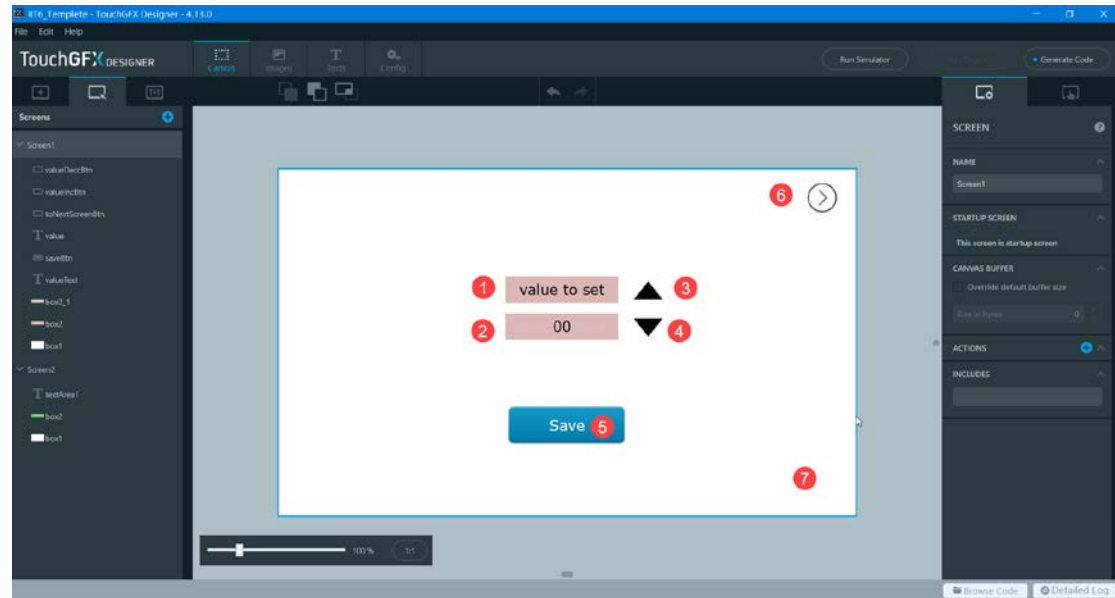


## 多屏数据共享

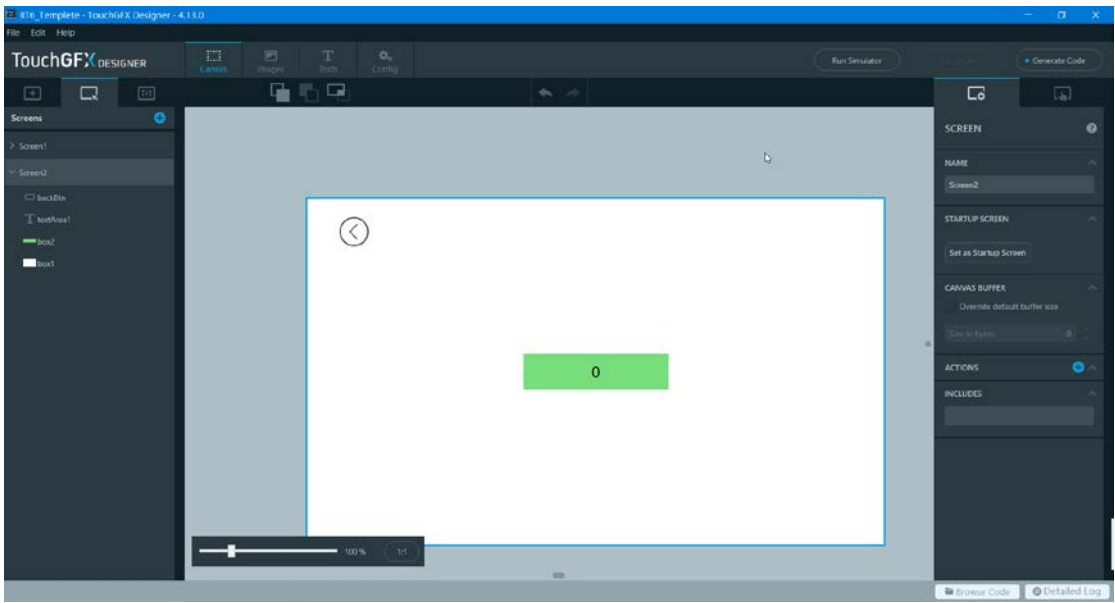
在实际应用中，很难只用一页屏幕内容就能把所有的功能展示出来，常常需要制作多个屏幕，屏幕之间的通信需要用到 MVP 的通信机制。在本教程中，您将学习如何在应用程序中创建多个屏幕以及如何两个屏幕之间共享数据。创建的程序共有两个屏幕，第一个屏幕用来设置数值，第二个屏幕显示数值。

