is one of two choices: "western" for western-style text which is broken at spaces, "eastasian" for Chinese/Japanese text, which can be broken between characters.

layout --- Controls how line-breaking occurs. This defaults to "greedy", which attempts to put as much as possible on each line. It may also be "subtitle", which tries to make the lines even in length.

line_spacing --- This is used to increase or decrease the spacing between lines of text by a constant number of pixels. A positive value increases, while a negative value decreases the spacing.

minwidth --- The minimum width in pixels of this text. If the rendered text is smaller than this, it is right-padded with whitespace until it is at least this many pixels long.

outlines --- A list of 2 or 4-component tuples, giving the size, color, and optionally the x and y offsets of outlines around the text. An outline is a block of text that has been expanded in size by a given number of pixels. The x and y offsets can be omitted. The first tuple in the list is furthest from the screen, while the last is closest. *(new in 6.8.0)*

rest_indent --- This is used to give, in pixels, the indentation of the second and later lines of a text widget. It can be used to give a hanging indent to quoted dialogue.

slow_cps --- If a number, causes text to be shown at a rate of that many characters per second. If True, takes that number from the "Text Speed" preference.

slow_cps_multiplier --- A multiplier that will be applied to the number of characters shown per second. For example, 2.0 causes characters to be shown twice as fast.

slow_abortable --- If True, slow text can be aborted by clicking the mouse.

text_align --- This is used to control the horizontal alignment of the lines of text in the area allocated to the Text widget containing that text. It only really has any effect if the text is more than one line long. It's a number between 0 and 1, which gives the fraction of empty space that should be to the left of each line of text. (To center text, it should be 0.5.)

text_y_fudge --- This fudges the y position of a block of text by adding whitespace above the first line.

underline --- If true, then this text will be rendered with an underline.

Window Properties

The properties control the look of Windows and Buttons.

background --- A Displayable that is used as the background for the window. This needs to be a Displayable that always draws exactly the size requested of it, which usually means either a Solid or a Frame. This can also be None, which means that there is no background on this window. (All the other window properties that refer to a background still work. Just think of them as if their background was transparent.)

foreground --- This is drawn in the same way as the background, except it is drawn above the contents of the window.

left_margin --- The amount of transparent space left to the left of this window. If a floating point number, it is scaled to the available width.

right_margin --- The amount of transparent space left to the right of this window. If a floating point number, it is scaled to the available width.

top_margin --- The amount of transparent space left to the top of this window. If a floating point number, it is scaled to the available height.

bottom_margin --- The amount of transparent space left to the bottom of this window. If a floating point number, it is scaled to the available height.

xmargin --- This is a convenient (and backwards compatible) way of setting left_margin and right_margin to the same value.

ymargin --- This is a convenient (and backwards compatible) way of setting top_margin and bottom_margin to the same value.

left_padding --- The amount of space left between the edge of the border and the left side of the contents of the window. If a float, it is scaled to the width of the window.

right_padding --- The amount of space left between the edge of the border and the right side of the contents of the window. If a float, it is scaled to the width of the window.

top_padding --- The amount of space left between the edge of the border and the top side of the contents of the window. If a float, it is scaled to the height of the window.

bottom_padding --- The amount of space left between the edge of the border and the bottom side of the contents of the window. If a float, it is scaled to the height of the window.

xpadding --- A convenient (and backwards compatible) way of setting left_padding and right_padding to the same value.

ypadding --- A convenient (and backwards compatible) way of setting top_padding and bottom_padding to the same value.

xfill --- If True, the window will expand to fill all available space in the x direction. If False, it will shrink to fit its contents.

yfill --- If True, the window will expand to fill all available space in the y direction. If False, it will shrink to fit its contents.

xmaximum --- The maximum size of this window in the x direction, including margins and padding. If a floating point number, it is scaled to the available width.

ymaximum --- The maximum size of this window in the y direction, including margins and padding. If a floating point number, it is scaled to the available height.

xminimum --- The minimum size of this window in the x direction, including margins and padding. If the window would be smaller than this, it is grown to be at least this size. If a floating point number, it is scaled to the available width.

yminimum --- The minimum size of this window in the y direction, including margins and padding. If the window would be smaller than this, it is grown to be at least this size. If a floating point number, it is scaled to the available height.

size_group --- If not None, this should be set to a string. All windows or buttons with a given string as their size_group are rendered at the same width. *(new in 6.6.0)*

clipping - If true and the child is bigger than the maximum size set for this window, it will be clipped to the size of the window.

Button Properties

These are only applicable to Buttons.

hover_sound --- The sound to play when this widget becomes hovered.

activate_sound --- The sound to play when the widget is activated, by clicking on or otherwise selecting it.

focus_mask --- This is used as a mask that determines if a button should be focused or not. If False, a default rectangle is used as the mask. Otherwise, if the special value True, the rendered displayable is used as the mask, or else the value is interpreted as a displayable, rendered to the size of the button, and used. In either case, the alpha value of the mask is what's used, with any non-zero amount of alpha indicating that a pixel is part of the mask. *New in 5.6.5.*

focus_rect --- If not None, this gives as a (x, y, width, height) tuple the focus rectangle of a button, as an offset from the upper-left corner of the button. For proper keyboard navigation, the focus rectangles of images are not allowed to overlap. By setting this and focus_mask on an imagebutton, one can have imagebuttons that overlap, but yet still support keyboard focus. This should almost certainly never be set on an individual image, but rather for each individual button (and it's rare to do even that).

time_policy --- A function that is responsible for modifying the displayable timebase of displayables used by buttons. This function expects to be given a time, a state object, and a style object, and is expected to return a new time and a new state. (The state object defaults to None the first time time_policy is called.) This function may want to determine the style prefix by inspecting its .prefix field. The default time_policy does not change the time.

child --- If not None, this displayable is displayed in the place of the child of this button.

mouse --- If not None, a string giving the name of the mouse cursor that is used when this button is focused. The name must be listed in config.mouse. This also works with other focusable displayables.

Bar Properties

The ui.bar has a few properties that are specific to it, that control the look of the bars. The bar has gutters on the left and the right. The remaining space is the space in which the bar can change, with the division of the bar being the fraction of this space representing the bar's value as a fraction of the range.

When the bar is drawn, the thumb's shadow is drawn first, followed by the left and right sides of the bar, followed by the thumb.

bar_vertical --- If true, the bar is drawn vertically, if false, horizontally.

bar_invert --- If true, the value of the bar is represented on the right/top side of the bar, rather then the left/bottom side of the bar.

bar_resizing --- If true, we resize the sub-bars, rather than rendering them full size and then cropping.

left_gutter --- The size of the left gutter of a horizontal bar, in pixels. (Named fore_gutter internally.)

right_gutter --- The size of the right gutter of a horizontal bar, in pixels. (Named aft_gutter internally.)

top_gutter --- The size of the top gutter of a vertical bar, in pixels. (Named fore_gutter internally.)

bottom_gutter --- The size of the bottom gutter of a vertical bar, in pixels. (Named aft_gutter internally)

left_bar --- A Displayable that is used to draw the left side of a horizontal bar. This displayable is first rendered at the full size of the bar, and then cropped so only the left side is visible. (Named fore_bar internally.)

right_bar --- A Displayable that is used to draw the right side of a horizontal bar. This displayable is first rendered at the full size of the bar, and then cropped so only the right side is visible. (Named aft_bar internally.)

top_bar --- A Displayable that is used to draw the top of a vertical bar. This displayable is first rendered at the full size of the bar, and then cropped so only the top is visible. (Named fore_bar internally.)

bottom_bar --- A Displayable that is used to draw the bottom of a vertical bar. This displayable is first rendered at the full size of the bar, and then cropped so only the bottom is visible. (Named aft_bar internally.)

thumb --- If not None, this is a thumb image that is drawn over the break between the sides of the bar.

thumb_shadow --- If not None, a shadow of the thumb that is drawn underneath the break between the sides of the bar.

thumb_offset --- The amount by which the thumb overlaps the bars, in pixels. To have the left and right bars continue unbroken, set this to half the width of the thumb. For compatibility's sake, we take the absolute value of this property.

unscrollable --- Controls what happens if the bar is unscrollable (if the range is set to 0, as is the case with a ui.viewport containing a displayable smaller than itself. There are three possible values:

- None - Renders the bar normally. (default)
- "insensitive" - Renders the bar in insensitive mode. This can be styled differently than a normal bar.
- "hide" - Prevents the bar from rendering at all. Space will be allocated for the bar, but nothing will be drawn in that space.

(new in 6.8.0)

Box Properties

The HBox, VBox, and Fixed displayables are instances of a common MultiBox displayable. These properties control how these displayables are laid out.

box_layout --- If "horizontal", the box is laid out in a horizontal manner. If "vertical", the box is laid out in a vertical fashion. If "fixed", the layout of the box is controlled by the position properties of the children of the box, and the rest of these properties are ignored. If None, the layout of the box is controlled by the function that created the box.

spacing --- The spacing between elements in the box, in pixels. This is also used for ui.grid and ui.side. This was formerly known as box_spacing, an alias which still works.

first_spacing --- If not None, the spacing between the first two elements in the box, in pixels. This is only used for boxes. This was formerly known as box_first_spacing, an alias that still works.

xfill --- If True, the box will expand to fill all available space in the x direction. If False, it will shrink to fit its contents.

yfill --- If True, the box will expand to fill all available space in the y direction. If False, it will shrink to fit its contents.

Position Properties

Position properties are applied to displayables that are smaller than the space allocated to them. They apply to all displayables, and control the placement of the widget in the space. For example, position properties can be used to control the placement of a dialogue window on the screen.

Position properties work best when a small displayables is placed into empty space. This is the case for say windows, and the menus that are displayed as part of the main menu. Position properties work on other displayables, but the vagaries of how space is allocated can make some of the results counterintuitive.

Positioning works by placing the displayable such that an anchor (specified relative to the displayable) is at the same location as the position (specified relative to the available space).

xanchor --- The x-axis location of the anchor, relative to the upper-left corner of the displayable. This may be an absolute number of pixels, or a floating point number which is interpreted as a fraction of the size of the displayable.

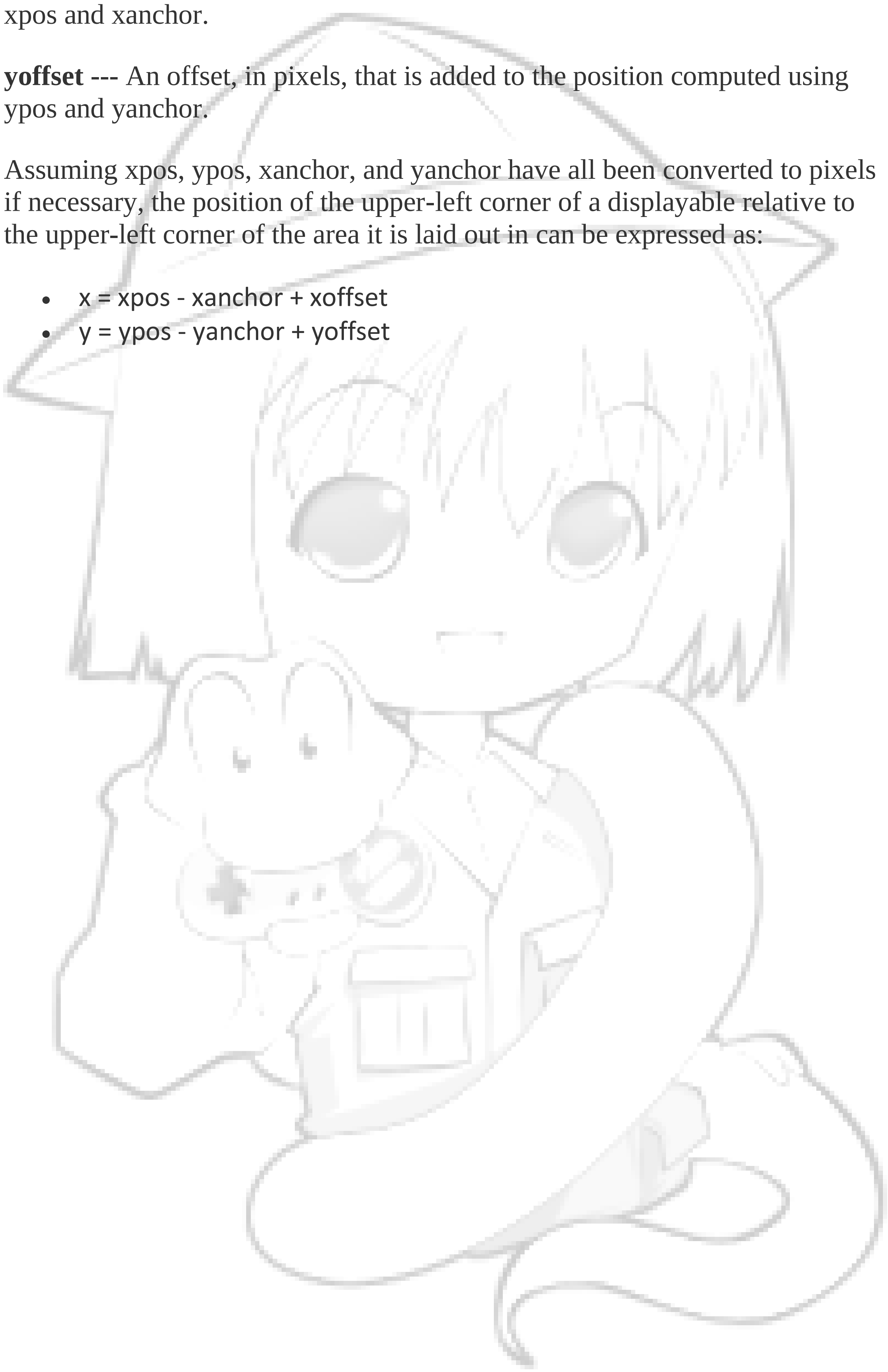
xpos --- The x-axis location of the position, relative to the upper-left corner of the available space. This may be an absolute number of pixels, or a floating point number which is interpreted as a fraction of the size of the available space.

xalign --- A shortcut for specifying xanchor and xpos at the same time. This should almost always be assigned a floating point number. As an example, xanchor=0.5 will center a displayable in the available space.

yanchor --- The y-axis location of the anchor, relative to the upper-left corner of the displayable. This may be an absolute number of pixels, or a floating point number which is interpreted as a fraction of the size of the displayable.

ypos --- The y-axis location of the position, relative to the upper-left corner of the available space. This may be an absolute number of pixels, or a floating point number which is interpreted as a fraction of the size of the available space.

yalign --- A shortcut for specifying yanchor and ypos at the same time. This should almost always be assigned a floating point number.



xoffset --- An offset, in pixels, that is added to the position computed using xpos and xanchor.

yoffset --- An offset, in pixels, that is added to the position computed using ypos and yanchor.

Assuming xpos, ypos, xanchor, and yanchor have all been converted to pixels if necessary, the position of the upper-left corner of a displayable relative to the upper-left corner of the area it is laid out in can be expressed as:

- x = xpos - xanchor + xoffset
- y = ypos - yanchor + yoffset

Customizing the Interface

In Ren'Py, there are four ways that one can customize the out-of-game menus:

- Layouts control the feel of the various screens that make up the game menu. They control the placement of components on those screen, and what those components do if they interact.
- Themes control the look of buttons, bars, frames, labels, and prompts.
- One can customize the main and game menus to add additional choices to those menus, over and above the ones provided by the layouts.
- Finally, one can use the style system to further customize the look and layout of the various out-of-game menu screens.

