12.1 控件介绍

在Kivy中控件是GUI的基础表现形式.它提供了一个可以在窗口上绘制的画布.它接收事件信息同时做出相应的动作.关于控件的详细解释请查阅控件章节的手册.

12.2 控件树的操作

在kivy中控件以树(tree)的形式存在.应用都有一个根控件,它下面有子控件,子控件还有子控件..子控件有其对应的属性列表(Kivy Listproperty).

控件树有以下三种操作:

1.add\_widget():添加一个子控件.

2.remove\_widget():在子控件列表中移除相应控件.

3.clear\_widget():清除所有子控件.

例如想要在BoxLayout 中添加一个Button控件可以这样:

layout = BoxLayout(padding=10)

button = Button(text='My first button')

layout.add\_widget(button)

button 添加到了layout上:button的父类属性指定为是layout,layout也添加了button到自己的子类列表.要移除button可以这样:

layout.remove\_widget(button)

当移除button,button对应的父类属性随之变成了None,同事layout的子类列表中也移除了button

如果想清空所有的子控件用如下方法:

layout.clear\_widget()

警告:(这个地方原文是有一个红色框框的, 排版请注意!)请不要改变子类列表,除非你真的能保证万无一失,因为控件树和图形树是相互关联的.当你添加一个控件到子类列表中但是并没有对应的画布到图形树中,那么这个控件在对应的列表中,但是画面上不会有对应的显示.更重要的是以后调用相应的add\_widget() remove\_widget() clear\_widget()方法可能会出现一些未知错误. layout = BoxLayout(padding=10)

button=Button(text='My first button') layout.add\_widget(button)

12.3 遍历控件树

控件对象的子类属性列表包含了所有的子类.可以很方便的遍历控件树:

root = BoxLayout()

# 遍历root上添加控件

for child in root.children:

print(child)

注意这么用的时候要非常的小心.就像前几章的那些方法一样,如果要想修改子类列表,你必须复制一个列表例子如下:

for child in root.children[:]:

# 清除所有宽度小鱼100 的控件.

if child.width < 100:

root.remove\_widget(child)

默认情况下控件不会影响子控件的大小和位置(size/pos), POS属性是屏幕坐标的绝对位置(除非用了relativelayout),size是一个绝对的大小.

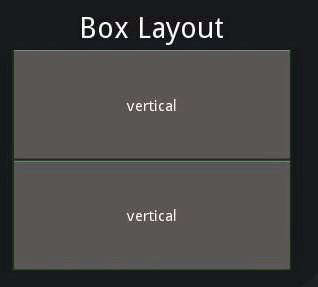
12.4 控件的z轴(z-index)位置

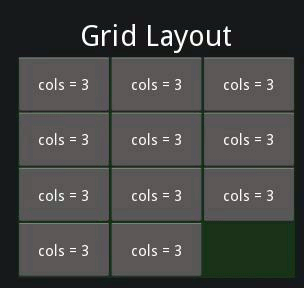
绘制控件的顺序是依据控件树的,绘制顺序是后添加后绘制(包含父控件),添加一个控件可以添加一个z index(设置在树中的位置)

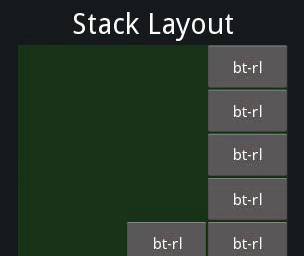
root.add\_widget(widget, index)

12.5 组织布局(Organize with Layouts)

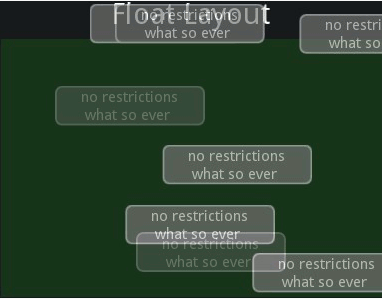
Layout 是一个控制子控件大小和位置的特殊控件. 多种不同种类的布局支持多种自动组织子类布局. Layout 有size\_hint 和pos\_hint 属性来决定子类的size和pos.