1、创建空白项目，搭建第一屏如下：



序号	控件类型	实例名称	说明
1	textArea	valueText	文本标签
2	textArea	value	启用通配符，3 字节
3	Button	valueIncBtn	增加
4	Button	valueDecBtn	减少
5	Button	saveBtn	保存
6	Button	toNextScreen	切换到下一屏
7	Box	box1	纯色背景
粉色框	Box	box2, box2_1	纯色背景

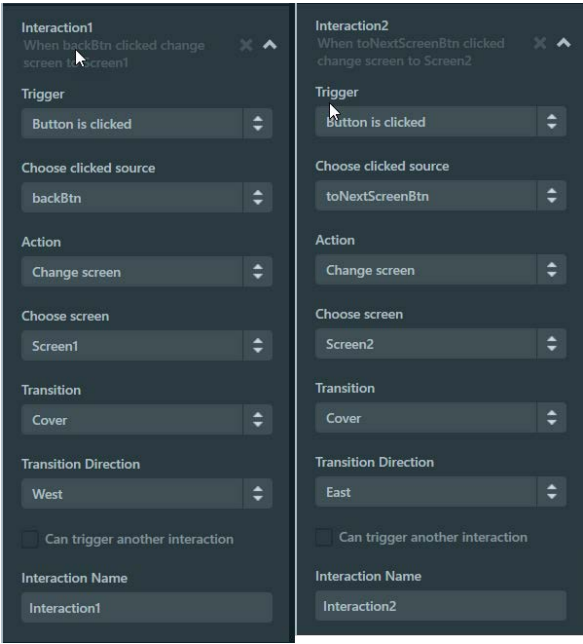
2、第二屏如图：



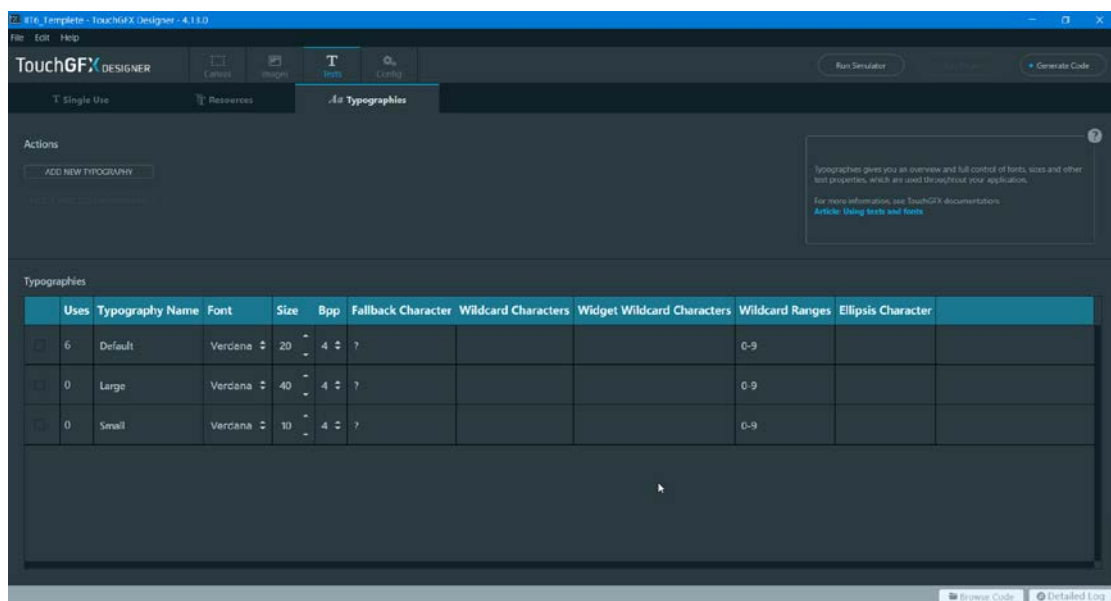
3、给按键增加交互：

按键名称	Action	FunctionName
valueIncBtn	Call new virtual function	increaseValue
valueDecBtn	Call new virtual function	decreaseValue
saveBtn	Call new virtual function	saveValue