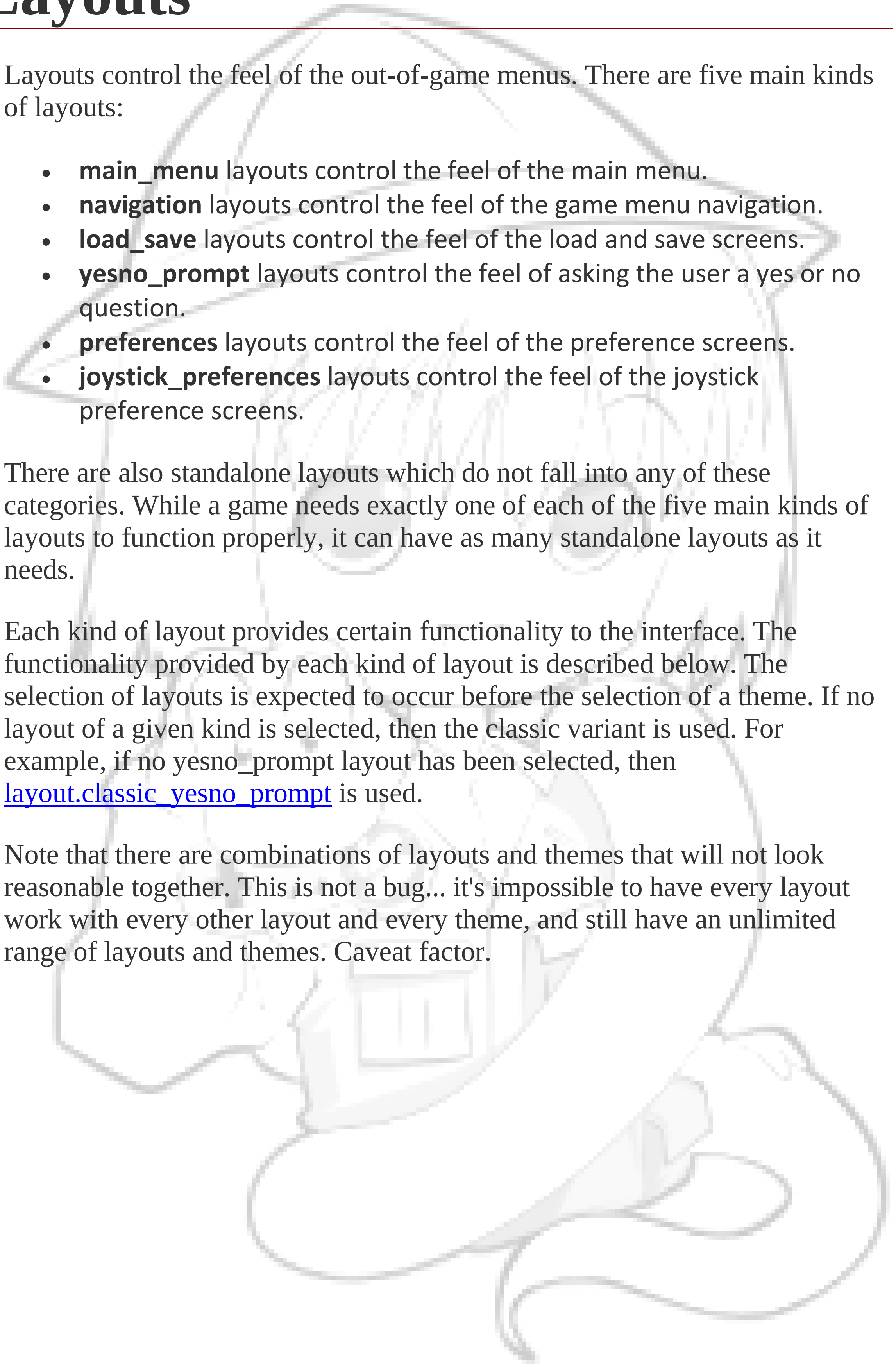
Of these, the only mandatory customization is the selection of a theme.

Order of Customizations

We expect customization of the game to occur in the following order:

1. If necessary, setting the script version (config.script_version), screen width (config.screen_width). and screen height (config.screen_height).
2. If necessary, selecting one or more layouts.
3. Selecting a master theme.
4. If necessary, selecting one or more theme components that override portions of the master theme.
5. If necessary, customize the main and game menus, and using the style system to further customize the interface.

Layouts



Layouts control the feel of the out-of-game menus. There are five main kinds of layouts:

- **main_menu** layouts control the feel of the main menu.
- **navigation** layouts control the feel of the game menu navigation.
- **load_save** layouts control the feel of the load and save screens.
- **yesno_prompt** layouts control the feel of asking the user a yes or no question.
- **preferences** layouts control the feel of the preference screens.
- **joystick_preferences** layouts control the feel of the joystick preference screens.

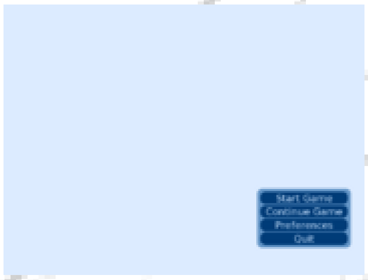
There are also standalone layouts which do not fall into any of these categories. While a game needs exactly one of each of the five main kinds of layouts to function properly, it can have as many standalone layouts as it needs.

Each kind of layout provides certain functionality to the interface. The functionality provided by each kind of layout is described below. The selection of layouts is expected to occur before the selection of a theme. If no layout of a given kind is selected, then the classic variant is used. For example, if no yesno_prompt layout has been selected, then layout.classic_yesno_prompt is used.

Note that there are combinations of layouts and themes that will not look reasonable together. This is not a bug... it's impossible to have every layout work with every other layout and every theme, and still have an unlimited range of layouts and themes. Caveat factor.

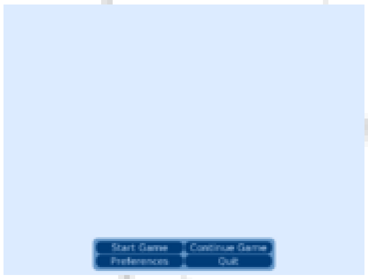
main_menu layouts

The main_menu layouts are responsible for defining the look of the main menu. This includes the background of the main menu, and the buttons that are used to go to the game menu or start a game.



Function: **layout.classic_main_menu** ():

This displays the classic main menu, which stacks the main menu buttons vertically.

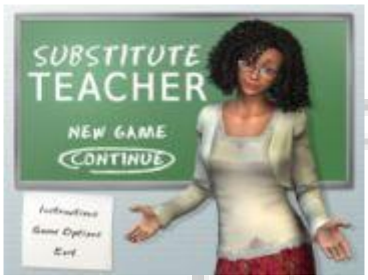


Function: **layout.grouped_main_menu** ():

This defines a main menu layout that groups buttons into rows horizontally, and then stacks those rows vertically.

Variable: **config.main_menu_per_group** = 2

The number of buttons placed in to each horizontal row.



Function: layout.imagemap_main_menu (ground, selected, hotspots, idle=None, variant=None):

ground - The ground image. This is used for parts of the screen that are not defined.

selected - The selected image. This used for hotspots that are hovered.

hotspots - A list of tuples defining the hotspot. Each tuple consists of:

1. The x-coordinate of the left side.
2. The y-coordinate of the top side.
3. The x-coordinate of the right side.
4. The y-coordinate of the bottom side.
5. Either the (untranslated) name of the main menu button this hotspot is equivalent to, or, a label in the program to jump to when this hotspot is clicked.

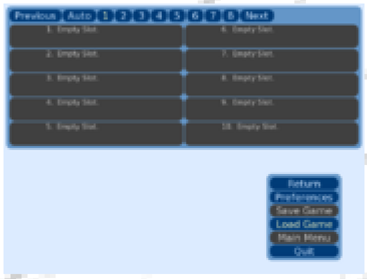
idle - If not None, an image that is used for images that are not hovered. If this is None, it defaults to ground. *(new in 6.10.1)*

variant - The variant of the main menu to define. The variable _main_menu_variant is used to select a variant to show to the user. *(new in 6.10.1)*

Despite the name, this function can take arbitrary displayables (like Animations) as well as images.

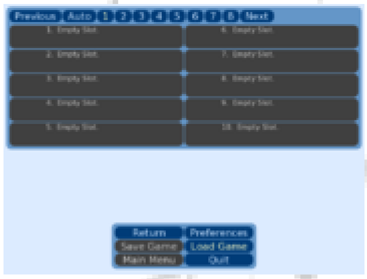
Navigation layouts

The navigation layouts are responsible for defining the navigation through the game menu. This includes the background of the game menus and the buttons that are used to go between game menu screens.



Function: layout.classic_navigation ():

This displays the navigation buttons in a vertical box.

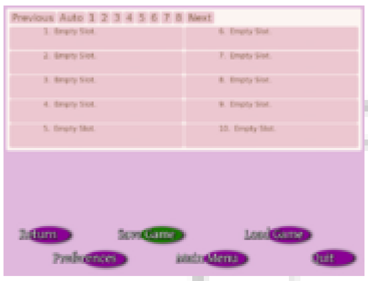


Function: layout.grouped_navigation ():

This displays the navigation buttons in horizontal groups, which are then stacked vertically.

Variable: **config.navigation_per_group** = 2

The number of navigation buttons per horizontal group.



Function: **layout.imagemap_navigation** (ground, idle, hover, selected_idle, selected_hover, hotspots):

This layout uses an imagemap to handle game menu navigation.

ground - The displayable used for disabled buttons, and areas that are not in a hotspot.

idle - The displayable used for unfocused unselected hotspots.

hover - The displayable used for focused unselected hotspots.

selected_idle - The displayable used for unfocused selected hotspots.

selected_hover - The displayable used for focused selected hotspots.

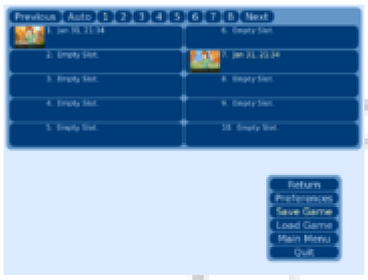
hotspots - A list of tuples defining the hotspots. Each tuple consists of

1. The x-coordinate of the left side.
2. The y-coordinate of the top side.
3. The x-coordinate of the right side.
4. The y-coordinate of the bottom side.
5. The (untranslated) name of the game menu button this hotspot is equivalent to.

Despite the name, this function can take arbitrary displayables as well as images.

load_save layouts

The load_save layouts are responsible for defining the load and save screens.



Function: layout.classic_load_save ():

This layout displays the the load and save screens as having a certain number of rows and columns of slots, with a row of navigation buttons on top.

Variable: **config.file_page_rows** = 5

The number of rows of file slots to display.

Variable: **config.file_page_cols** = 2

The number of columns of file slots to display.

Variable: **config.file_quick_access_pages** = 8

The number of pages of files to provide quick access buttons for.

Variable: **config.disable_file_pager** = False

If True, the pager is disabled. This prevents access to pages other than the first page of files.

Variable: **config.disable_thumbnails** = False

If True, thumbnails are not shown.

Variable: **config.load_save_empty_thumbnail** = None

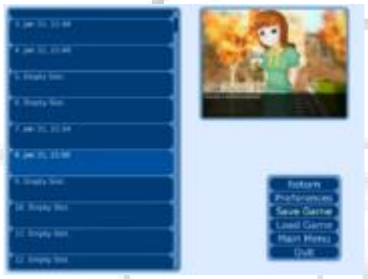
If not None, this should be a displayable that will be shown with empty load/save slots.

Variable: **config.time_format** = "%b %d, %H:%M"

The format used for file times in the file entry slots.

Variable: **config.file_entry_format** = "%(time)s\n%(save_name)s"

The format of file entries in the file entry slots.



Function: **layout.scrolling_load_save** ():

This uses a scrolling area that contains file picker entries. The user picks one of these entries to load or save a file. There is one big thumbnail to the right of the screen, which corresponds to the currently hovered entry. (config.thumbnail_width and config.thumbnail_height control the size of this thumbnail.)

Variable: **config.load_save_slots** = 50

The number of normal slots to show.

Variable: **config.load_save_auto_slots** = 5

The number of autosave slots to show.

Variable: **config.load_save_quick_slots** = 5

The number of quicksave slots to show.

Variable: **config.load_save_empty_thumbnail** = None

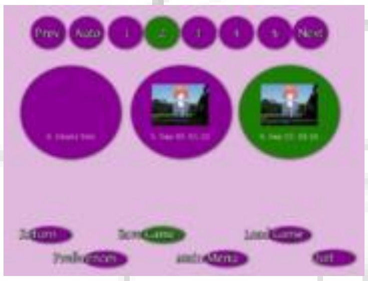
When not None, this should be a displayable that will be shown in the thumbnail frame when no save slot has been hovered.

Variable: **config.time_format** = "%b %d, %H:%M"

The format used for file times in the file entry slots.

Variable: **config.file_entry_format** = "%(time)s\n%(save_name)s"

The format of file entries in the file entry slots.



Function: **layout.imagemap_load_save** (ground, idle, hover, selected_idle, selected_hover, hotspots, variant=None):

This layout uses an imagemap to handle loading and saving.

ground - The displayable used for disabled buttons, and areas that are not in a hotspot.

idle - The displayable used for unfocused unselected hotspots.

hover - The displayable used for focused unselected hotspots.

selected_idle - The displayable used for unfocused selected hotspots.

selected_hover - The displayable used for focused selected hotspots.

hotspots - A list of tuples defining the hotspots. Each tuple consists of

1. The x-coordinate of the left side.
2. The y-coordinate of the top side.
3. The x-coordinate of the right side.
4. The y-coordinate of the bottom side.
5. The function of this hotspot.

The function of the hotspot should be one of:

- "previous" - go to the previous page.

- "next" - go to the next page.
- "page_auto" - go to the auto-save page.
- "page_quick" - go to the quick-save page.
- "page_1", "page_2", "page_3", ... - go to the specified page. The page numbers must start with 1, and be dense (not skip any numbers).
- "slot_0", "slot_1", "slot_2", ... - a save/load slot. The slot numbers must start with 0, and be dense.

variant - Allows us to define only the save or load screens. This can be "save", "load", or None to define the save and load screens at once. If a save screen is defined, a load screen must be defined (perhaps with different parameters), and vice versa.

Screenshots and slot text are placed inside windows that are laid out relative to the slot. Adjusting style.file_picker_ss_window controls the screenshot placement, and adjusting style.file_picker_text_window controls the placement of per-slot test. It will usually be necessary to adjust these styles, as the default places both in the upper left.

Hotspot functions may also include the untranslated names of game menu buttons. If at least one such button is defined, the navigation is not shown, and the imagemap is expected to define all relevant game menu buttons.

Despite the name, this function can take arbitrary displayables as well as images. The images or displayables used should be transparent to allow the navigation to show through, unless the game menu buttons are defined here.

Variable: **config.disable_thumbnails** = False

If True, thumbnails are not shown.

Variable: **config.load_save_empty_thumbnail** = None

If not None, this should be a displayable that will be shown with empty load/save slots.

Variable: **config.time_format** = "%b %d, %H:%M"

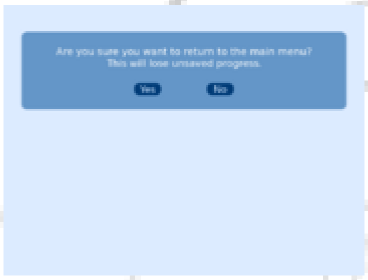
The format used for file times in the file entry slots.

Variable: **config.file_entry_format** = "%(time)s\n%(save_name)s"

The format of file entries in the file entry slots.

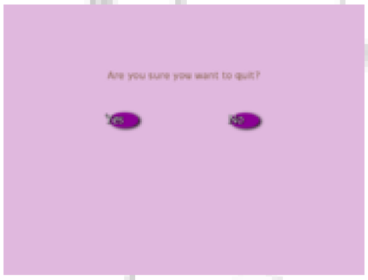
yesno_prompt layouts

The yesno_prompt layouts are responsible for defining yes/no prompt screens.


Are you sure you want to return to the main menu?
This will lose unsaved progress.

Function: layout.classic_yesno_prompt ():

This displays the classic yes/no prompt, which is just a prompt above two buttons.


Are you sure you want to quit?

Function: layout.imagemap_yesno_prompt (ground, idle, hover, hotspots,
prompt_images={}):

This layout uses an imagemap to handle prompting the user with yes/no questions.

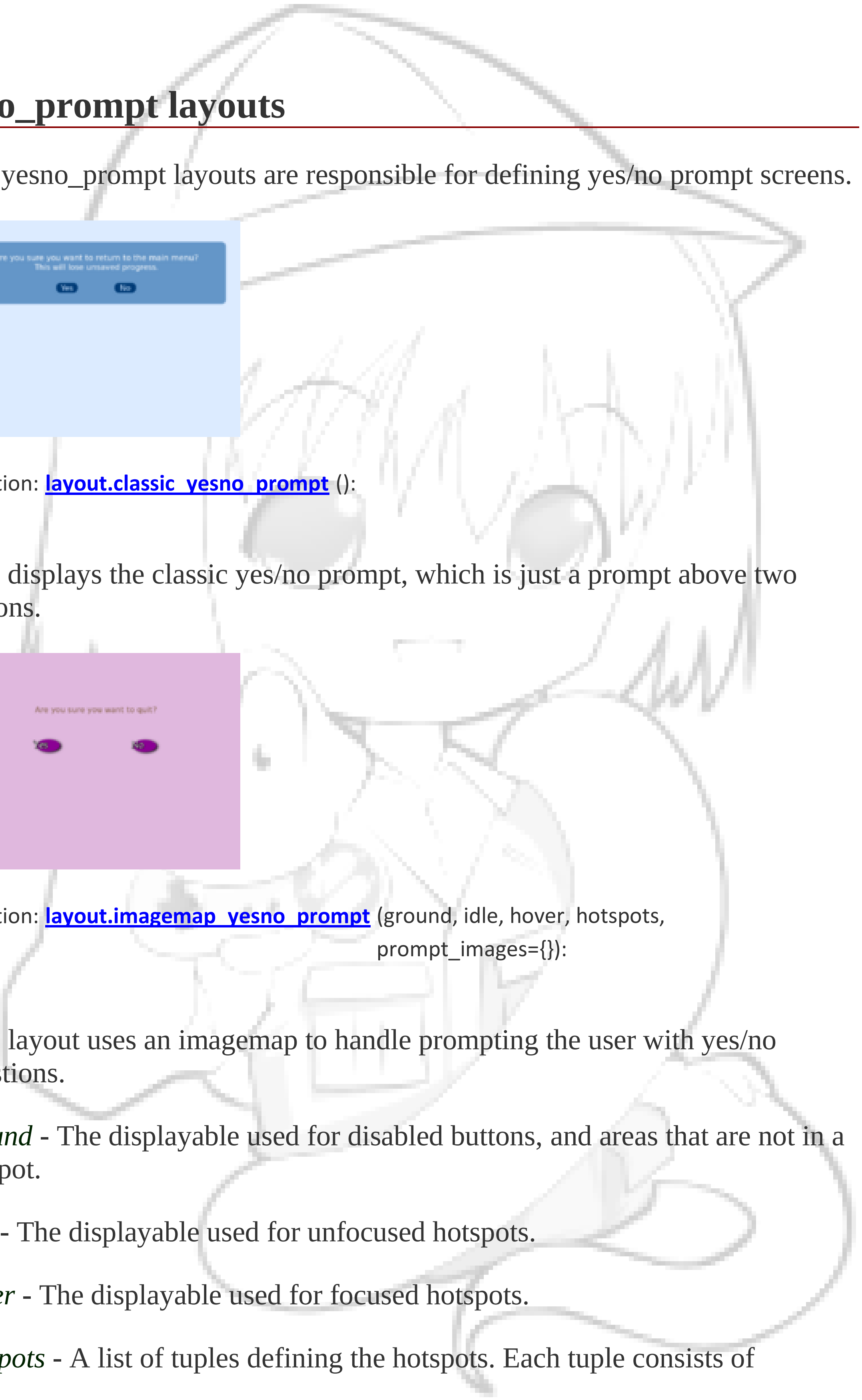
ground - The displayable used for disabled buttons, and areas that are not in a hotspot.