BoxLayout:以相邻的方式来安排控件(无论水平还是垂直)使其填满空间. size\_hint 属性能用来改变每一个子控件的比例,或者对某些子控件设置一个合适的size.

GridLayout:以网格的方式排布控件.至少需要指定一个网格的纬度,然后kivy可以计算出各个元素的size然后把它们安置在屏幕上.

StackLayout:按照设定防止一个一个的排布控件.设置好其中一个维度的size,但是不需要填满整个空间.用这个来显示预定义子控件的size时非常有用.

AnchorLayout:仅需要知道子控件位置的简单布局.可以将子控件放在相对于当前布局的边界的位置.size\_hint这时是不起作用的.

FloatLayout:允许子控件有任意的大小并且可以放在任意的位置,无论是绝对的或者是相对的layout size.默认的 size\_hint(1,1)能使得每一个子控件都和整个layout的size相同,所以当你有多个子控件的时候就需要修改这个数值了

你可以设置size\_hint（None，None）来使用绝对尺寸.这个控件也遵循 pos\_hint,post\_hint以字典的形式来设置和layout位置的相关性.

RelativeLayout:作用方法如同FloatLayout,除了子控件的位置是相对布局,并不包括屏幕(screen)

查看布局的相关文档以加深了解.

size\_hint和pos\_hint:

floatlayout

boxlayout

gridlayout

stack layout

relativelayout

anchorlayout

size\_hint是ReferenceListProperty类型有size\_hint\_x和size\_hint\_y.它的值是0-1或者是None,默认的是(1,1).这也就是说明如果控件在布局范围内的话,当前布局会尽可能的在两个方向上分配更大的空间(和layout size相关).

举例来说设置size\_hint为(0.5, 0.8)这样使得控件在当前的布局下面占有效布局宽度的50%有效高度的80%.

例子如下:

BoxLayout: Button:

text: 'Button 1'

# 默认的就是(1,1)其实不用写出来的不过写出来会更加明了

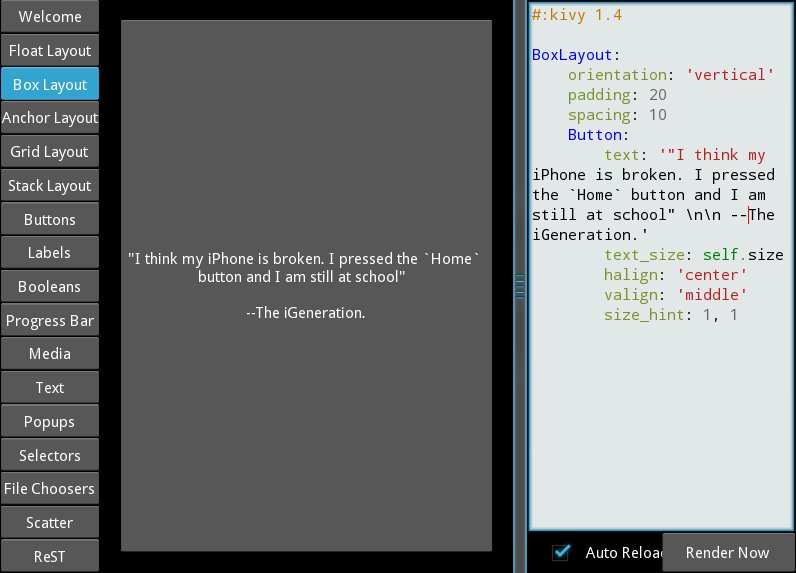
size\_hint: 1, 1

加载 kivy catalog:

cd $KIVYDIR/examples/demo/kivycatalog

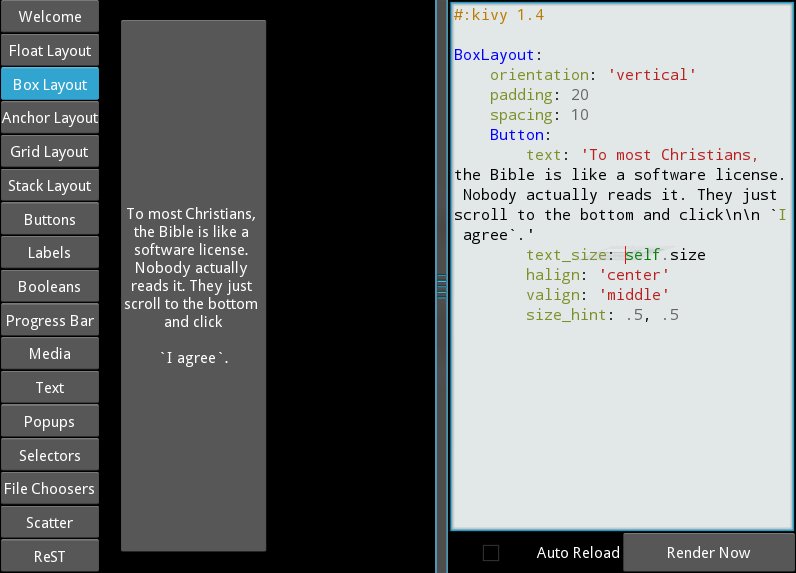
python main.py

用你的安装路径替换KIVYDIR(译者注:这里可以去github上下载demo运行安装的kivy不一定带有demo),点击左面的Box Layout,现在粘贴上面的代码到代码框里面然后就能在面板上看到相应的结果了.



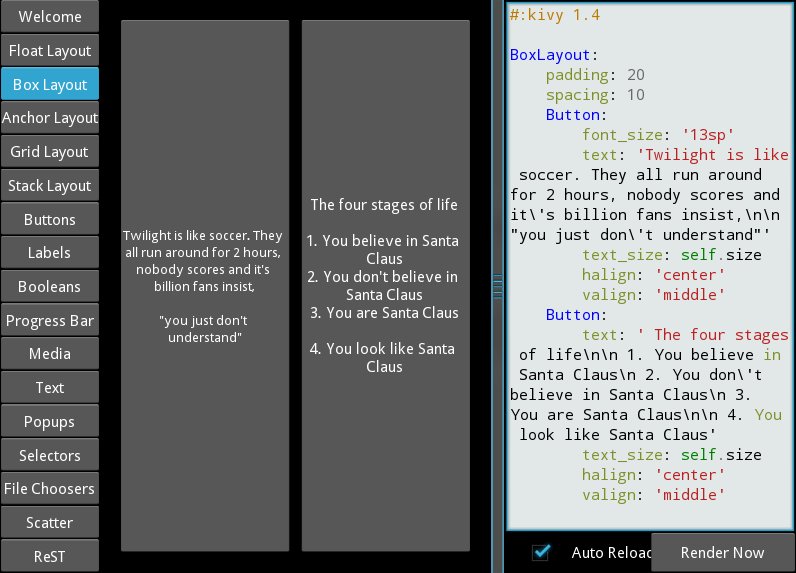
可以看到上面的照片显示出 button 占据了layout size大小的100%.