给两个按键添加切换屏幕交互：

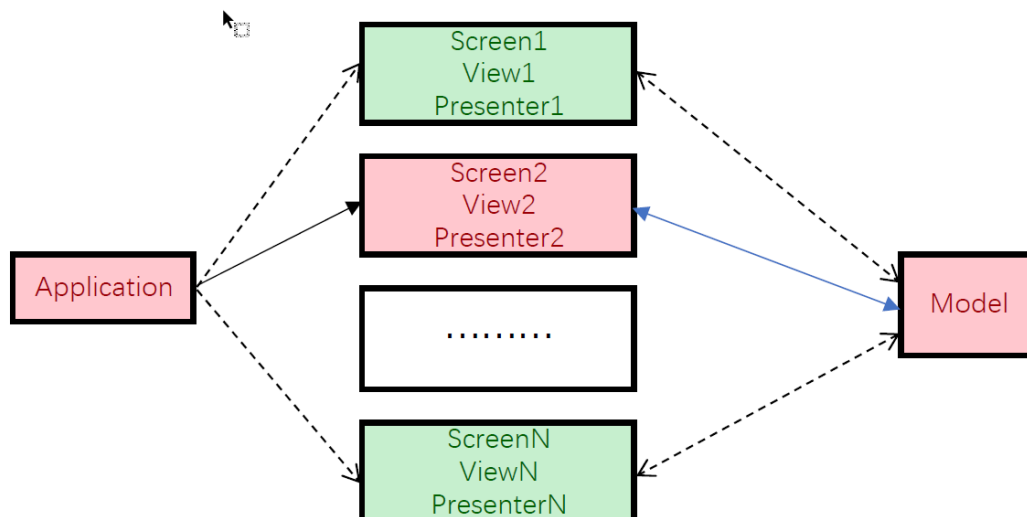


4、将数字添加进通配符：



5、touchgfx 里配置的部分到此结束，接下来需要在 vs 中实现这几个交互函数。在这之前，我发需要补充一点 touchgfx MVP 架构在切换屏幕时的工作原理。

每一个屏幕都有一对 View 和 Presenter，当需要切换屏幕时，后台运行的 Application 对象负责创建新的 View 和 Presenter 对象，被切出屏幕的 View 对象和 Presenter 对象会在某个时刻被回收，切入的屏幕会重新初始化。所以，保存在 View 对象和 Presenter 对象中的数据可能会丢失，唯一可以安全存放用户数据的地方是 Model 对象。虽然可以采用声明全局变量的方式解决数据传递的问题，但是却破坏了 MVP 的结构，不推荐这么做。当需要共享数据的屏幕都实现了访问 Model 成员变量的方法时，他们就能通过 Model 共享数据了。如下示意图是多屏应用的结构图：