idle - The displayable used for unfocused hotspots.

hover - The displayable used for focused hotspots.

hotspots - A list of tuples defining the hotspots. Each tuple consists of

1. The x-coordinate of the left side.



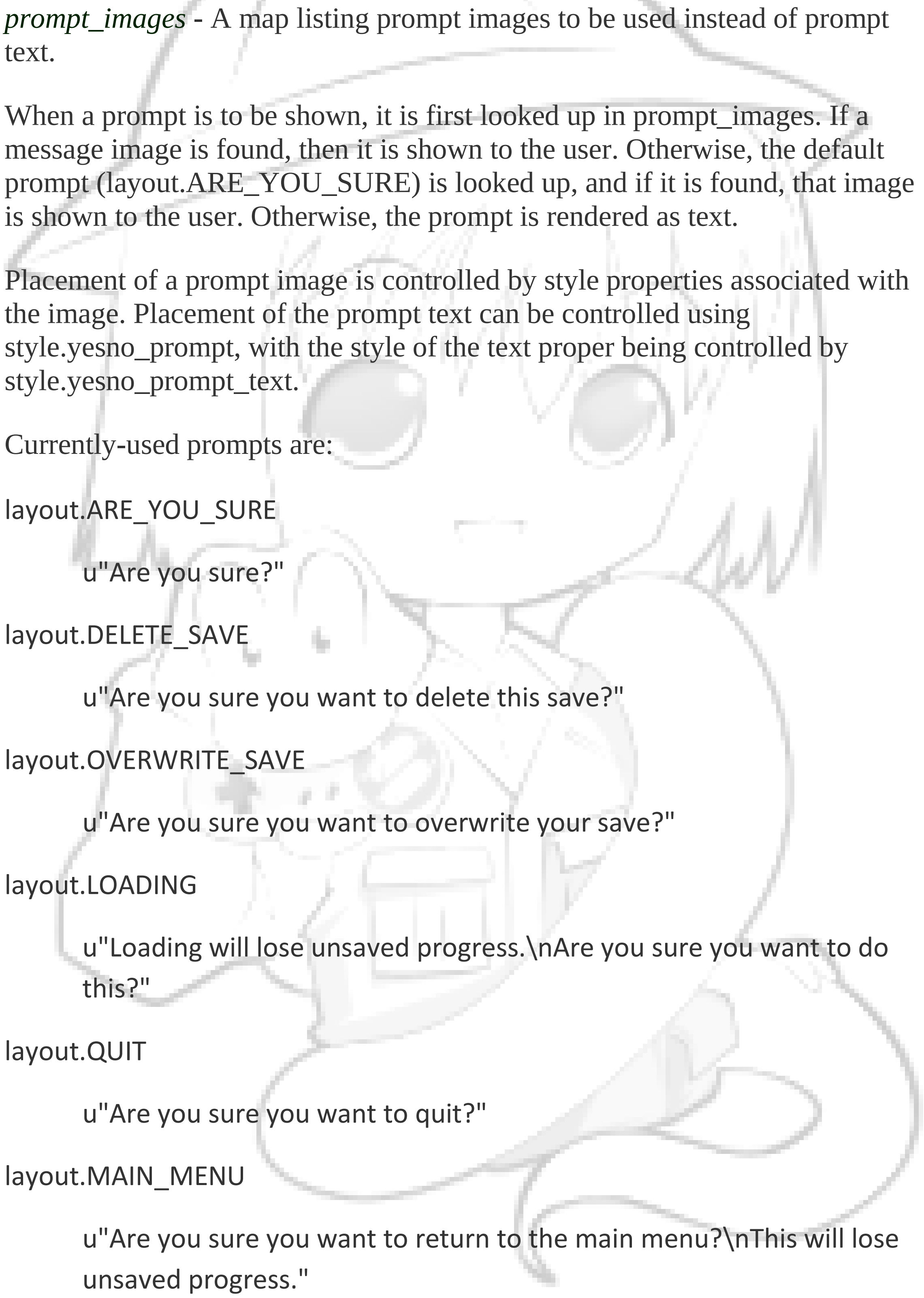
2. The y-coordinate of the top side.
3. The x-coordinate of the right side.
4. The y-coordinate of the bottom side.
5. The button replaced by this hotspot (one of "Yes" or "No").

prompt_images - A map listing prompt images to be used instead of prompt text.

When a prompt is to be shown, it is first looked up in prompt_images. If a message image is found, then it is shown to the user. Otherwise, the default prompt (layout.ARE_YOU_SURE) is looked up, and if it is found, that image is shown to the user. Otherwise, the prompt is rendered as text.

Placement of a prompt image is controlled by style properties associated with the image. Placement of the prompt text can be controlled using style.yesno_prompt, with the style of the text proper being controlled by style.yesno_prompt_text.

Currently-used prompts are:



layout.ARE_YOU_SURE

u"Are you sure?"

layout.DELETE_SAVE

u"Are you sure you want to delete this save?"

layout.OVERWRITE_SAVE

u"Are you sure you want to overwrite your save?"

layout.LOADING

u"Loading will lose unsaved progress.\nAre you sure you want to do this?"

layout.QUIT

u"Are you sure you want to quit?"

layout.MAIN_MENU

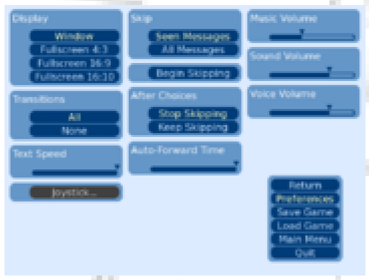
u"Are you sure you want to return to the main menu?\nThis will lose unsaved progress."

Hotspot functions may also include the untranslated names of game menu buttons. If at least one such button is defined, the navigation is not shown, and the imagemap is expected to define all relevant game menu buttons.

Despite the name, this function can take arbitrary displayables as well as images. The images or displayables used should be transparent to allow the navigation to show through, unless the game menu buttons are defined here.

preferences layouts

The preferences layouts are used to define the preferences screen.



Function: layout.classic_preferences ():

This uses a three-column preferences layout.

Variable: **config.sample_sound** = None

A sound file used to test the sound volume.

Variable: **config.sample_voice=None** = None

A sound file used to test the voice volume.

Variable: **config.has_transitions** = True

If True, the option to enable or disable transitions is shown.

Variable: **config.has_cps** = True

If True, the option to control text speed is shown.

Variable: **config.has_afm** = True

If True, the option to control auto-forward mode is shown.

Variable: **config.has_skipping** = True

If True, the option to control skipping read messages or not is shown

Variable: **config.has_skip_after_choice** = True

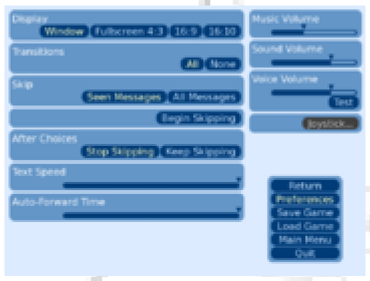
If True, the option to control skipping after choices is shown

Variable: **config.always_has_joystick** = False

If True, the link to the joystick page is always active.

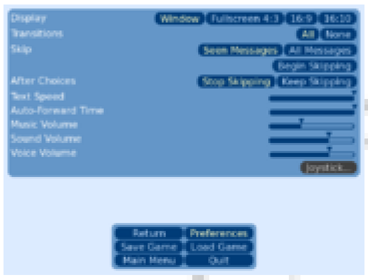
Variable: **config.has_joystick** = True

If True, the link to the joystick page is shown.



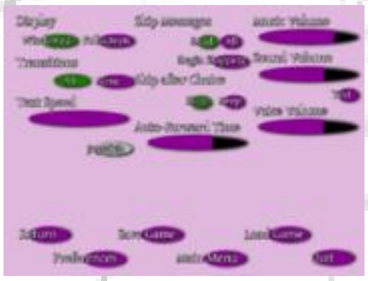
Function: **layout.two_column_preferences** ():

This uses a two-column preferences layout. Configuration variables are the same as for layout.classic_preferences.



Function: **[layout.one_column_preferences]()** ():

This uses a one-column preferences layout. Configuration variables are the same as for [layout.classic_preferences]().



Function: **[layout.imagemap_preferences]()** (ground, idle, hover, selected_idle, selected_hover, hotspots):

This layout uses an imagemap to handle preferences.

ground - The displayable used for disabled buttons, and areas that are not in a hotspot.

idle - The displayable used for unfocused unselected hotspots.

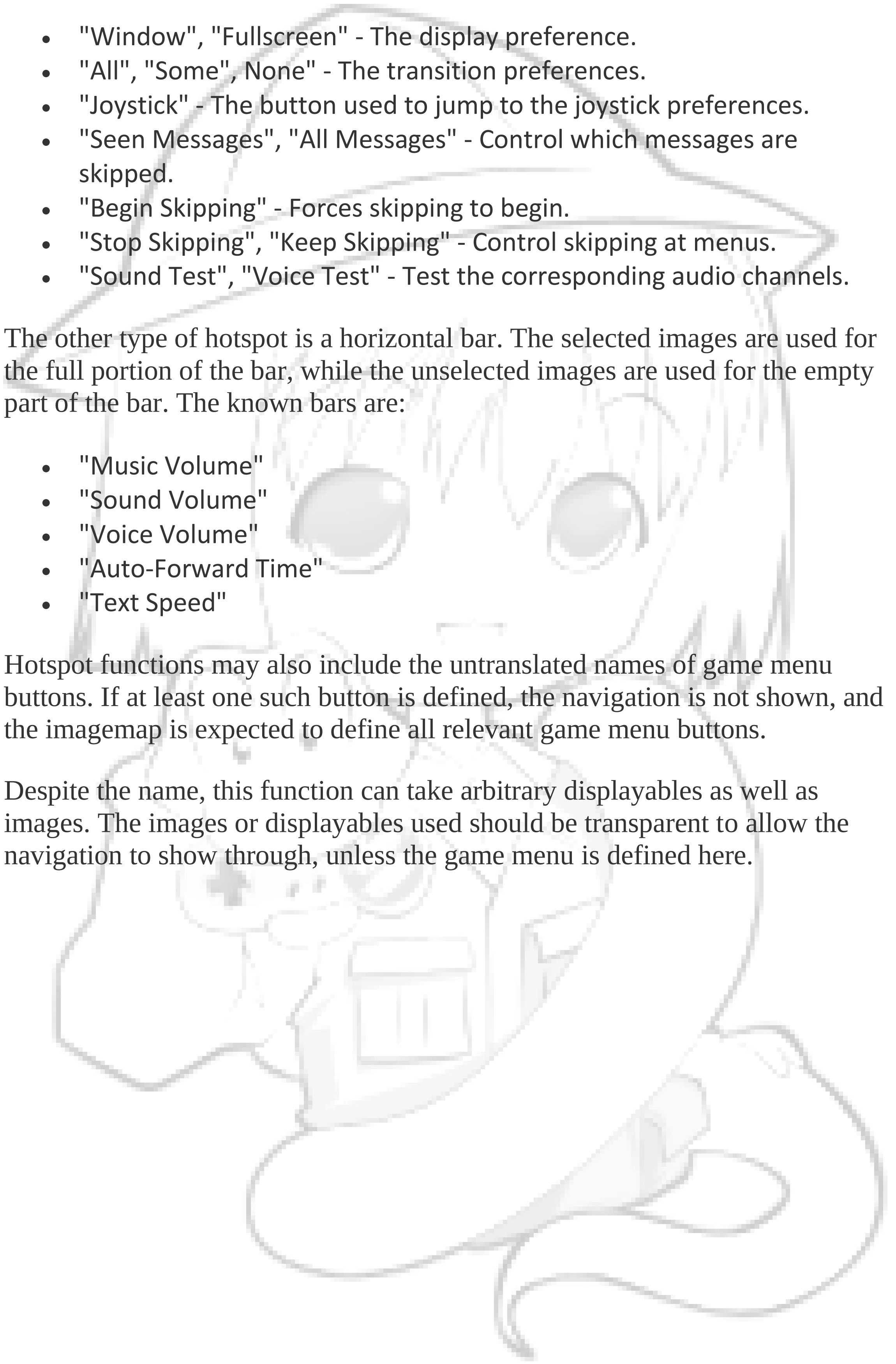
hover - The displayable used for focused unselected hotspots.

selected_idle - The displayable used for unfocused selected hotspots.

selected_hover - The displayable used for focused selected hotspots.

hotspots - A list of tuples defining the hotspots. Each tuple consists of

1. The x-coordinate of the left side.
2. The y-coordinate of the top side.
3. The x-coordinate of the right side.
4. The y-coordinate of the bottom side.
5. The function of this hotspot.



There are two kinds of hotspots, buttons and bars. The buttons are:

- "Window", "Fullscreen" - The display preference.
- "All", "Some", None" - The transition preferences.
- "Joystick" - The button used to jump to the joystick preferences.
- "Seen Messages", "All Messages" - Control which messages are skipped.
- "Begin Skipping" - Forces skipping to begin.
- "Stop Skipping", "Keep Skipping" - Control skipping at menus.
- "Sound Test", "Voice Test" - Test the corresponding audio channels.

The other type of hotspot is a horizontal bar. The selected images are used for the full portion of the bar, while the unselected images are used for the empty part of the bar. The known bars are:

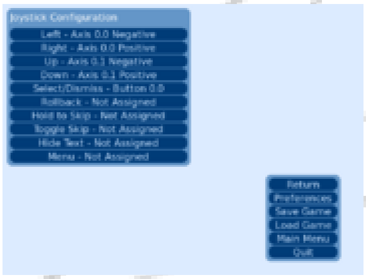
- "Music Volume"
- "Sound Volume"
- "Voice Volume"
- "Auto-Forward Time"
- "Text Speed"

Hotspot functions may also include the untranslated names of game menu buttons. If at least one such button is defined, the navigation is not shown, and the imagemap is expected to define all relevant game menu buttons.

Despite the name, this function can take arbitrary displayables as well as images. The images or displayables used should be transparent to allow the navigation to show through, unless the game menu is defined here.

joystick_preferences layouts

The joystick_preferences layout are used to define the joystick preferences screen.

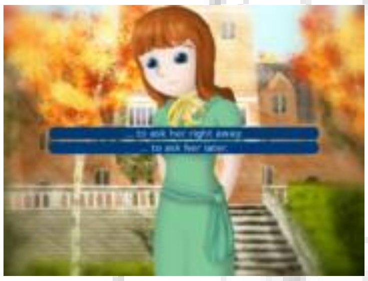


Function: **layout.classic_joystick_preferences** ():

The standard joystick preferences layout.

standalone layouts

These provide interesting functionality to Ren'Py games, and they stand alone, so it's possible to call as many of them as you want.



Function: **layout.button_menu** ():

This changes the in-game menus to use buttons defined in the current theme.

Defining New Layouts

This section contains some notes on how to define your own layouts. If you're not interested in defining your own labels, then you can safely ignore this section.

When it comes to defining layouts, there are two important principles you should follow:

1. layout.provides must be called with the type of layout being defined, and it must be called before any theme function is called. For example, if a navigation layout is being defined, then `layout.provides("navigation")` should be called.
2. The layout needs to implement the contract of it's layout kind. These contracts are described below.

While the default layouts are enable through the use of functions on the `layout` object, this is by no means the only way to supply a layout. For a user-defined layout, it should be enough to place the call to layout.provides in an `init -2 python` block, with the rest of the code residing in init blocks or labels as appropriate.