改变size\_hint\_x/size\_hint\_y 为0.5 这样控件就将占据50%的layout width/height

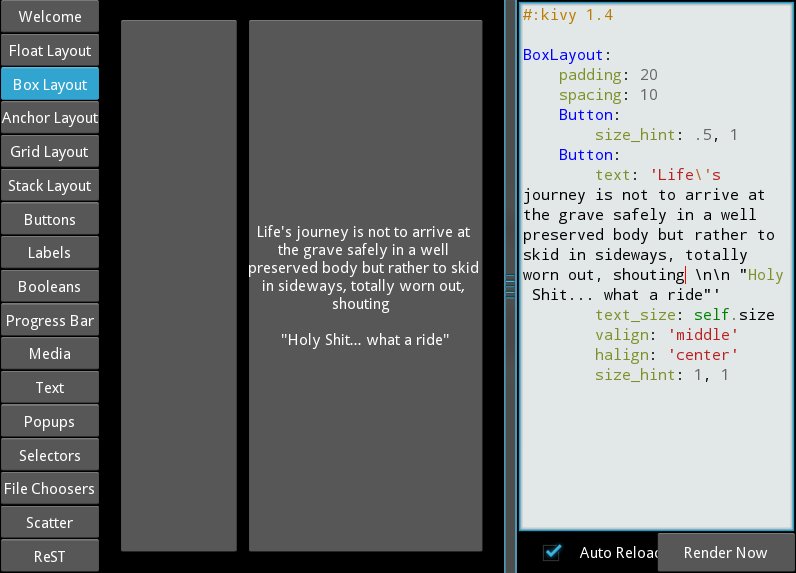


从上面的图片可以看出虽然你设置了size\_hint\_x 和 size\_hint\_y 但是貌似只有size\_hint\_x执行了,这是因为boxlayout会在你设定orientation(方向)为vertical(竖直)的时候控制size\_hint\_y,当orientation设置为’horizontal’时box layout会控制size\_hint\_x.被控制的维度(指的是size\_hint\_x或者是size\_hint\_y)是通过对boxlayout内全部的子控件计算获得的.在上图例子中,其中的子控件的size\_hint\_y是被控制的(.5/.5 = 1),这样计算起来这个子控件就是占据了layout的整个高度(也就是根据公式的来的结果).

让我恩么呢看看另外一个button在对应layout下的表现吧!



boxlayout很自然的将空间分给了子控件,在上图例子中,是55开的分割方法,因为只有两个子控件.下面来设置一下子控件的size\_hint然后再来看看对应的结果吧!



子控件指定了size\_hint之后,这就指定了boxlayout如何去分配空间给对应的控件.在我们的例子中,第一个button分配了.5给size\_hint\_x,那么计算控件的方法就像下面这样:

first child's size\_hint divided by

first child's size\_hint + second child's size\_hint + ...n(no of children)

.5/(.5+1) = .333…(译者注:这地方要注意一下是不是设置了orientation)

剩下的宽度依照剩下的子控件的size\_hint进行分配.在我们的例子意味着剩下的66.66%的layout 宽度都讲分配给第二个button.

不断的尝试size\_hint去寻找一个合适的宽度.

如果你想控制一个小部件的绝对尺寸,你可以设置 size\_hint\_x/size\_hint\_y 或者是两者都设置成None,这样控件的宽高属性就会起作用.

pos\_hint是一个字典,默认是空.相对比size\_hint,pos\_hint的布局规则有所不同,但是可以往pos的属性里面添加多种属性(x, y, left, top, center\_x, center\_y),通过他的父类确定控件的位置..

让我们在kivycatalog中输入如下的代码来看看视觉效果:

FloatLayout:

Button:

text: "We Will"

pos: 100, 100

size\_hint: .2, .4

Button:

text: "Wee Wiill"

pos: 200, 200

size\_hint: .4, .2

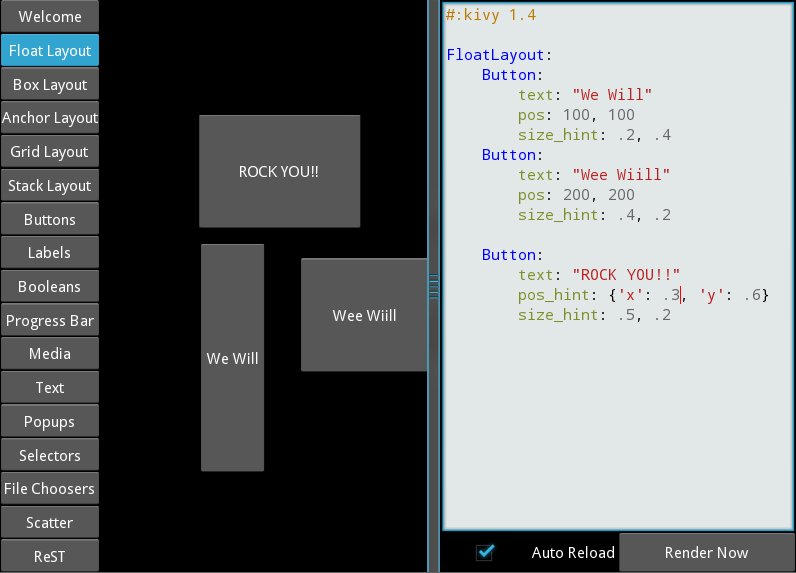
Button:

text: "ROCK YOU!!"

pos\_hint: {'x': .3, 'y': .6}

size\_hint: .5, .2

执行结果如图所示:



就像size\_hint一样,你需要用实验的方法来对pos\_hint来加深理解.(译者注:可以设置pos属性来确定位置这里面的坐标系是数学坐标系).