**Application:** 负责切换屏幕

**Model:** 负责存储数据、与硬件系统接口所以，两个屏幕如果想共享数据，这些数据便可以放在第三方——Model。

理论上重要的成员变量都应该设置成 `private`，并增加 `public` 的方法来进行读写操作，以保证数据安全，这里为了突出屏幕之间数据共享的过程，直接把该成员声明成了 `public` 属性，通过其实例就能进行访问：

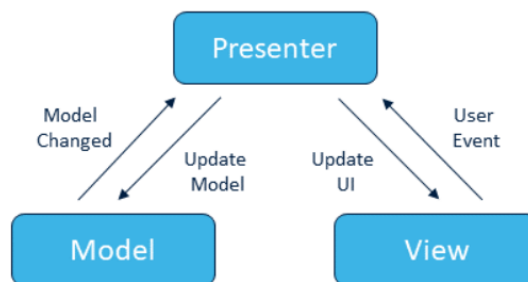
```

Model.h
class Model
{
public:
    Model();
    void bind(ModelListener* listener)
    {
        modelListener = listener;
    }
    int value;
    void tick();
protected:
    ModelListener* modelListener;
};

Model.cpp
#include <gui/model/Model.h>
#include <gui/model/ModelListener.h>
Model::Model() : modelListener(0), value(0)
{
}
void Model::tick()
{
}

```

6、Model、View、Presenter 之间数据流向关系如图：



Model-View-Presenter Design Pattern

可以看出 Model 和 View 之间只能通过 Presenter 实现数据传递，所以我们在 Presenter 类中添加访问 Model 成员变量的方法(简单的函数体可以直接写在头文件里)。

对屏幕 1：(1)在 Presenter 里添加访问 value 的方法：

```

Screen1Presenter.cpp
void Screen1Presenter::setValue(int a)
{
    model->value = a;
}
int Screen1Presenter::getValue(void)
{
    return model->value;
}

```

(2)在 View 里实现 increaseValue、 decreaseValue、 saveValue 三个回调函数：

```

14
15 virtual void increaseValue()
16 {
17     valueToSet++;
18     Unicode::snprintf(valueBuffer, 3, "%02d", valueToSet);
19     box2.invalidate();
20 }
21
22 virtual void decreaseValue()
23 {
24     valueToSet--;
25     Unicode::snprintf(valueBuffer, 3, "%02d", valueToSet);
26     box2.invalidate();
27 }
28 void savevalue()
29 {
30     presenter->setValue(valueToSet);
31 }
32 protected:
33 int valueToSet;
34 };
  