If your layout defines new configuration variables, you should set config.locked to False before creating them, and then back to True when you're done creating them. We suggest that configuration variables should be prefixed with the type of layout they're used by. Load/save variables should be prefixed by `config.load_save_`, navigation variables with `config.navigation_` and so on. (For compatibility reasons, the default layouts do not conform to this standard.)

main_menu layouts

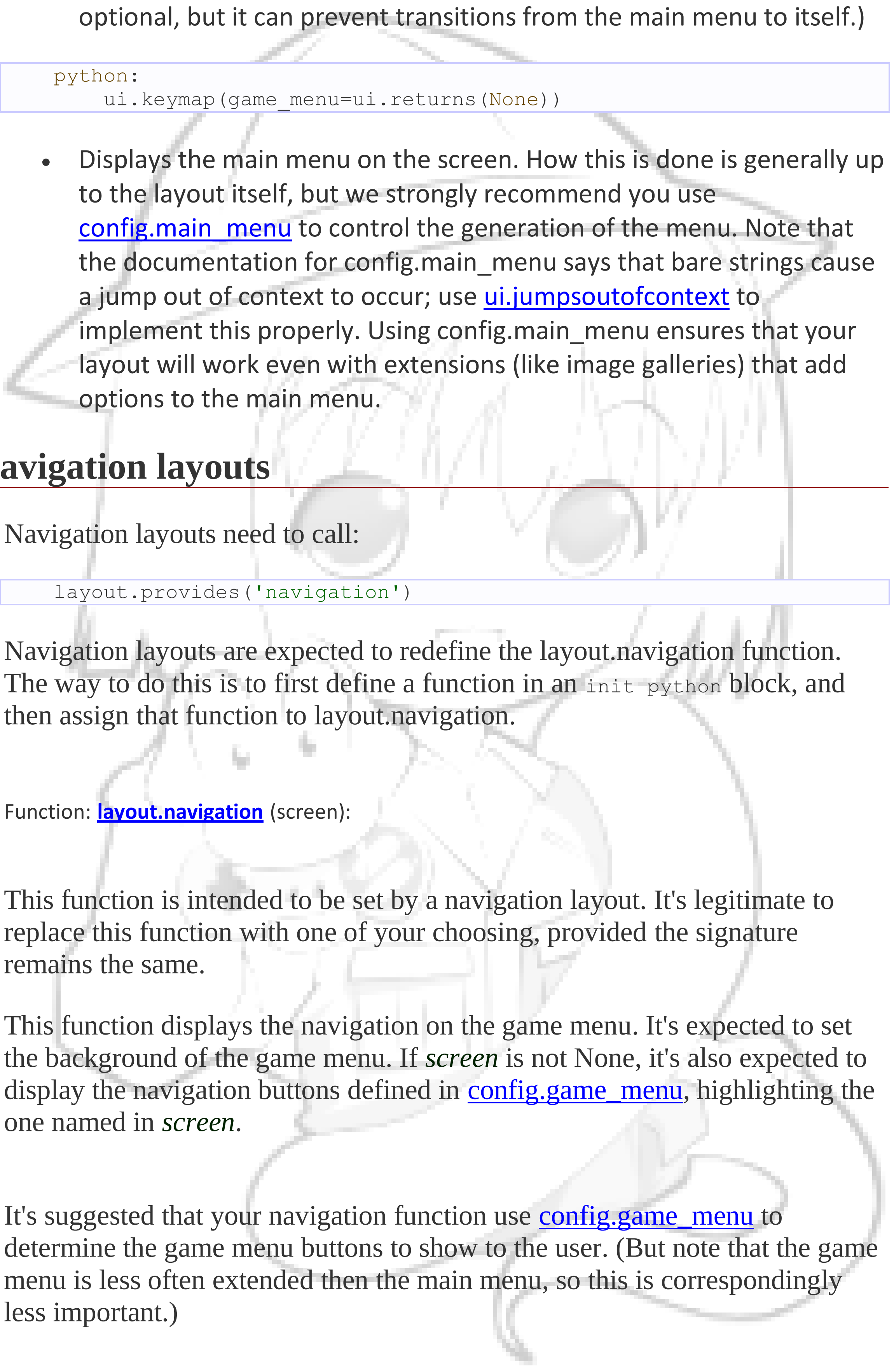
Main_menu layouts need to call:

```
    layout.provides('main_menu')
```

Main menu layouts are expected to define a `main_menu_screen` label. This label is expected to:

- Show a fullscreen window of type mm_root. The code to do this is:

```
    python:
        ui.window(style='mm_root')
        ui.null()
```



- Create a keymap that disables the game_menu binding. (This is optional, but it can prevent transitions from the main menu to itself.)

```
    python:
        ui.keymap(game_menu=ui.returns(None))
```

- Displays the main menu on the screen. How this is done is generally up to the layout itself, but we strongly recommend you use config.main_menu to control the generation of the menu. Note that the documentation for config.main_menu says that bare strings cause a jump out of context to occur; use ui.jumpsoutofcontext to implement this properly. Using config.main_menu ensures that your layout will work even with extensions (like image galleries) that add options to the main menu.

navigation layouts

Navigation layouts need to call:

```
    layout.provides('navigation')
```

Navigation layouts are expected to redefine the layout.navigation function. The way to do this is to first define a function in an `init python` block, and then assign that function to layout.navigation.

Function: **layout.navigation** (screen):

This function is intended to be set by a navigation layout. It's legitimate to replace this function with one of your choosing, provided the signature remains the same.

This function displays the navigation on the game menu. It's expected to set the background of the game menu. If *screen* is not None, it's also expected to display the navigation buttons defined in config.game_menu, highlighting the one named in *screen*.

It's suggested that your navigation function use config.game_menu to determine the game menu buttons to show to the user. (But note that the game menu is less often extended then the main menu, so this is correspondingly less important.)

load_save layouts

Load/save layouts need to call:

```
    layout.provides('load_save')
```

Load/save layouts are expected to provide two labels: `load_screen` and `save_screen`, which are used to load and save, respectively. These screens should call layout.navigation with "load" or "save" as an argument, and are otherwise unconstrained in how they provide loading and saving functionality.

Please see the section on load/save functions for the functions that would be used to actually implement these screens.

yesno_prompt layouts

Yes/no prompt layouts need to call:

```
    layout.provides('yesno_prompt')
```

Navigation layouts are expected to redefine the layout.yesno_prompt function. The way to do this is to first define a function in an `init python` block, and then assign that function to layout.yesno_prompt.

Function: **layout.yesno_prompt** (screen, message):

This function is intended to be customized by yesno_prompt layouts. It can be replaced by another function, provided the signature remains the same.

screen - The screen button that should be highlighted when this prompt is shown. If None, then no game menu navigation is shown.

message - The message that is shown to the user to prompt them to answer yes or no.

This function returns True if the user clicks Yes, or False if the user clicks No.

preferences layouts

Preferences layouts need to call:

```
    layout.provides('preferences')
```

The preferences layout should define a `preferences_screen` label, which contains code to set the preferences. This screen should call layout.navigation("preferences"). It then needs to allow the user to alter the preferences, using the variables and functions defined in the preferences section. Finally, it should:

- Allow the user to begin skipping, by jumping to the `_begin_skipping` label.
- Allow the user to go to the joystick preferences, by jumping to the `joystick_preferences_screen` label.

joystick_preferences layouts

Joystick preferences layouts need to call:

```
    layout.provides('joystick_preferences')
```

The preferences layout should define a `joystick_preferences_screen` label, which contains code to set the joystick preferences. This screen should call layout.navigation("joystick_preferences"). Unfortunately, the joystick preferences are undocumented at this time.

Example

An example of a new main_menu layout:

```
init -2 python:
    layout.provides('main_menu')

    style.mm_button = Style(style.button, help="main menu button")
    style.mm_button_text = Style(style.button_text, help="main menu
button (text)")
    style.mm_button.size_group = "mm"

label main_menu_screen:
    python:
        layout.button(u"Start Game", "mm",
clicked=ui.jumpsoutofcontext('start'), xpos=400, ypos=400)
        layout.button(u"Continue Game", "mm",
clicked=_intra_jumps("load_screen", "main_game_transition"), xpos=450,
ypos=430)
```

```
        layout.button(u"Preferences", "mm",
clicked=_intra_jumps("preferences_screen", "main_game_transition"),
xpos=500, ypos=460),
        layout.button(u"Quit", "mm", clicked=ui.jumps("_quit"),
xpos=550, ypos=490)

        ui.interact()
```

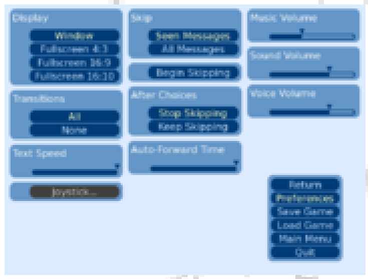
Themes

Themes provide a simple way of changing the look of the main and game menus. A single function call applies styles to many of the elements of the main and game menus, giving a consistent look to the interface.

Theme functions should be called after the config.screen_width, config.screen_height, and library.script_version variables have been set, and after any layout functions have been called. They should be called before any style is changed by hand.

Theme Functions

These theme functions are



Function: **theme.roundrect** (widget="#003c78", widget_hover="#0050a0", widget_text="#c8e1ff", widget_selected="#ffffc8", disabled="#404040", disabled_text="#c8c8c8", label=,"#ffffff" frame="#6496c8", window="#000000c0", text_size=None, small_text_size=None, mm_root=..., gm_root=..., less_rounded=False):

This enables the use of the roundrect theme. By default, this theme styles the game in a blue color scheme. However, by supplying one or more of the parameters given below, the color scheme can be changed.

widget - The background color of non-focued buttons and sliders.

widget_hover - The background color of focused buttons and sliders.

widget_text - The text color of non-selected buttons.

widget_selected - The text color of selected buttons.

disabled - The background color of disabled buttons.

disabled_text - The text color of disabled buttons.

label - The text color of non-selected labels.

frame - The background color of frames.

mm_root - A displayable (such as an Image or Solid) that will be used as the background for the main menu.

gm_root - A displayable (such as an Image or Solid) that will be used as the background for the game menu.

less_rounded - If True, causes the buttons to appear less rounded in 800x600 mode (has no effect in 640x480 mode).

text_size - The size of text, such as button captions, labels, and prompts. Defaults to 18 if the screen is 640 pixels wide or less, and 22 otherwise.

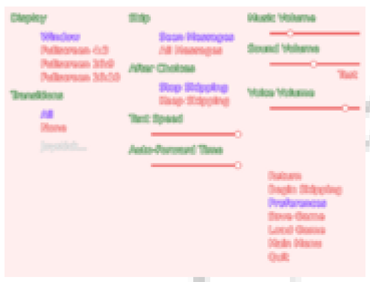
small_text_size - The size of the text on large buttons. Defaults to 12 if the screen is 640 pixels wide or less, and 16 otherwise.

widget, *widget_hover*, *disabled*, and *frame* may either be single colors, or tuples containing two colors. In the latter case, a vertical gradient is used.

Component functions

The following functions exist to allow you to add elements of the roundrect theme to another theme. The other theme must have been set up before these functions can be used. Arguments are as for roundrect, except that all must be specified (no defaulting occurs).

- **theme.roundrect_labels**(text_size, label)
- **theme.roundrect_frames**(less_rounded, frame)
- **theme.roundrect_buttons**(text_size, less_rounded, widget, widget_hover, widget_text, widget_selected, disabled, disabled_text)
- **theme.roundrect_large_buttons**(text_size, less_rounded, widget, widget_hover, widget_text, widget_selected, disabled, disabled_text)
- **theme.roundrect_prompts**(text_size, label)
- **theme.roundrect_bars**(widget, widget_hover)



Function: **theme.outline** (inside="#fff", idle="#e66", hover="#48f", selected="#84f", insensitive="#ccc", label="#484", prompt="#484", background="#fee", large_button="#fff8f8", text_size=22, small_text_size=16):

This function selects a theme that is based on outlining text in different colors.

inside - The color of text inside the various outlines.

idle - The outline color of the text of an idle button or bar.

hover - The outline color of a hovered button or bar.

selected - The outline color of a selected button.

insensitive - The outline color of an insensitive button.

label - The outline color of a label.

prompt - The outline color of a prompt.

background - A displayable used for the game and main menu backgrounds.

large_button - The background color of large backgrounds.

text_size - The size of large text. (Used for buttons, labels, and prompts.)

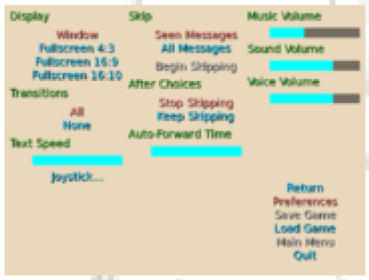
small_text_size - The size of small text. (Used in large buttons.)

Component functions

The following functions exist to allow you to add elements of the outline theme to another theme. The other theme must have been set up before these functions can be used. Arguments are as for theme.outline, except that all must be specified (no defaulting occurs).

- theme.outline_frames()

- theme.outline_buttons(inside, idle, hover, selected, insensitive, text_size)
- theme.outline_large_buttons(inside, idle, hover, selected, insensitive, text_size, large_button)
- theme.outline_prompts(inside, prompt, text_size)
- theme.outline_labels(inside, label, text_size)
- theme.outline_bars(inside, idle, hover)



Function: **theme.ancient** ():

This is a theme that attempts to emulate the theme used by Ren'Py 6.5.0 when no theme was explicitly specified.

Theme Modifiers

These are functions that can be called after a theme function, allowing you to change a portion of a theme.

Function: **theme.image_buttons** (d):

Used to define buttons in terms of 5-tuples of image filenames. This expects its single parameter, *d*, to be a dictionary mapping untranslated button names to 5-tuples. Each 5-tuple should contain 5 filenames, giving the image used for the button when:

- The button is idle.
- The button is hovered.
- The button is selected and idle.
- The button is selected and hovered.
- The button is insensitive.

in that order.

Function: **theme.image_labels** (d):

Replaces labels with images. This takes a single parameter, *d*, which is expected to be a map from the text of a label to an image file.

Custom Theme

It's also possible to define your own Ren'Py theme. A custom theme consists of Ren'Py code that does the following.

- Calls layout.defaults() to initialize the layouts. Calling this function is the responsibility of the theme, and custom themes must also do it.
- Set up the base styles to reflect the requirements of the theme.

Often, the base styles come in *name*/*name*_text pairs. In these cases, *name* represents a Button or Window with style *name*, in which a Text with style *name*_text lives.

The base styles are:

style.frame (inherits from style.default)

Used for frames on top of which the rest of the interface can comfortably sit.

style.button (inherits from style.default)

style.button_text (inherits from style.default)

Used for buttons, especially buttons whose primary purpose is navigating through the interface. (Like the main menu and the game menu navigation buttons.)

style.small_button (inherits from style.button)

style.small_button_text (inherits from style.button_text)

Used for smaller navigation buttons. It might make sense to set a minimum width on buttons, but small_buttons should be allowed to shrink as small as possible.

style.radio_button (inherits from style.button)

style.radio_button_text (inherits from style.button_text)

Used for buttons that are arranged in a group, such that only one of the buttons in the group can be selected at a time.

style.check_button (inherits from style.button)

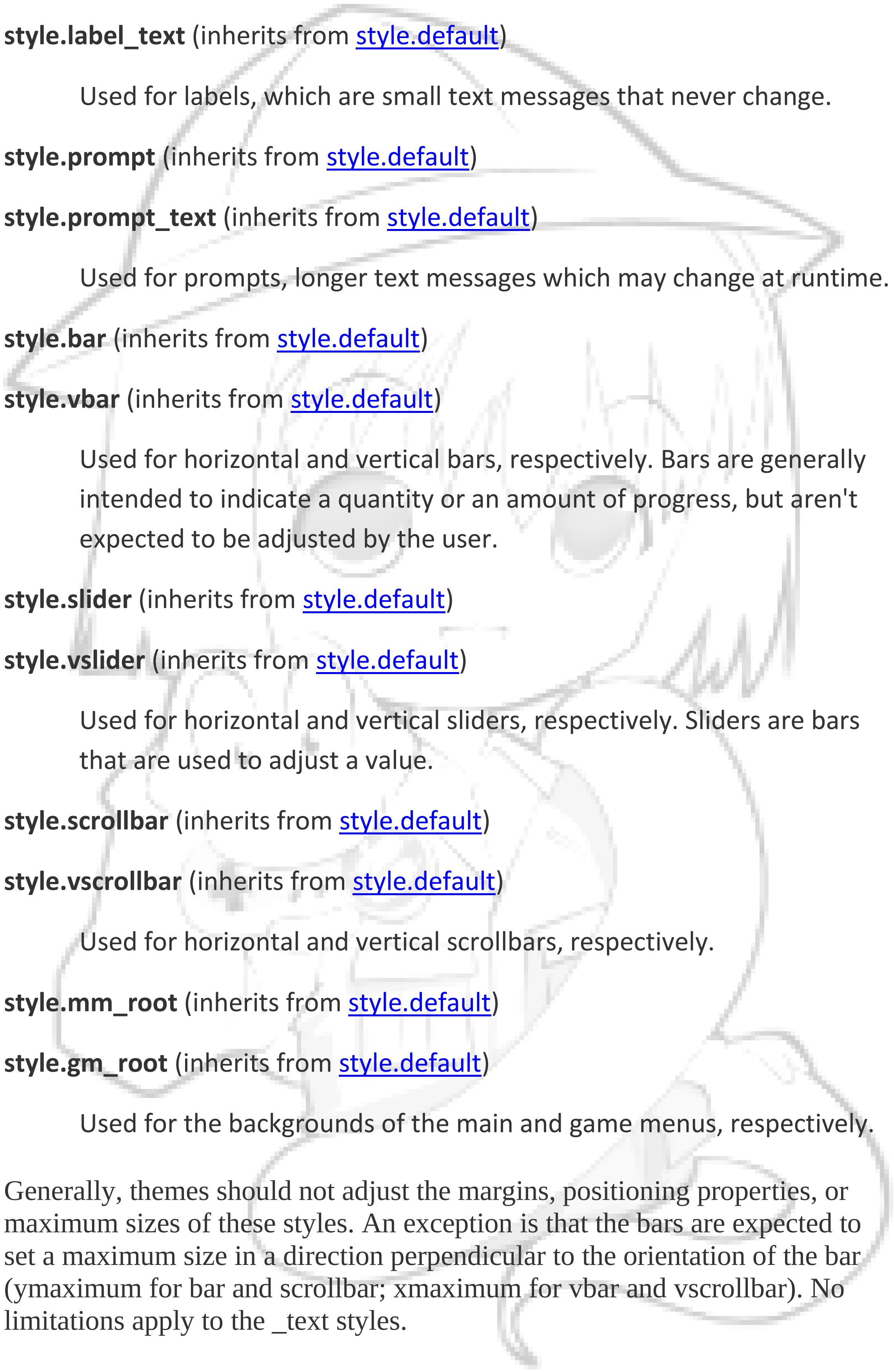
style.check_button_text (inherits from style.button_text)

Used for buttons that toggle their selected state to indicate if an option is set or not. (These aren't used in any of the pre-defined layouts.)

style.large_button (inherits from style.default)

style.large_button_text (inherits from style.default)

Used for large buttons, such as file picker entries, that can contain a large amount of text and other information.



style.label (inherits from style.default)

style.label_text (inherits from style.default)

Used for labels, which are small text messages that never change.

style.prompt (inherits from style.default)

style.prompt_text (inherits from style.default)

Used for prompts, longer text messages which may change at runtime.

style.bar (inherits from style.default)

style.vbar (inherits from style.default)

Used for horizontal and vertical bars, respectively. Bars are generally intended to indicate a quantity or an amount of progress, but aren't expected to be adjusted by the user.

style.slider (inherits from style.default)

style.vslider (inherits from style.default)

Used for horizontal and vertical sliders, respectively. Sliders are bars that are used to adjust a value.

style.scrollbar (inherits from style.default)

style.vscrollbar (inherits from style.default)

Used for horizontal and vertical scrollbars, respectively.

style.mm_root (inherits from style.default)

style.gm_root (inherits from style.default)

Used for the backgrounds of the main and game menus, respectively.