12.6 给布局(layout)添加一个背景(Background)

一个关于布局最常见的问题是:

如何添加一个 背景图片/背景颜色/背景视频 到 一个布局上.

布局因为其性质是不可见的:他们没有布局指令.不过你能往一个布局对象上轻松添加布局指令,下面就添加一个背景颜色:

python中:

from kivy.graphics import Color, Rectangle

with layout\_instance.canvas.before:

Color(0, 1, 0, 1) # green; colors range from 0-1 instead of 0-255

self.rect = Rectangle(size=layout\_instance.size,pos=layout\_instance.pos)

很不幸, 这么做只能在初始的位置以size大小绘制一个矩形. 为了确保矩形是绘制在布局内,我们需要监听任何的size/pos的改变去对矩形的位置和大小进行调整.我们可以写如下代码来解决这个问题:

with layout\_instance.canvas.before:

Color(0, 1, 0, 1) # green; colors range from 0-1 instead of 0-255

self.rect = Rectangle(size=layout\_instance.size,pos=layout\_instance.pos)

def update\_rect(instance, value):

instance.rect.pos = instance.pos instance.rect.size = instance.size

# 监听size 和pos 的变化.

layout\_instance.bind(pos=update\_rect, size=update\_rect)

在 kv 文件中:

FloatLayout:

canvas.before:

Color:

rgba: 0, 1, 0, 1

Rectangle:

# self here refers to the widget i.e BoxLayout

pos: self.pos

size: self.size

kv文件声明设置了一个隐式绑定:kv文件的最后两行确保在floatlayout的pos变化的时候矩形的pos和size能得到相应的更新或者变化.

现在我们将上面的代码段插入一段Kivy App代码中:

纯python方法:

from kivy.app import App

from kivy.graphics import Color, Rectangle

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.button import Button

class RootWidget(FloatLayout):

def \_\_init\_\_(self, \*\*kwargs):

# 确保我们不会覆盖任何重要的功能.

super(RootWidget, self).\_\_init\_\_(\*\*kwargs)

# 添加一个控件到当前的layout上

self.add\_widget(

Button(

text="Hello World",

size\_hint=(.5, .5),

pos\_hint={'center\_x': .5, 'center\_y': .5}))

class MainApp(App):

def build(self):

self.root = root = RootWidget()

root.bind(size=self.\_update\_rect, pos=self.\_update\_rect)

with root.canvas.before:

Color(0, 1, 0, 1) # green; colors range from 0-1 not 0-255

self.rect = Rectangle(size=root.size, pos=root.pos)

return root

def \_update\_rect(self, instance, value):

self.rect.pos = instance.pos

self.rect.size = instance.size

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

使用kivy语言:

from kivy.app import App

from kivy.lang import Builder

root = Builder.load\_string('''

FloatLayout:

canvas.before:

Color:

rgba: 0, 1, 0, 1

Rectangle:

# self here refers to the widget i.e FloatLayout

pos: self.pos

size: self.size

Button:

text: 'Hello World!!'

size\_hint: .5, .5

pos\_hint: {'center\_x':.5, 'center\_y': .5}

''')

class MainApp(App):