Generally, themes should not adjust the margins, positioning properties, or maximum sizes of these styles. An exception is that the bars are expected to set a maximum size in a direction perpendicular to the orientation of the bar (ymaximum for bar and scrollbar; xmaximum for vbar and vscrollbar). No limitations apply to the _text styles.

Main and Game Menus

This section describes how the content of the various menus can be customized. If you just want to change the look of the menus (to the extent made possible by the style system), use themes or styles. In particular, if you want to change the image background to the main menu and game menu, use the properties *mm_root* and *gm_root* in a theme.

To change just the text of menu items, consider using config.translations. To change the screens on the menus, you'll need to change the contents of the config.main_menu and config.game_menu variables. Otherwise, you'll want to use a layout that changes the look of the menus, or write your own layout.

Main Menu

The main menu can be customized by setting the config.main_menu variable. This variable should be a list of tuples. The first component of each tuple is the name of the button on the main menu. The second component can be one of three things:

- a string, in which case it is a label at which the game execution starts (after a jump out of the menu context).
- a function, in which case the function is called when the button is clicked.
- Finally, it can be None, which causes the button to be insensitive.

The third component must be a string containing a python expression, which determines when the button is enabled. The optional fourth component is a string containing a python expression, that determines if the button is shown at all. *(changed in 6.6.2)*

To jump from the main menu to one of the screens in the game menu, one should use the _intra_jumps function to specify the appropriate transition.

Function: **_intra_jumps** (label, transition):

Jumps to *label* while remaining in the same context.

transition should be a string giving a field on the `config` object. When this function is run, that field is accessed, and is used to specify the transition that is used.

The default value of config.main_menu is:

```
    config.main_menu = [
        (u"Start Game", "start", "True"),
        (u"Continue Game", _intra_jumps("load_screen",
"main_game_transition"), "True"),
        (u"Preferences", _intra_jumps("preferences_screen",
"main_game_transition"), "True"),
        (u"Quit", ui.jumps("_quit"), "True")
        ]
```

Game Menu

The first thing one may wish to do when modifying the game menu is to add a screen to it. This is done in two steps. The first is to create the screen, and the second is to add it to the config.game_menu list so that it can be reached from the game menu.

Each screen is represented in the code as a label that is jumped to to display the screen. There are five steps that this label should do in order to display the screen. First, it should call layout.navigation(screen), where screen is the name of the screen (the first component of the tuples in config.game_menu, described below). Second, it should call the ui functions to add components to the the screen. Third, it should call ui.interact to interact with the user. Fourth, it should examine the results of _game_interact, and react appropriately. Finally, it should jump back up to the screen label, showing the screen again after each interaction.

So that the user can see it, the screen should be added to config.game_menu. This is a list of four- or five-component tuples. The first is the name of the screen. It's used to determine if the button used to reach that screen should be indicated as selected. The second component is the text used for that button. The third component is a function that executes when the button is clicked. Normally, an appropriate function is `_intra_jumps`(*label*, *transition*), where label is the name of the label of your screen. The fourth component is a string containing a python expression. If the expression is not true, the button is insensitive. The optional fifth component is also a string containing a python expression. If the expression is not true, the button is not shown. *(changed in 6.6.2)*

The default value of config.game_menu is:

```
    config.game_menu = [
        ( None, u"Return", ui.jumps("_return"), 'True'),
        ( "preferences", u"Preferences",
_intra_jumps("preferences_screen", "intra_transition"), 'True' ),
        ( "save", u"Save Game", _intra_jumps("save_screen",
"intra_transition"), 'not main_menu' ),
        ( "load", u"Load Game", _intra_jumps("load_screen",
"intra_transition"), 'True'),
        ( None, u"Main Menu",
ui.callsinnewcontext("_main_menu_prompt"), 'not main_menu' ),
        ( None, u"Quit", ui.callsinnewcontext("_quit_prompt"), 'True'
),
        ]
```

Variable: **_game_menu_screen** = "_load_screen"

One can customize the screen the game menu jumps to by default by changing the value of _game_menu_screen. For example, one could set this to "load_screen" for the first few interactions, and then set it to "save_screen" for the rest of the game. This is especially useful for a game with no main menu.

If set to None, the user cannot enter the game menu. This is set to None at the start of the splashscreen, and reset to its old value when the splashscreen completes.

Use the layout.yesno_prompt function to ask yes/no questions of the user.

Function: **layout.yesno_prompt** (screen, message):

screen - The screen button that should be highlighted when this prompt is shown. If None, then no game menu navigation is shown.

message - The message that is shown to the user to prompt them to answer yes or no.

This function returns True if the user clicks Yes, or False if the user clicks No.

Pre-defined Screens. Layouts define the load screen, save screen, and preferences screen are define at the `load_screen`, `save_screen`, and `preferences_screen`, respectively.

Please see the page on Saving, Loading, and Rollback for a list of functions that can be used in the load and save screens, and the page on the Preferences for a list of preferences that can be customized.

Jump to `_return` to return to the game or the main menu, as appropriate. Jump to `_noisy_return` to play config.exit_sound before returning.

Entering the Game Menu

The game menu can be activated immediately using renpy.game_menu, or ui.gamemenus can be used to enter the game menu when a button is clicked. Both functions take an optional argument, the name of the screen to enter, normally one of "load_screen", "save_screen", or "preferences_screen". This can be omitted to use the default screen.

```
init python:
    def overlay():
         ui.textbutton("Prefs",
clicked=ui.gamemenus("preferences_screen"), xalign=1.0, yalign=0.0)

    config.overlay_functions.append(overlay)

label battle_begin:
    menu:
        "A battle is about to begin! Would you like to save your
game?"
        "Yes.":
              $ renpy.game_menu("save_screen")
        "No.":
              pass
```

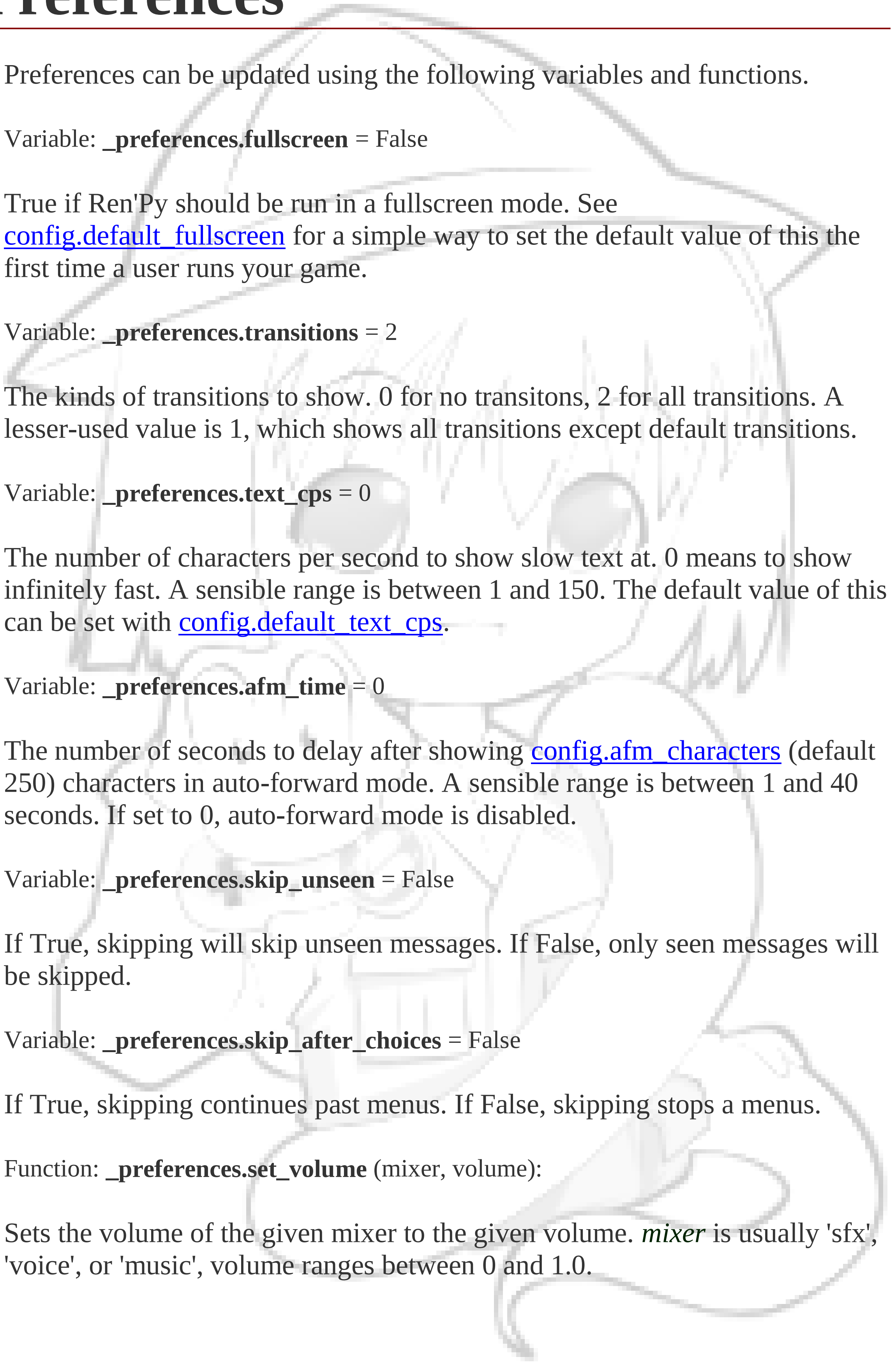
Compatibility Mode

Ren'Py also supports a compatibility mode that makes layouts and themes more compatible with how they were used in Ren'Py 6.5.0. This compatibility mode is automatically enabled when config.script_version is set to (6, 5, 0) or less. It can also be enabled explicitly by calling layout.compat().

Function: **layout.compat** ():

This enables a compatibility mode that sets up themes and styles as they were used in Ren'Py 6.5.0 and prior versions.

Preferences



Preferences can be updated using the following variables and functions.

Variable: **_preferences.fullscreen** = False

True if Ren'Py should be run in a fullscreen mode. See config.default_fullscreen for a simple way to set the default value of this the first time a user runs your game.

Variable: **_preferences.transitions** = 2

The kinds of transitions to show. 0 for no transitons, 2 for all transitions. A lesser-used value is 1, which shows all transitions except default transitions.

Variable: **_preferences.text_cps** = 0

The number of characters per second to show slow text at. 0 means to show infinitely fast. A sensible range is between 1 and 150. The default value of this can be set with config.default_text_cps.

Variable: **_preferences.afm_time** = 0

The number of seconds to delay after showing config.afm_characters (default 250) characters in auto-forward mode. A sensible range is between 1 and 40 seconds. If set to 0, auto-forward mode is disabled.

Variable: **_preferences.skip_unseen** = False

If True, skipping will skip unseen messages. If False, only seen messages will be skipped.

Variable: **_preferences.skip_after_choices** = False

If True, skipping continues past menus. If False, skipping stops a menus.

Function: **_preferences.set_volume** (mixer, volume):

Sets the volume of the given mixer to the given volume. *mixer* is usually 'sfx', 'voice', or 'music', volume ranges between 0 and 1.0.

Function: **_preferences.get_volume** (mixer):

Gets the volume of *mixer*, which is usually 'sfx', 'voice', or 'music'. The returned volume ranges between 0 and 1.0.

Customizing the Keymap

The variable config.keymap contains a map from functions that can be performed by the various user-interface elements of Ren'Py to a list of keysyms (a keysym is a **key sym**bol) that actually perform those functions. Modifying the contents of the keymap can change the keys and mouse buttons that cause things to happen.

While this functionality has been added to Ren'Py at a user's request (and because it simplifies the Ren'Py code), it's not altogether clear that it should be used. Having a common set of keybindings makes games easier to play by reducing the learning curve of users. It's probably better to build consensus around a change in keybindings, rather than unilaterally making one game different from every other game.

Anyway, in Ren'Py keysyms are strings. The first kind of keysym is of the form 'mouseup_#' or 'mousedown_#', for a number between 1 and 5. These keysyms are generated by mouse button presses, releases, or turns of the mouse wheel. For example, "mousedown_1" is generally a press of the left mouse button, "mouseup_1" is a release of that button, and "mousedown_4" is a turn of the the mouse wheel to the top.

A second kind of keysym is a joystick keysym. These begin with joy_. They are defined in config.joystick_keys, and mapped to actual joystick events by the user.

A third kind of keysym is a string containing a character that is generated when a key is pressed. This is useful for binding alphabetic keys and numbers. Examples of these keysyms include "a", "A", and "7".

The final kind of keysym is the symbolic name for the key. This can be any of the K_ constants taken from pygame.constants. This type of keysym looks like "K_BACKSPACE", "K_RETURN", and "K_TAB"; a full list of this kind of keysyms may be found here.

To change a binding, update the appropriate list in config.keymap. The following code adds the 't' key to the list of keys that dismiss a say statement, and removes the space key from that list.

```
init:
    $ config.keymap['dismiss'].append('t')
    $ config.keymap['dismiss'].remove('K_SPACE')
```

The default keymap is contained inside the python code implementing Ren'Py, and as of version 6.9.1 is as follows:

```
config.keymap = dict(

    # Bindings present almost everywhere, unless explicitly
    # disabled.
    rollback = [ 'K_PAGEUP', 'mousedown_4', 'joy_rollback' ],
    screenshot = [ 's' ],
    toggle_fullscreen = [ 'f', 'alt_K_RETURN', 'alt_K_KP_ENTER' ],
    toggle_music = [ 'm' ],
    game_menu = [ 'K_ESCAPE', 'mouseup_3', 'joy_menu' ],
    hide_windows = [ 'mouseup_2', 'h', 'joy_hide' ],
    launch_editor = [ 'E' ],
    dump_styles = [ 'Y' ],
    reload_game = [ 'R' ],
    inspector = [ 'I' ],
    developer = [ 'D' ],
    quit = [ 'meta_q', 'alt_K_F4', 'alt_q' ],
    iconify = [ 'meta_m', 'alt_m' ],
    help = [ 'K_F1', 'meta_shift_/' ],

    # Say.
    rollforward = [ 'mousedown_5', 'K_PAGEDOWN' ],
    dismiss = [ 'mouseup_1', 'K_RETURN', 'K_SPACE', 'K_KP_ENTER',
'joy_dismiss' ],

    # Pause.
    dismiss_hard_pause = [ ],

    # Focus.
    focus_left = [ 'K_LEFT', 'joy_left' ],
    focus_right = [ 'K_RIGHT', 'joy_right' ],
    focus_up = [ 'K_UP', 'joy_up' ],
    focus_down = [ 'K_DOWN', 'joy_down' ],

    # Button.
    button_select = [ 'mouseup_1', 'K_RETURN', 'K_KP_ENTER',
'joy_dismiss' ],

    # Input.
    input_backspace = [ 'K_BACKSPACE' ],
    input_enter = [ 'K_RETURN', 'K_KP_ENTER' ],

    # Viewport.
    viewport_up = [ 'mousedown_4' ],
    viewport_down = [ 'mousedown_5' ],
    viewport_drag_start = [ 'mousedown_1' ],
    viewport_drag_end = [ 'mouseup_1' ],

    # These keys control skipping.
    skip = [ 'K_LCTRL', 'K_RCTRL', 'joy_holdskip' ],
    toggle_skip = [ 'K_TAB', 'joy_toggleskip' ],
    fast_skip = [ '>' ],

    # These control the bar.
    bar_activate = [ 'mousedown_1', 'K_RETURN', 'K_KP_ENTER',
'joy_dismiss' ],
    bar_deactivate = [ 'mouseup_1', 'K_RETURN', 'K_KP_ENTER',
'joy_dismiss' ],
    bar_decrease = [ 'K_LEFT', 'joy_left' ],
    bar_increase = [ 'K_RIGHT', 'joy_right' ],
    )
```

User-Defined Displayables

It's possible for a user to define displayables that extend Ren'Py's behavior. This section will explain the steps that are involved in defining your own displayables, and how to use them with Ren'Py.

A displayable is an object that inherits from renpy.Displayable, a class that has several methods that can be overridden to control the behavior of the displayable. Probably the most important of these is render, a method that must be overridden to return a renpy.Render object, which contains the result of drawing the displayable. The event method can be overridden to respont to events, and to terminate the interaction when necessary. Other methods allow a displayable to respond to a variety of events.

Displayables must be capable of being serialized. This means that they must not contain references to objects that are incapable of being serialized. Of particular note is that the renpy.Render object is incapable of being serialized.

Displayables have a number of methods and fields, not all of which are documented here. We strongly recommend you use a unique prefix for the names of fields and methods you add, to minimize the chance of conflicts.

renpy.Displayable

This is the base class for user-defined displayables. It has the following fields and methods.

Since you will be implementing many of these methods, rather than calling them, we include the normally hidden self parameter in the definition.

Field: **focusable** = False

This field should be True if it's possible for the displayable to gain focus, and false otherwise.

Field: **style** = ...

The style of this displayable. This is automatically created by the default constructor.

Field: **delay** = *undefined*

If this displayable is used as a transition, then delay should be set to the number of seconds that the transition will take.

Method: **renpy.Displayable.__init__** (self, default=False, focus=None, style='default', **properties):

The __init__ method may be overridden with your own parameter list, but the default method must be called to properly initialize a Displayable. The parameters the default methods take are:

focus - if given, a string giving the name of the focus group in the object. The focus group is used to determine which displayable should be focused after an interaction. For example, if the third displayable in focus group "foo" has focus at the end of an interaction, the third displayable in "foo" will have focus at the start of the next interaction, even if the two displayables are not the same.

default - determines if this object can be focused by default. This is only used if no object with the same focus name is found in an interaction.

style - The name of the default style, used to construct a style object that is placed in the `style` field.

properties - Additional keyword arguments are treated as style properties when constructing the style.

Method: **renpy.Displayable.focus** (self, default=False):

Called to indicate that this displayable has been given the focus. *default* is true if the focus has been given to this displayable at the start of an interaction.

The default implementation sets the style prefix of this displayable and its children to 'hover_'.

If this returns a non-None value, the current interaction is terminated and the value is returned.

Method: **renpy.Displayable.unfocus** (self):

Called to indicate this displayable has lost focus.

The default implementation resets the style prefix of this displayable and its children to 'idle_'.

This should return None.

Method: **renpy.Displayable.is_focused** (self):

This method determines if the current displayable has focus. It probably doesn't make sense to override it.

Method: **renpy.Displayable.set_style_prefix** (self, prefix):

Sets the style prefix of this displayable and its children to one of: "insensitive_", "idle_", "hover_", "activate_", "selected_insensitive_", "selected_idle_", "selected_hover_", or "selected_activate_".

Method: **renpy.Displayable.render** (self, width, height, st, at):

This is responsible for drawing the displayable. This is done by creating a renpy.Render object, blitting other objects to that object in appropriate places, and then returning that Render.

width, *height* - The area allocated to this displayable. The created displayable may be bigger or smaller than this, without consequence.

st - The displayable timebase, the number of seconds this displayable has been shown for.

at - The animation timebase, the number of seconds an image with the same tag as this displayable has been shown for.

This method should not be called directly on child displayables. (See renpy.render instead.)

Method: **renpy.Displayable.event** (self, ev, x, y, st):

This is called when an event occurs.

ev - The pygame event object. This can be read to get the type of event and other parameters.

x, *y* - The position of the mouse at the time of the event. This is relative to the upper-left corner of this displayable, and should be used in preference to similar data contained in *ev*.

st - The displayable timebase.

If this method returns a non-None value, then that value is returned from ui.interact. If it returns None, processing of the event is continued by the other displayables. If renpy.IgnoreEvent is raised, then no other displayables will see the event.

It may make sense to call this method on child displayables.

Method: **renpy.Displayable.get_placement** (self):

Returns a tuple of (xpos, ypos, xalign, yalign, xoffset, yoffset), that is used to place this displayable on the screen.

This defaults to getting the proper values out of style.

Method: **renpy.Displayable.visit** (self):

This should be overridden to return a list of all the child displayables fo this displayable.

Method: **renpy.Displayable.per_interact** (self):

This is called once per interaction, to inform the displayable that a new interaction has begun.

Often, this should call renpy.redraw(self, 0), to make sure that the displayable is redrawn for each interaction. This is especially useful if the displayable participates in rollback.

renpy.Render

Since one will not be subclassing renpy.Render, we omit the self parameter from method descriptions.

Function: **renpy.Render** (width, height):

Constructs a new render object with the given width and height.

Method: **renpy.Render.blit** (source, pos):

Copies the contents of the *source* render to this one. *Pos* is a 2-element tuple giving the position of the upper-left-hand corner of the source render, relative to the upper-left-hand corner of this render, in pixels.

Method: **renpy.Render.fill** (color):

Fills the render with the given color, which should be an RGBA tuple.

Method: **renpy.Render.get_size** ():

Returns the width and height of this render as a pair.

Method: **renpy.Render.kill** ():

Deallocates the memory used by this Render. This must be called on a Render returned by renpy.render or renpy.Render.subsurface if the render is not blitted to another Render. Once this is called, the render should not be blitted to a surface or have any other method called. It is safe to call this on a Render that has been blitted somewhere, as these calls are ignored.

Method: **renpy.Render.subsurface** (pos):

Returns a render that is a subsurface of this render. Pos should be an (x,y,w,h) tuple, where x and y are the coordinates of the upper-left corner of the subsurface, w is the width, and h is the height of the subsurface.

Method: **renpy.Render.add_focus** (displayable, arg, x, y, w, h, mx=None, my=None, mask=None):

This is used to indicate that a sub-region of this Render is eligible for being focused.

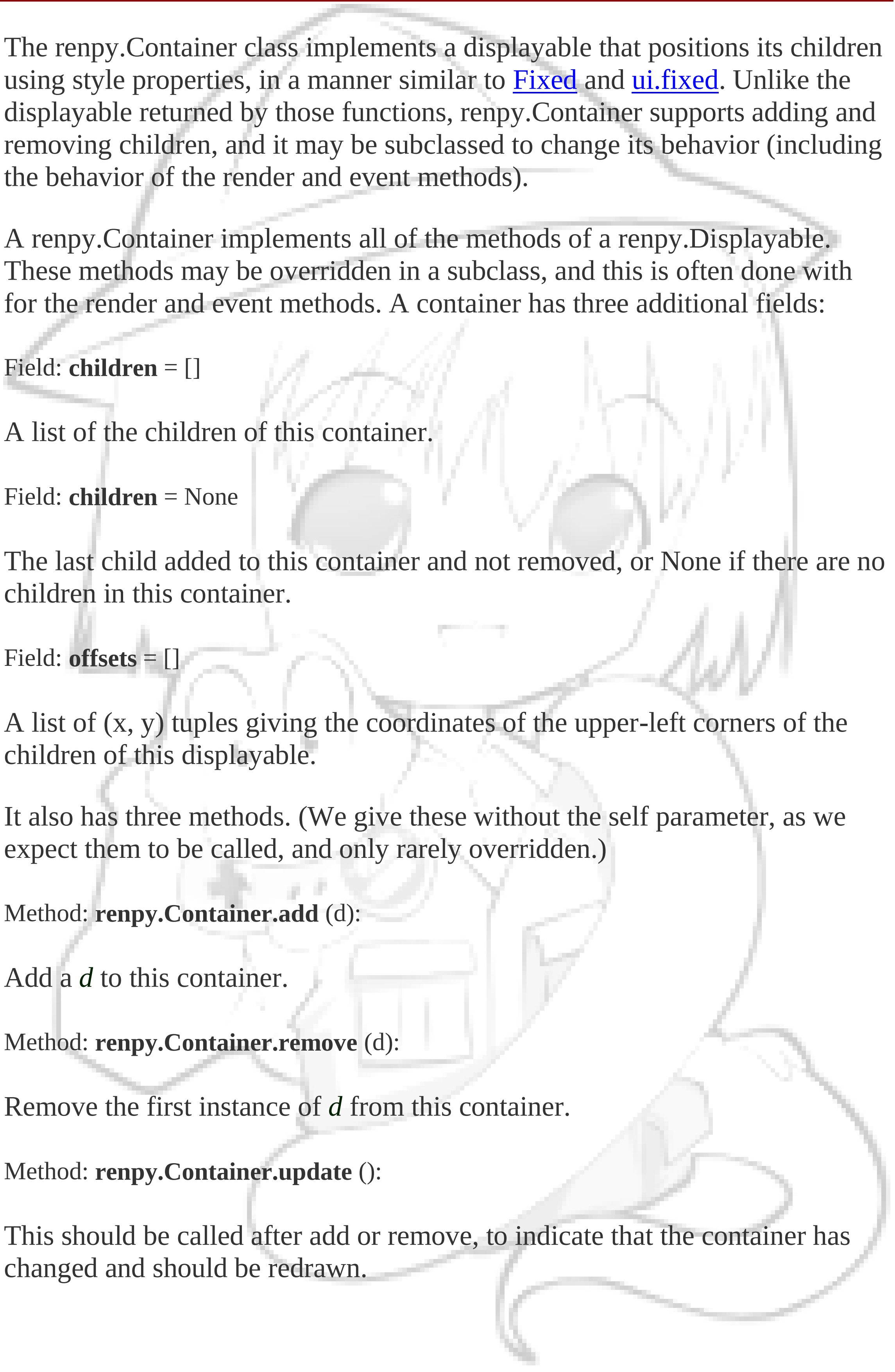
displayable - The displayable that will be focused. *arg* - Ignored, should be None. *x*, *y*, *w*, *h* - If not None, a rectangular subregion of this Render that will be focused. If all are None, this displayable has full-screen focus. *mx*, *my* - If not None, offsets of the focus mask from the upper-left hand corner of this Render. *mask* - A Render that is used as a focus mask when mouse focus is needed. The mouse will only focus the displayable when it is over a non-transparent pixel in the mask.

Method: **renpy.Render.canvas** ():

This function returns a canvas object that has methods corresponding to the pygame.draw functions, except that the first argument (the surface) is omitted. For example, instead of calling pygame.draw.line(surface, (255, 255, 255, 255), (100, 100), (200, 200)), one would call c.line((255, 255, 255, 255), (100, 100), (200, 200)).

Code that uses a canvas may not be portable to non-pygame platforms, should we choose to support them in the future.

renpy.Container



The renpy.Container class implements a displayable that positions its children using style properties, in a manner similar to Fixed and ui.fixed. Unlike the displayable returned by those functions, renpy.Container supports adding and removing children, and it may be subclassed to change its behavior (including the behavior of the render and event methods).

A renpy.Container implements all of the methods of a renpy.Displayable. These methods may be overridden in a subclass, and this is often done with for the render and event methods. A container has three additional fields:

Field: **children** = []

A list of the children of this container.

Field: **children** = None

The last child added to this container and not removed, or None if there are no children in this container.

Field: **offsets** = []

A list of (x, y) tuples giving the coordinates of the upper-left corners of the children of this displayable.

It also has three methods. (We give these without the self parameter, as we expect them to be called, and only rarely overridden.)

Method: **renpy.Container.add** (d):

Add a *d* to this container.

Method: **renpy.Container.remove** (d):

Remove the first instance of *d* from this container.

Method: **renpy.Container.update** ():

This should be called after add or remove, to indicate that the container has changed and should be redrawn.

Utility Functions

Function: **renpy.render** (displayable, width, height, st, at):

Call this function to get a Render from a displayable. The object returned from this function should be treated as immutable, as it may be stored in a cache. If an object with the same width and height has already been rendered (and has not been invalidated using the renpy.redraw), the cached render will return.

Function: **renpy.redraw** (displayable, time):

Causes the supplied displayable to be redrawn after *time* seconds have elapsed. *Time* may be zero to ensure an immediate redraw.

Function: **renpy.timeout** (time):

Causes an event to occur in *time* seconds. The type of the event is undefined.

Function: **renpy.easy_displayable** (d, none=False):

This takes an "easy displayable", and returns a Displayable. Easy displayables may be Displayables, or may be strings. If a string, then:

- If the string starts with #, a Solid of the given color is returned.
- Otherwise, an Image is returned.

If *none* is True and d is None, then None is returned. Otherwise, if d is None, an exception is thrown.

Using User-Defined Displayables

Once a user-defined displayable class has been defined, objects of that class can be created. Such an object can be shown with a show statement, or added to the transient layer with ui.add.

Defining At-functions and Transitions

An At-function is a function that is suitable for use in an at-list, as part of a show statement. These are functions that take a displayable as an argument, and return a second displayable. (User-defined or not.)

Transitions take two keyword arguments, new_widget and old_widget, representing the new and old scenes, respectively. They return a displayable that performs the transition, which should have its delay fields set to the duration of the transition.

Notes

Although we will do our best to avoid breaking code unnecessarily, we reserve the right to change the displayable API as necessary to support the future development of Ren'Py. As a result, code using the displayable API may not be forward-compatible with future versions of Ren'Py.

Python Modules

It's possible to write python modules, and import them into Ren'Py. These python modules define their own namespace, which is different from the namespace in which python-in-Ren'Py executes. Objects defined in python modules do not participate in rollback, although they may be saved if they are reachable from the main Ren'Py store.

Python modules may be placed in the game directory, while python packages are subdirectories of the game directory containing __init__.py files. The only encoding we support is utf-8.

We suggest beginning each module with:

```
import renpy.store as store
import renpy.exports as renpy
```

This makes the store, the namespace in which python-in-Ren'Py executes, available as `store`. It also makes the various functions documented in this manual available under `renpy`. (A simple `import renpy` will not do this, it will access the internals of Ren'Py instead. Sorry for the additional complexity.)

In general, objects created inside python modules do not participate in rollback. This means that you should not change such objects after their initial creation. If you want to produce an object that participates in rollback, you can use the equivalents defined in the `store` module: `store.list`, `store.dict`, `store.set`, `store.object`. You can also create a class inheriting from `store.object`, and objects of that class will participate in rollback.

It's probably easier to not change objects that are created in python modules.

Example

As a trivial example, we could have the following module, which is placed in magic.py in the game directory.

```
import renpy.store as store
import renpy.exports as renpy

def add(a, b):
    return a + b
```

In a Ren'Py script file, we can import the module using an import statement:

```
init:
    $ import magic
```

You can then call methods from the module:

```
$ result = magic.add(1, 2)

"1 + 2 = %(result)d"
```

Developer Tools

Ren'Py includes a number of features to make a developer's life easier. Many of them need the variable config.developer to be set to True to operate.

Shift+E Editor Support

The config.editor variable allows a developer to specify an editor command that is run when the launch_editor keypress (by default, shift-E) occurs.

Please see Integrating SciTE with Ren'Py for information about how to use Ren'Py with the customized SciTE text editor.

See Integrating Crimson Editor with Ren'Py for information on how to use Ren'Py with Crimson Editor.

Shift+D Developer Menu

When config.developer is true, hitting shift+D will display a developer menu that provides easy access to some of the features given below.

Shift+R Reloading

When config.developer is true, hitting shift+R will save the current game, reload the game script, and reload the game. This will often place you at the last unchanged statement encountered before shift+R was pressed.

This allows the developer to make script changes with an external editor, and not have to exit and restart Ren'Py to see the effect of the changes.

Note that game state, which includes variable values and scene lists, is preserved across the reload. This means that if one of those statements is changed, it is necessary to rollback and re-execute the statement to see its new effect.

Shift+I Style Inspecting

When config.developer is true, pressing shift+I will cause style inspection to occur. This will display a list of displayables underneath the mouse. For each displayable, it will display the type, the style used, and the size it is being rendered at.

Shift+Y Style Dumping

When config.developer is True, pressing the dump_styles key (by default, shift-Y), will write a description of every style Ren'Py knows about to the file `styles.txt`. This description includes every property that is part of the style, the value of that property, and the style the property is inherited from.

> Fast Skipping

When config.developer is True, pressing the fast_skip key (by default, '>') causes the the game to immediately skip to the next important interaction. For this purpose, an important interaction is one that is not caused by a say statement, transition, or pause command. Usually, this means skipping to the next menu, but it will also stop when user-defined forms of interaction occur.

Warping to a Line

Ren'Py supports warping to a line in the script, without the developer to play through the entire game to get there. While this warping technique has a number of warnings associated with it, it still may be useful in providing a live preview.

To invoke warping, run Ren'Py with the `--warp` command-line argument followed by a filename:line combination, to specify where you would like to warp to. For example:

```
run_game --warp script.rpy:458
```

When warping is invoked, Ren'Py does a number of things. It first finds all of the scene statements in the program. It then tries to find a path from the scene statements to every reachable statement in the game. It then picks the reachable statement closest to, but before or at, the given line. It works backwards from that statement to a scene statement, recording the path it took. Ren'Py then executes the scene statement and any show or hide statements found along that path. Finally, it transfers control to the found statement.