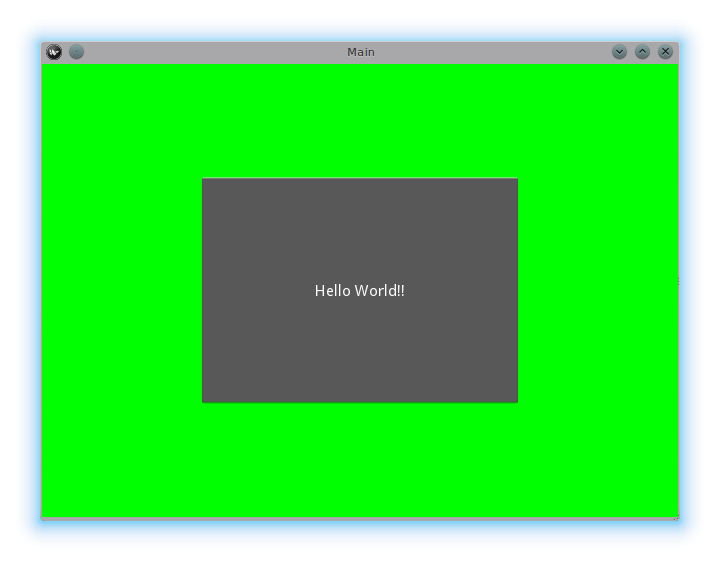
def build(self):

return root

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

上面两段代码运行结果都是这样:



为自定义的布局规则/类添加一个背景颜色:

如果我们需要多重布局那么以上我们给布局对象添加颜色的方法很快就会非常麻烦.为了解决这个问题,你可以继承Layout,创建一个你自己的layout然后添加一个背景.

使用python的方式:

from kivy.app import App

from kivy.graphics import Color, Rectangle

from kivy.uix.boxlayout import BoxLayout

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.image import AsyncImage

class RootWidget(BoxLayout):

pass

class CustomLayout(FloatLayout):

def \_\_init\_\_(self, \*\*kwargs):

# make sure we aren't overriding any important functionality

super(CustomLayout, self).\_\_init\_\_(\*\*kwargs)

with self.canvas.before:

Color(0, 1, 0, 1) # green; colors range from 0-1 instead of 0-255

self.rect = Rectangle(size=self.size, pos=self.pos)

self.bind(size=self.\_update\_rect, pos=self.\_update\_rect)

def \_update\_rect(self, instance, value):

self.rect.pos = instance.pos

self.rect.size = instance.size

class MainApp(App):

def build(self):

root = RootWidget()

c = CustomLayout()

root.add\_widget(c)

c.add\_widget(

AsyncImage(

source="http://www.everythingzoomer.com/wp-content/uploads/2013/01/Monday-joke-28",

size\_hint=(1, .5),

pos\_hint={'center\_x': .5, 'center\_y': .5}))

root.add\_widget(AsyncImage(source='http://www.stuffistumbledupon.com/wp-content/uploads/2'))

c = CustomLayout()

c.add\_widget(

AsyncImage(

source="http://www.stuffistumbledupon.com/wp-content/uploads/2012/04/Get-a-Girlfr",

size\_hint=(1, .5),

pos\_hint={'center\_x': .5, 'center\_y': .5}))

root.add\_widget(c)

return root

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

使用 kv 的方法:

from kivy.app import App

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.boxlayout import BoxLayout

from kivy.lang import Builder

Builder.load\_string('''

<CustomLayout>

canvas.before:

Color:

rgba: 0, 1, 0, 1

Rectangle:

pos: self.pos

size: self.size

<RootWidget>

CustomLayout:

AsyncImage:

source: 'http://www.everythingzoomer.com/wp-content/uploads/2013/01/Monday-joke-289x2'

size\_hint: 1, .5

pos\_hint: {'center\_x':.5, 'center\_y': .5}

AsyncImage:

source: 'http://www.stuffistumbledupon.com/wp-content/uploads/2012/05/Have-you-seen-this-'

CustomLayout

AsyncImage:

source: 'http://www.stuffistumbledupon.com/wp-content/uploads/2012/04/Get-a-Girlfrien'

size\_hint: 1, .5

pos\_hint: {'center\_x':.5, 'center\_y': .5}

''')

class RootWidget(BoxLayout):

pass

class CustomLayout(FloatLayout):

pass

class MainApp(App):

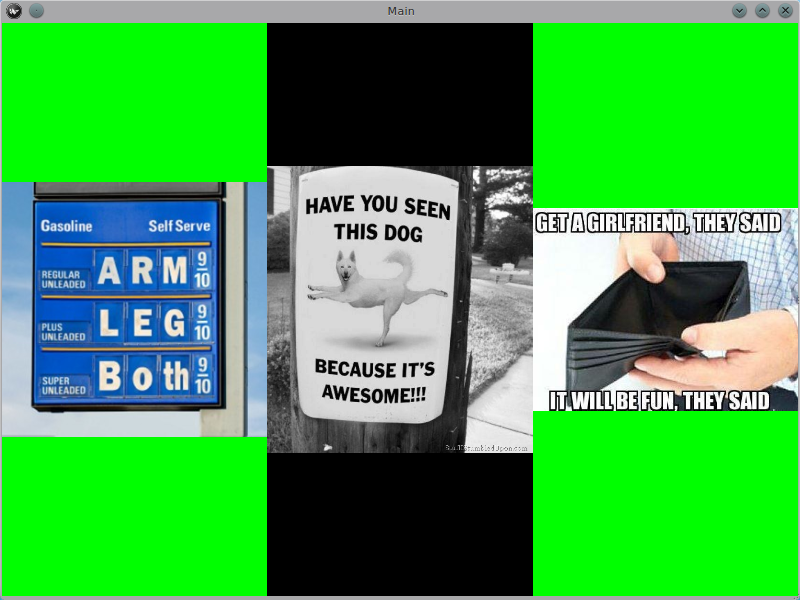
def build(self):

return RootWidget()

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

两份代码的运行结果都是如下所示(译者注:其中的图片可能加载不出来,可以换个图片):



在自定义布局类定义了背景要确保它将用于customlayout每个实例.

现在,为一个内建的ivy Layout添加一个图像或者颜色背景,我们需要为有问题(或者翻译为争议)的layout(布局)全局的重写 kv 规则.考虑一下GridLayout:

<GridLayout>

canvas.before:

Color:

rgba: 0, 1, 0, 1

BorderImage:

source: '../examples/widgets/sequenced\_images/data/images/button\_white.png'

pos: self.pos

size: self.size

将上面的代码片段添加到一个Kivy App中:

from kivy.app import App

from kivy.uix.floatlayout import FloatLayout

from kivy.lang import Builder

Builder.load\_string('''

<GridLayout>

canvas.before:

BorderImage:

# BorderImage behaves like the CSS BorderImage

border: 10, 10, 10, 10

source: '../examples/widgets/sequenced\_images/data/images/button\_white.png'

pos: self.pos

size: self.size

<RootWidget>

GridLayout:

size\_hint: .9, .9

pos\_hint: {'center\_x': .5, 'center\_y': .5}

rows:1

Label:

text: "I don't suffer from insanity, I enjoy every minute of it"

text\_size: self.width-20, self.height-20

valign: 'top'

Label:

text: "When I was born I was so surprised; I didn't speak for a year and a half."

text\_size: self.width-20, self.height-20

valign: 'middle'

halign: 'center'

Label:

text: "A consultant is someone who takes a subject you understand and makes it sound

text\_size: self.width-20, self.height-20

valign: 'bottom'

halign: 'justify'

''')

class RootWidget(FloatLayout):

pass

class MainApp(App):

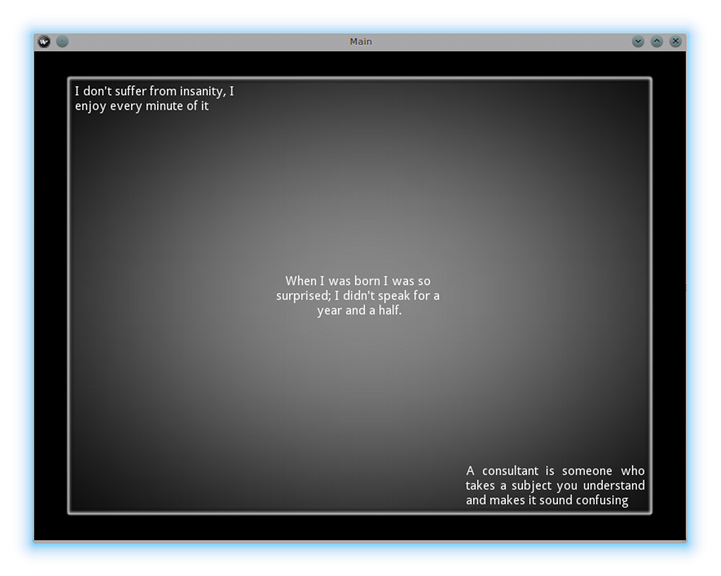
def build(self):