There are a number of fairly major caveats to the warp feature. The first is that it only examines a single path, which means that while the path may be representative of some route of execution, it's possible that there may be a bug along some other route. In general, the path doesn't consider game logic, so it's also possible to have a path that isn't actually reachable. (This is only really a problem on control-heavy games, espcially those that use a lot of python code.

The biggest problem, though, is that no python code is executed before the statement that is warped to. This means that all variables will be uninitalized, which can lead to crashes when they are used. To overcome this, one can define a label `after_warp`, which is called after a warp but before the warped-to statement executes. The code reached by this label can set up variables in the program, and then return to the preview.

The warp feature requires config.developer to be True to operate.

When using renpy.sh or renpy.py from the SDK to run Ren'Py, you will also need to give the *--game* option. For example:

```
./renpy.sh --game ./the_question/game/ --warp script.rpy:98
```

NVL Mode

NVL mode is a mode in which Ren'Py shows more than one line of dialogue on the screen at once. This is used to better emulate the Japanese visual novel format, whereas the default settings of Ren'Py emulate the Japanese adventure format.

To use NVL-mode, one must declare Characters with a kind of `nvl`, or with a kind that has a kind of `nvl`. For example:

The narrator can also be made an NVLCharacter, by setting the narrator variable to an NVLCharacter with the name of None.

```
init:
    $ e = Character("Eileen", color="#c8ffc8", kind=nvl)
    $ narrator = NVLCharacter(None, kind=nvl)
```

When dialogue is spoken with an NVL-mode Character, it is added to the end of a buffer, and then the entire contents of that buffer are shown to the user. The buffer can be cleared using the `nvl clear` statement. It should be cleared before the first use.

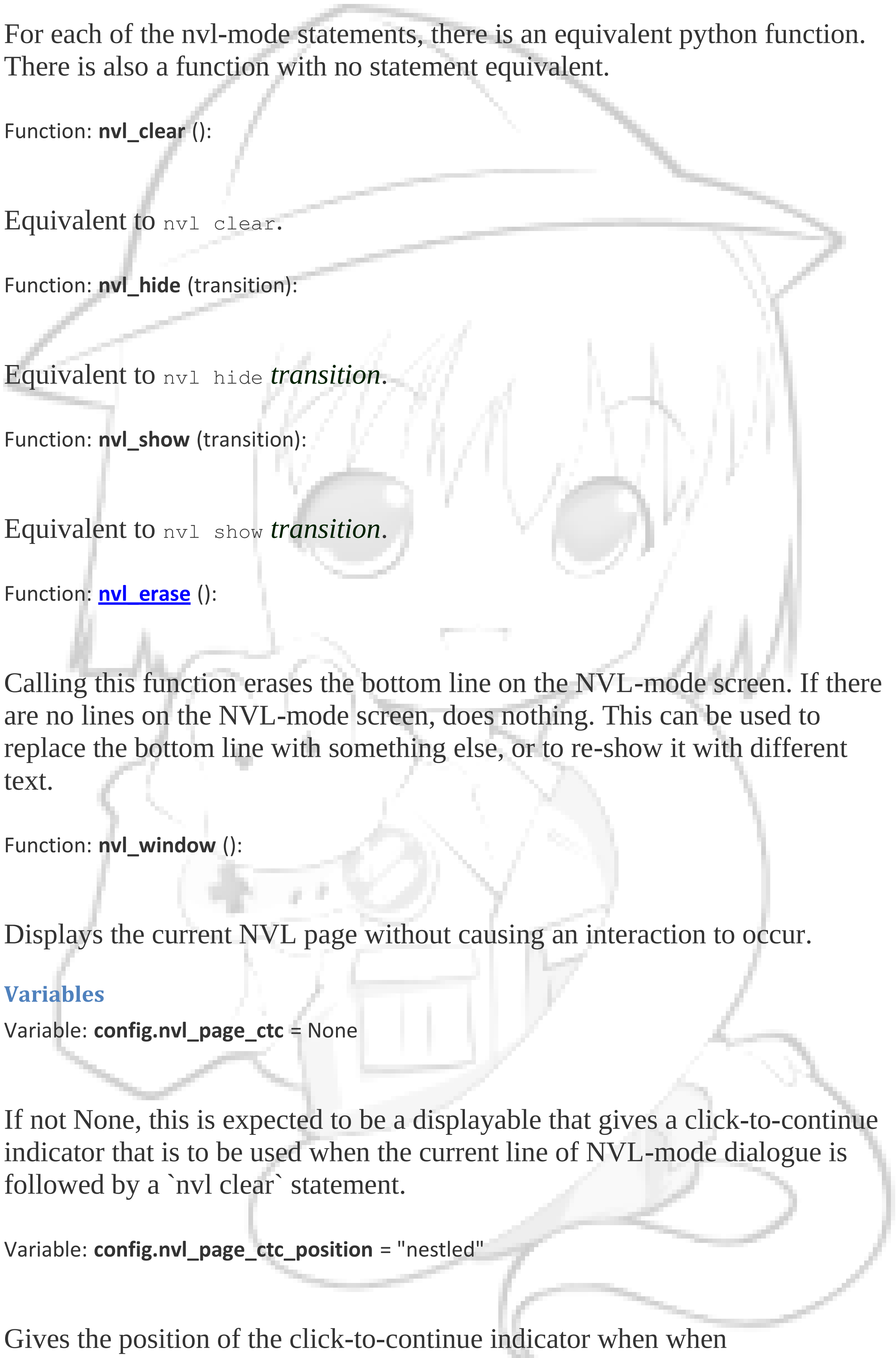
```
nvl clear

e "This is the first line of the first screen."

e "This is the second line of the first screen."

nvl clear

e "This is the first line of the second screen."

e "This is the second line of the second screen."
```

The `nvl show` *transition* statement causes a transition to occur from the previous screen to a screen with the buffer shown. The `nvl hide` *transition* statement causes a transition to occur from a screen with the buffer shown to a screen without the buffer shown.

Menus. The `nvl_menu` function shows a menu in NVL-mode. To use it, it should be assigned to `menu` in an init block, using code like:

```
init:
    $ menu = nvl_menu
```

As nvl-mode menus are shown without being added to the buffer, we suggest clearing the buffer (using `nvl clear`) immediately after each menu.



Functions

For each of the nvl-mode statements, there is an equivalent python function. There is also a function with no statement equivalent.

Function: **nvl_clear** ():

Equivalent to `nvl clear`.

Function: **nvl_hide** (transition):

Equivalent to `nvl hide` *transition*.

Function: **nvl_show** (transition):

Equivalent to `nvl show` *transition*.

Function: **nvl_erase** ():

Calling this function erases the bottom line on the NVL-mode screen. If there are no lines on the NVL-mode screen, does nothing. This can be used to replace the bottom line with something else, or to re-show it with different text.

Function: **nvl_window** ():

Displays the current NVL page without causing an interaction to occur.

Variables

Variable: **config.nvl_page_ctc** = None

If not None, this is expected to be a displayable that gives a click-to-continue indicator that is to be used when the current line of NVL-mode dialogue is followed by a `nvl clear` statement.

Variable: **config.nvl_page_ctc_position** = "nestled"

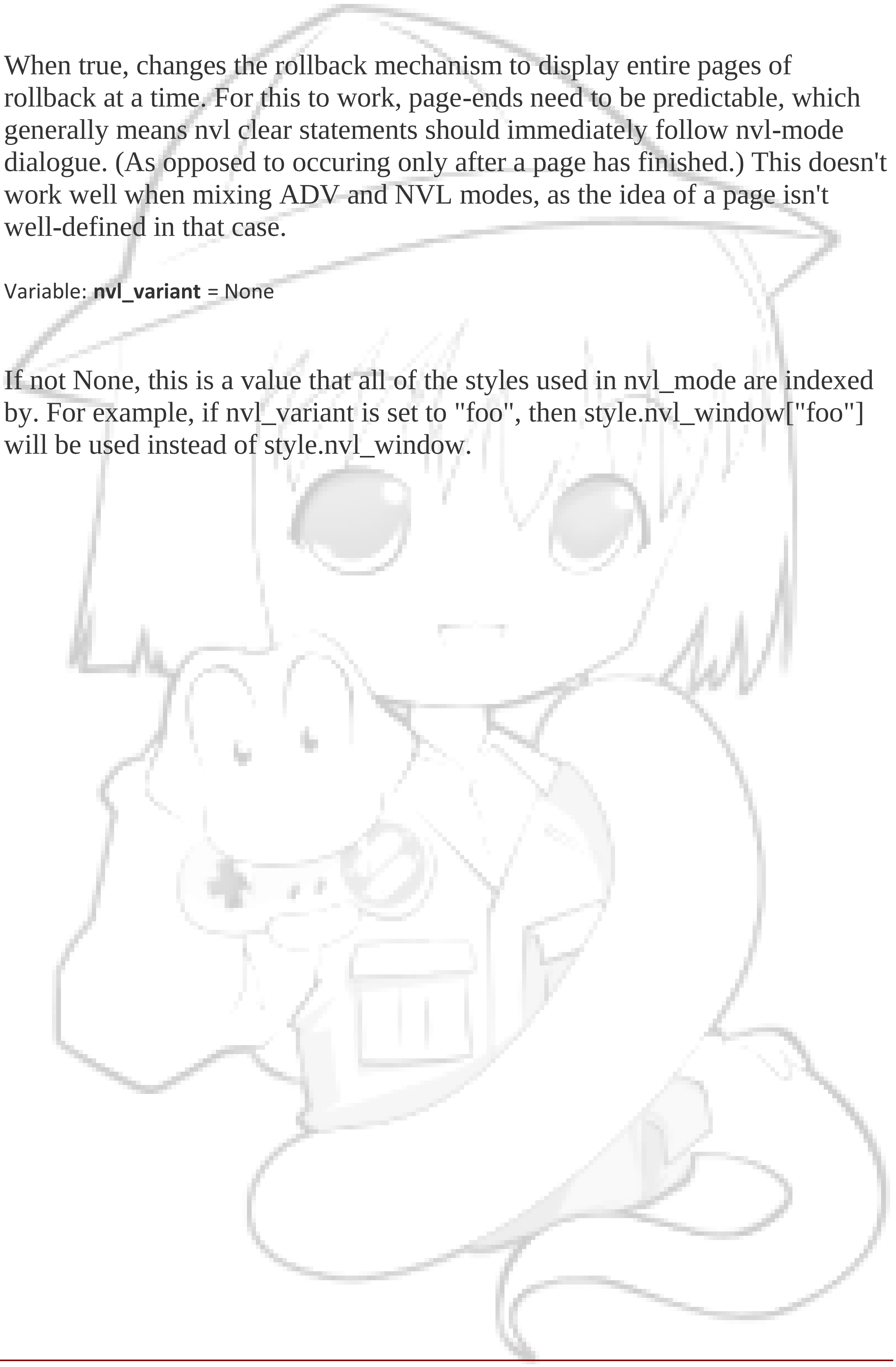
Gives the position of the click-to-continue indicator when when config.nvl_page_ctc is used. See the *ctc_position* argument to Character for legal values.

Variable: **config.nvl_paged_rollback** = False

When true, changes the rollback mechanism to display entire pages of rollback at a time. For this to work, page-ends need to be predictable, which generally means nvl clear statements should immediately follow nvl-mode dialogue. (As opposed to occuring only after a page has finished.) This doesn't work well when mixing ADV and NVL modes, as the idea of a page isn't well-defined in that case.

Variable: **nvl_variant** = None

If not None, this is a value that all of the styles used in nvl_mode are indexed by. For example, if nvl_variant is set to "foo", then style.nvl_window["foo"] will be used instead of style.nvl_window.



Voice Support

Ren'Py includes support for adding voice acting to a game. This is done through two additional statements.

The `voice` "*filename*" statement plays the voice file given in *filename*.

The `voice sustain` statement continues playing the previous voice file.

A voice file is played at the start of the first interaction following a voice statement. It is terminated at the start of the next interaction, unless a `voice sustain` statement is given that continues playing it.

```
voice "eileen1.ogg"
e "I'm now saying something."

"But she stopped saying it, because this statement started."

voice "eileen2.ogg"
e "Now I'm saying something else..."

voice sustain
e "And I'll keep saying it!"
```

Function Equivalents. There are two functions corresponding to the two voice statements.

Function: **voice** (filename):

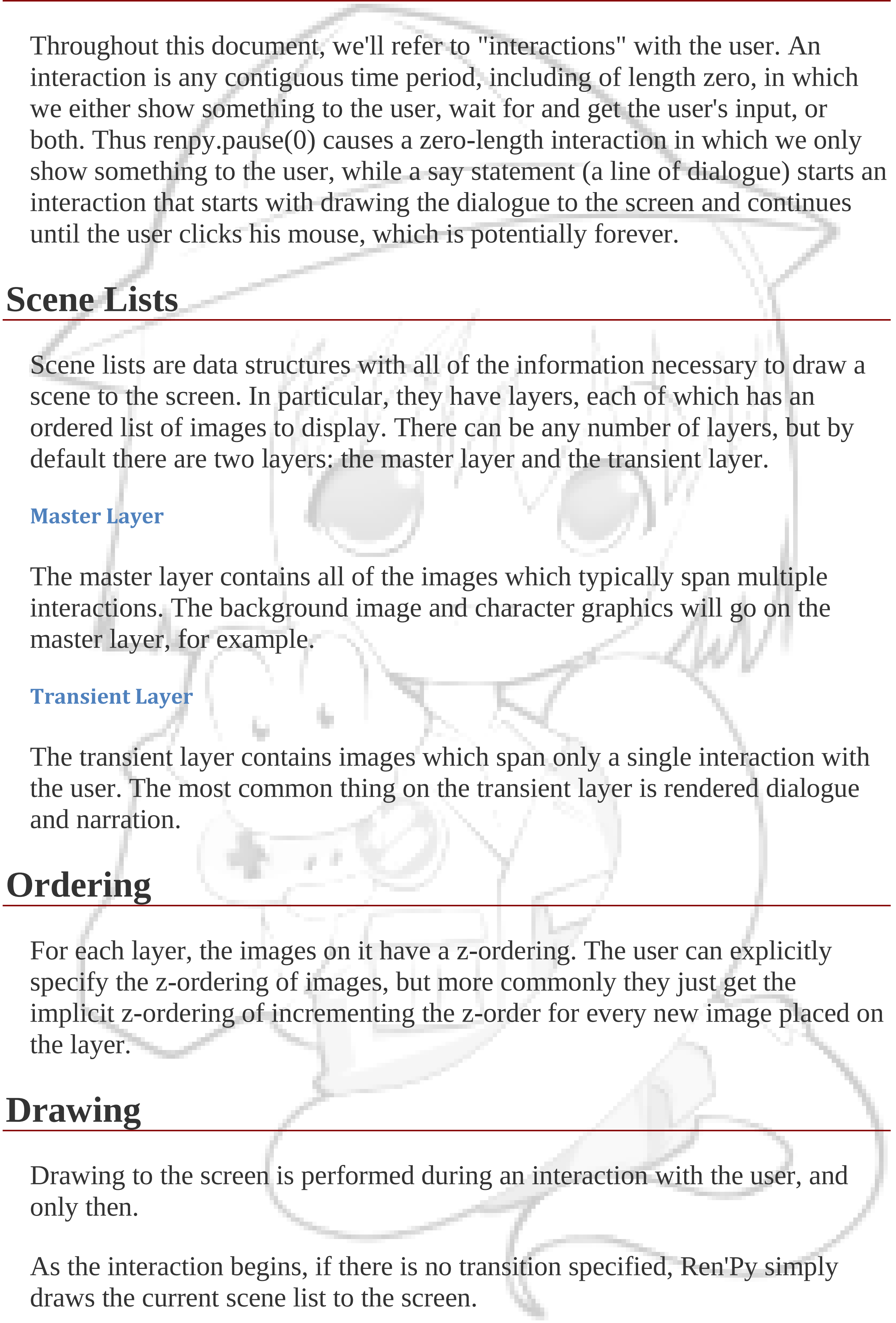
Plays the voice in *filename*. Equivalent to `voice`.

Function: **voice_sustain** ():

Continues playing the current voice file. Equivalent to `voice sustain`.

Rendering Model

There are three main entities in the Ren'Py rendering model: the current scene list, the old scene list, and the screen. The first two need a bit of explanation, but the screen is just the thing rendered on the user's monitor.



Interactions

Throughout this document, we'll refer to "interactions" with the user. An interaction is any contiguous time period, including of length zero, in which we either show something to the user, wait for and get the user's input, or both. Thus renpy.pause(0) causes a zero-length interaction in which we only show something to the user, while a say statement (a line of dialogue) starts an interaction that starts with drawing the dialogue to the screen and continues until the user clicks his mouse, which is potentially forever.

Scene Lists

Scene lists are data structures with all of the information necessary to draw a scene to the screen. In particular, they have layers, each of which has an ordered list of images to display. There can be any number of layers, but by default there are two layers: the master layer and the transient layer.

Master Layer

The master layer contains all of the images which typically span multiple interactions. The background image and character graphics will go on the master layer, for example.

Transient Layer

The transient layer contains images which span only a single interaction with the user. The most common thing on the transient layer is rendered dialogue and narration.