return RootWidget()

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

上面的运行结果如下:



当我们重写了FridLayout的布局规则之后,任何一个使用了这个类创建的app都将显示image.

那么动态背景咋整?

你可以设置绘图指令如矩形/椭圆/ borderimage /…使用特殊纹理:

Rectangle:

texture: reference to a texture

我们用这个来显示一个动态背景:

from kivy.app import App

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.gridlayout import GridLayout

from kivy.uix.image import Image

from kivy.properties import ObjectProperty

from kivy.lang import Builder

Builder.load\_string('''

<CustomLayout>

canvas.before:

BorderImage:

# BorderImage behaves like the CSS BorderImage

border: 10, 10, 10, 10

texture: self.background\_image.texture

pos: self.pos

size: self.size

<RootWidget>

CustomLayout:

size\_hint: .9, .9

pos\_hint: {'center\_x': .5, 'center\_y': .5}

rows:1

Label:

text: "I don't suffer from insanity, I enjoy every minute of it"

text\_size: self.width-20, self.height-20

valign: 'top'

Label:

text: "When I was born I was so surprised; I didn't speak for a year and a half."

text\_size: self.width-20, self.height-20

valign: 'middle'

halign: 'center'

Label:

text: "A consultant is someone who takes a subject you understand and makes it sound"

text\_size: self.width-20, self.height-20

valign: 'bottom'

halign: 'justify'

''')

class CustomLayout(GridLayout):

background\_image = ObjectProperty(

Image(source='../examples/widgets/sequenced\_images/data/images/button\_white\_animated.zip', anim\_delay=.1))

class RootWidget(FloatLayout):

pass

class MainApp(App):

def build(self):

return RootWidget()

if \_\_name\_\_ == '\_\_main\_\_':

MainApp().run()

想要了解发生了什么??从第13行开始:

texture: self.background\_image.texture

这行代码指定了当texture的background\_image属性更新的时候texture的Borderimage属性也会更新.在低40行定义了background\_image属性:

background\_image = ObjectProperty(…

当我们添加一个图片控件(Image widget)时,这个设定好的background\_image可以当做一个Objectproperty(译者注:当做固有属性)

一个图片控件拥有一个texture属性,无论在哪里使用了self.background\_image.texture都会设置一个引用和texture属性.这个图片控件支持动画效果:无论何时动画改变了,动画的texture都会随之改变同时在此过程中BorderImgae指令的texture也会随之改变.

你也可以显示自定义数据结构,详情请看Texture的文档.

12.7 布局嵌套(Nesting Layouts)

Yes! 这个扩展的过程一定会很有趣.

12.8 位置和大小的度量

Kivy的默认长度单位是像素,所有的大小和位置都是以像素位置和大小来表现的.当然如果可以让你的程序的承载设备有更好的一致性,你也可以用其他的丈量单位来展现它们(这些丈量的方法会自动的转换为像素).

可用的丈量单位有pt, mm, cm, inch, dp 和 sp. 你可以在metrics的文档中学习相关内容.

你也可以用多种不同设备来模拟你的应用程序的在其屏幕闪的表现.

12.9 屏幕从屏幕管理器中分离.

如果你的应用程序要适配多个屏幕,你可能想要有非常简单的方法能实现从一个到另一个的适配.幸运的是,有一个ScreenManager的类,这个类能让你分开定义屏幕,并且设置一个屏幕到另一个屏幕的Transitionbase.