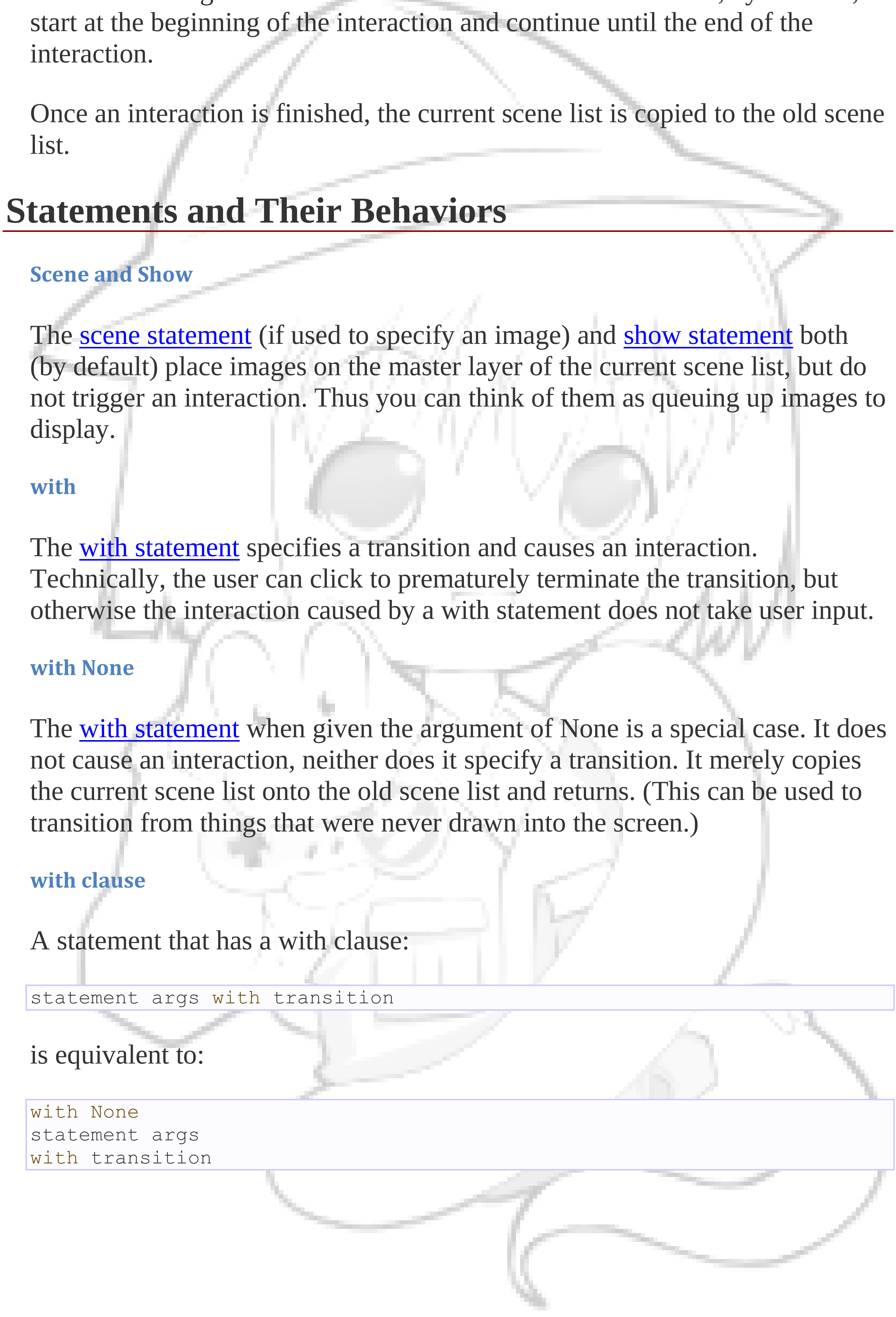
Ordering

For each layer, the images on it have a z-ordering. The user can explicitly specify the z-ordering of images, but more commonly they just get the implicit z-ordering of incrementing the z-order for every new image placed on the layer.

Drawing

Drawing to the screen is performed during an interaction with the user, and only then.

As the interaction begins, if there is no transition specified, Ren'Py simply draws the current scene list to the screen.



If a transition is specified, Ren'Py starts the transition at the beginning of the interaction and goes until the transition is finished. Animations, by contrast, start at the beginning of the interaction and continue until the end of the interaction.

Once an interaction is finished, the current scene list is copied to the old scene list.

Statements and Their Behaviors

Scene and Show

The scene statement (if used to specify an image) and show statement both (by default) place images on the master layer of the current scene list, but do not trigger an interaction. Thus you can think of them as queuing up images to display.

with

The with statement specifies a transition and causes an interaction. Technically, the user can click to prematurely terminate the transition, but otherwise the interaction caused by a with statement does not take user input.

with None

The with statement when given the argument of None is a special case. It does not cause an interaction, neither does it specify a transition. It merely copies the current scene list onto the old scene list and returns. (This can be used to transition from things that were never drawn into the screen.)

with clause

A statement that has a with clause:

```
statement args with transition
```

is equivalent to:

```
with None
statement args
with transition
```

And behaves exactly like that.

say

The say statement causes an interaction which last until the user terminates it (with the keyboard or mouse), except if auto-reading is enabled, in which case the interaction lasts for the auto-read duration. In particular, anything which was shown prior to a say statement will be displayed at the start of the say statement.

User-Defined Statements

User-defined statements allow you to add your own statements to Ren'Py. This makes it possible to add things that are not supported by the current syntax of Ren'Py. User-defined statements are currently limited to a single line, and may not contain blocks.

User-defined statements must be defined in a python early block. What's more, the filename containing the user-defined statement must be be loaded earlier than any file that uses it. Since Ren'Py loads files in unicode sort order, it generally makes sense to prefix the name of any file containing a user-defined statement with 00. A user-defined statement cannot be used in the file in which it is defined.

Function: **renpy.statements.register** (name, parse=..., execute=..., predict=..., lint=..., next=...):

This registers a user-defined statement.

name is either a space-separated list of names that begin the statement, or the empty string to define a new default statement (the default statement will replace the say statement).

parse is a function that takes a lexer object. This function should parse the statement, and return an object. This object is passed as an argument to all the other functions. The lexer argument has the following methods:

- l.eol() - True if we are at the end of the line.
- l.match(re) - Matches an arbitrary regexp string.
- l.keyword(s) - Matches s.
- l.name() - Matches any non-keyword name. Note that this only counts built-in keywords.
- l.word() - Matches any word, period.
- l.string() - Matches a renpy string.
- l.integer() - Matches an integer, returns a string containing the integer.
- l.float() - Matches a floating point number.
- l.simple_expression() - Matches a simple python expression, returns it as a string.
- l.rest() - Skips whitespace, then returns the rest of the line.
- l.checkpoint() - Returns an opaque object representing the current point in the parse.
- l.revert(o) - Given the object returned by l.checkpoint(), returns there.

execute is a function that is called when the statement executes. It is passed a single object, the argument returned from parse.

predict is a function that is called to predict the images used by the statement. It is passed a single object, the argument returned from parse. It should return a list of displayables used by the statement.

lint is called to check the statement. It is passed a single object, the argument returned from parse. It should call renpy.error to report errors.

next is called to determine the next statement. It is passed a single object, the argument returned from parse. It should either return a label, or return None if execution should continue to the next statement.

Lint Utility Functions

These functions are useful in writing lint functions.

Function: **renpy.lint.check_text_tags** (s):

Checks the text tags in s for correctness. Returns an error string if there is an error, or None if there is no error.

Example

The creates a new statement "line" that allows lines of text to be specified without quotes.

```
python early:

    def parse_smartline(lex):
        who = lex.simple_expression()
        what = lex.rest()
        return (who, what)

    def execute_smartline(o):
        who, what = o
        renpy.say(who, what)

    def lint_smartline(o):
        who, what = o
        try:
            eval(who)
        except:
            renpy.error("Character not defined: %s" % who)

        tte = renpy.lint.check_text_tags(what)
        if tte:
            renpy.error(tte)
    renpy.statements.register("l", parse=parse_smartline,
execute=execute_smartline, lint=lint_smartline)
```

This is used like:

```
    line e "These quotes will show up," Eileen said, "and don't need
to be backslashed."
```

Environment Variables

The following environment variables control the behavior of Ren'Py:

RENPY_SCALE_FACTOR
: If set, this is parsed as a floating point number, and the display screen is scaled by that amount. For example, if RENPY_SCALE_FACTOR is set to "0.5", everything is half normal size.

RENPY_SCALE_FAST
: If set and Ren'Py starts scaling the display screen, the display screen will use nearest-neighbor filtering rather than slower but higher-quality bilinear filtering. It should generally be unnecessary to set this.

RENPY_DISABLE_JOYSTICK
: If set, joystick detection is disabled. Use this if a faulty joystick is causing Ren'Py to advance when not desired.

RENPY_DISABLE_FULLSCREEN
: If set, Ren'Py will refuse to go into fullscreen mode.

RENPY_DISABLE_SOUND
: This prevents sound playback from occuring. If this variable contains "pss", sound playback will be disabled. If it contains "mixer", volume control will be disabled. A value of "pss,mixer" will disable both.

RENPY_SOUND_BUFSIZE
: This controls the sound buffer size. Values larger than the default (2048) can prevent sound from skipping, at the cost of a larger delay from when a sound is invoked to when it is played.

RENPY_NOMIXER
: If set, prevents Ren'Py from trying to control the system audio mixer.

RENPY_EDITOR
: The default value of config.editor.

RENPY_EDITOR_FILE_SEPARATOR
: The default value of config.editor_file_separator.

RENPY_EDITOR_TRANSIENT
: The default value of config.editor_transient.

RENPY_SCREENSHOT_PATTERN
: A pattern used to create screenshot filenames. It should contain a single %d substitution in it. For example, setting this to "screenshot%04d.jpg" will cause Ren'Py to write out jpeg screenshots rather than the usual pngs.

RENPY_LESS_MEMORY
: This causes Ren'Py to reduce its memory usage, in exchange for reductions in speed.

RENPY_LESS_UPDATES
: This causes Ren'Py to reduce the number of screen updates that occur.

RENPY_LESS_MOUSE

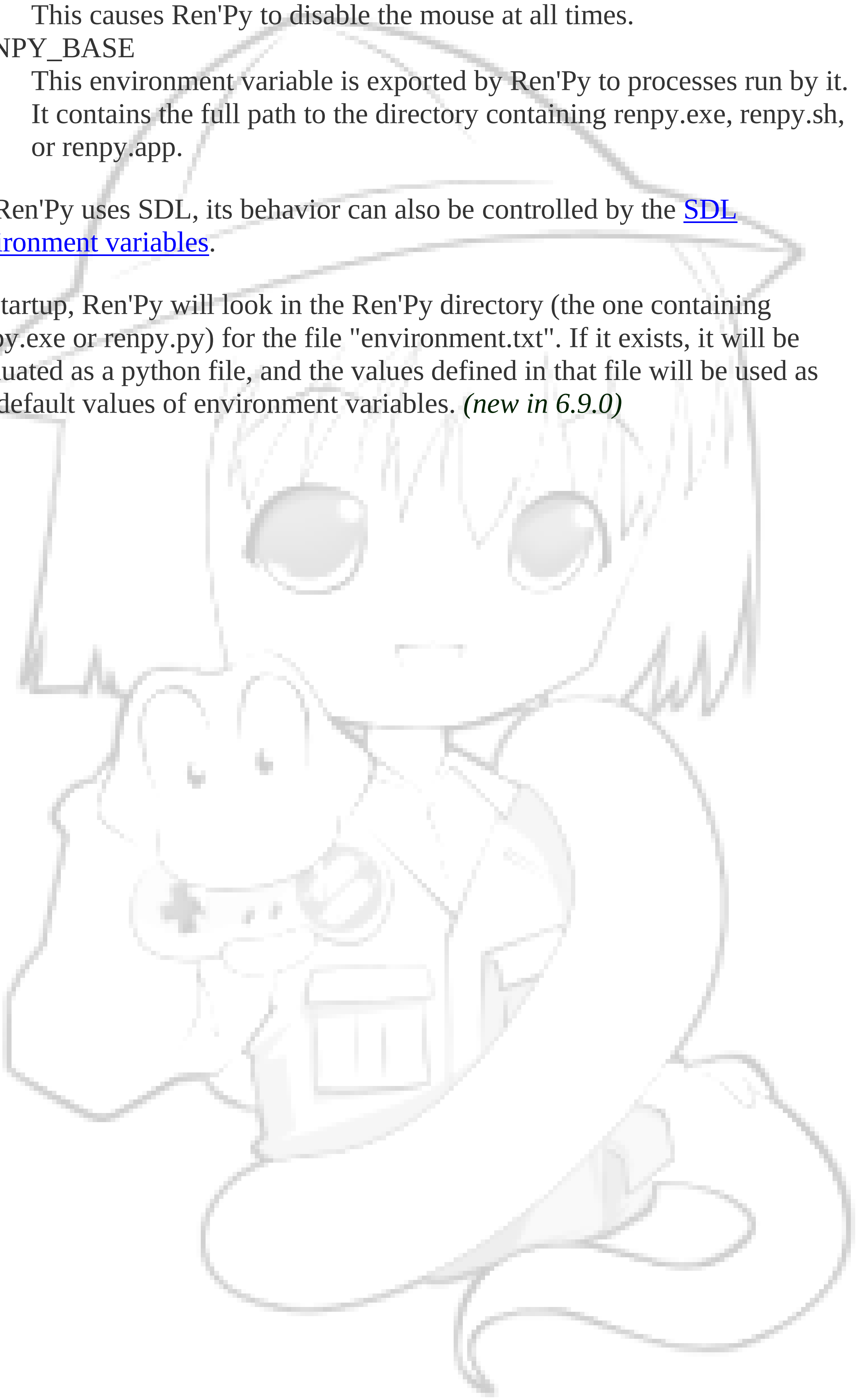
This causes Ren'Py to disable the mouse at all times.

RENPY_BASE

This environment variable is exported by Ren'Py to processes run by it. It contains the full path to the directory containing renpy.exe, renpy.sh, or renpy.app.

As Ren'Py uses SDL, its behavior can also be controlled by the SDL environment variables.

At startup, Ren'Py will look in the Ren'Py directory (the one containing renpy.exe or renpy.py) for the file "environment.txt". If it exists, it will be evaluated as a python file, and the values defined in that file will be used as the default values of environment variables. *(new in 6.9.0)*



Function Index

Note that due to a limitation of mediawiki, underscores are replaced by spaces on this page.

- renpy.hide
- renpy.image
- renpy.imagemap
- renpy.input
- renpy.invoke in new context
- renpy.in rollback
- renpy.jump
- renpy.jump out of context
- renpy.last interact type
- renpy.layer at list
- renpy.list saved games
- renpy.load
- renpy.loadable
- renpy.load module
- renpy.log
- renpy.movie cutscene
- renpy.movie start displayable
- renpy.movie stop
- renpy.music.get playing
- renpy.music.play
- renpy.music.queue
- renpy.music.register channel
- renpy.music.set mixer
- renpy.music.set music
- renpy.music.set pan
- renpy.music.set queue empty callback
- renpy.music.set volume
- renpy.music.stop
- renpy.partial
- renpy.pause
- renpy.play
- renpy.pop return
- renpy.predict
- renpy.predict display say
- renpy.quit
- renpy.random.choice
- renpy.random.randint
- renpy.random.random
- renpy.redraw
- renpy.register bmfont
- renpy.register mudgefont
- renpy.register sfont
- renpy.rename save
- renpy.render

- renpy.Render (constructor)
- renpy.reshow say
- renpy.restart interaction
- renpy.rollback
- renpy.roll forward info
- renpy.save
- renpy.scan saved game
- renpy.scene
- renpy.seen audio
- renpy.seen image
- renpy.seen label
- renpy.show
- renpy.showing
- renpy.shown window
- renpy.show display say
- renpy.sound.is playing
- renpy.sound.play
- renpy.sound.queue
- renpy.sound.set mixer
- renpy.sound.set pan
- renpy.sound.set volume
- renpy.sound.stop
- renpy.take screenshot
- renpy.timeout
- renpy.transition
- renpy.unlink save
- renpy.watch
- renpy.with statement
- Revolve
- RotoZoom
- ShowingSwitch
- SizeZoom
- SnowBlossom
- Solid
- SplineMotion
- Style
- Style.clear
- style.create
- style.rebuild
- Style.set parent
- Style.take
- Text
- theme.ancient
- theme.image buttons

- theme.image labels
- theme.outline
- theme.roundrect
- theme.roundrect red
- Transform
- ui.add
- ui.adjustment
- ui.at
- ui.autobar
- ui.bar
- ui.button
- ui.callsinnewcontext
- ui.clear
- ui.close
- ui.detached
- ui.fixed
- ui.frame
- ui.gamemenus
- ui.grid
- ui.hbox
- ui.image
- ui.imagebutton
- ui.imagemap
- ui.input
- ui.interact
- ui.jumps
- ui.jumpsoutofcontext
- ui.keymap
- ui.layer
- ui.menu
- ui.null
- ui.pausebehavior
- ui.remove
- ui.returns
- ui.saybehavior
- ui.side
- ui.sizer
- ui.soundstopbehavior
- ui.tag
- ui.text
- ui.textbutton
- ui.timer
- ui.transform
- ui.vbox

- ui.viewport
- ui.window
- VBox
- Viewport
- Window
- Zoom