```

valueToSet 成员用来暂时存储设定值，当点击 save 按键的时候才把值写到 Model 类的 value 变量里。

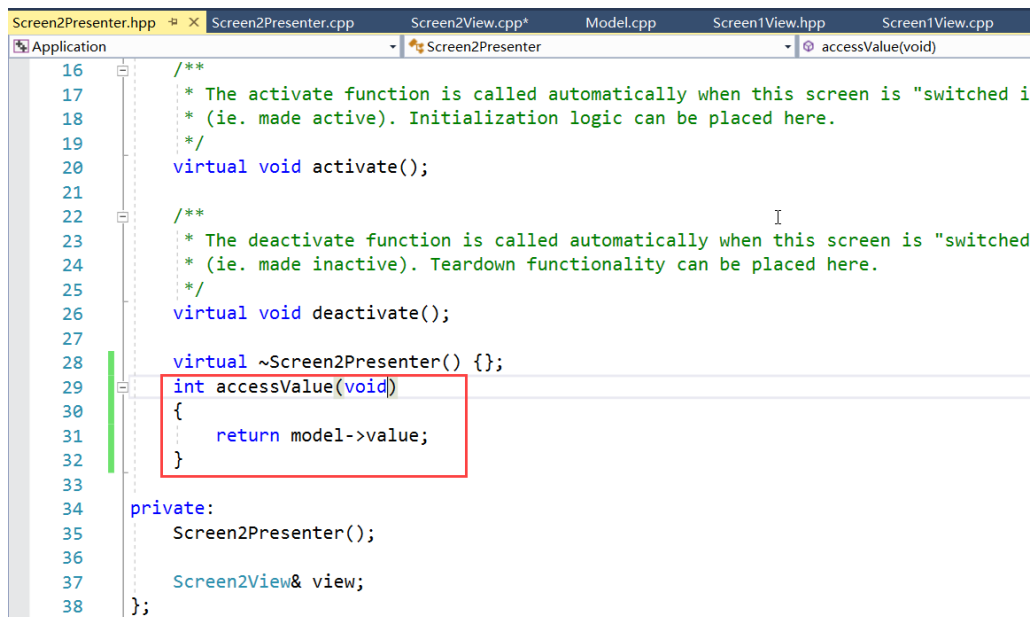
(3)在屏幕 1 切入的时候，应该让界面上显示 Model 里设置的值，所以在 View 的 setupScreen 函数里增加更新显示的代码：

```

Model.cpp*  Screen1View.hpp  Screen1View.cpp*
Application  Screen1View
1  #include <gui/screen1_screen/Screen1View.hpp>
2
3  Screen1View::Screen1View()
4  {
5      valueToSet = 0;
6  }
7
8  void Screen1View::setupScreen()
9  {
10     Screen1ViewBase::setupScreen();
11     valueToSet = presenter->getValue();
12     Unicode::snprintf(valueBuffer, 3, "%02d", valueToSet);
13 }
14
15 void Screen1View::tearDownScreen()
16 {
17     Screen1ViewBase::tearDownScreen();
18 }
  
```

对屏幕 2 也需要做同样的操作：

(1)在 Presenter 里添加访问 value 的方法：



```

Screen2Presenter.cpp
Application
Screen2Presenter
accessValue(void)
16  /**
17   * The activate function is called automatically when this screen is "switched i
18   * (ie. made active). Initialization logic can be placed here.
19   */
20  virtual void activate();
21
22  /**
23   * The deactivate function is called automatically when this screen is "switched
24   * (ie. made inactive). Teardown functionality can be placed here.
25   */
26  virtual void deactivate();
27
28  virtual ~Screen2Presenter() {};
29  int accessValue(void)
30  {
31      return model->value;
32  }
33
34  private:
35      Screen2Presenter();
36
37      Screen2View& view;
38  };
  
```

(2)在 setupScreen 方法中调用 accessValue 并更新显示，这样每次切入屏幕 2 都会获取最新的 value 值，并且显示出来：



```

Screen2View.cpp
Application
Screen2View
setupScreen()
1  #include <gui/screen2_screen/Screen2View.hpp>
2
3  Screen2View::Screen2View()
4  {
5
6  }
7
8  void Screen2View::setupScreen()
9  {
10     Screen2ViewBase::setupScreen();
11     //获取value值，并在TextArea控件中显示
12     Unicode::snprintf(textArea1Buffer, 3, "%02d", presenter->accessValue());
13     //适应文本长宽，如果不调整一下长宽，可能文本内容显示不全
14     textArea1.resizeToCurrentText();
15     //刷新粉色背景，如果不刷新，显示的数值可能不会改变
16     box2.invalidate();
17 }
18
19 void Screen2View::tearDownScreen()
20 {
21     Screen2ViewBase::tearDownScreen();
22 }
  
```

至此，屏幕 1 上的参数就已经能被屏幕 2 访问了。

知识补充：

拿屏幕 1 为例，ScreenView1 类除了有与之同名的构造函数外，setupScreen 和 tearDownScreen 两个函数，在屏幕切出时，tearDownScreen 函数会被执行，通常可以在这里面执行保存数据的操作；在屏幕切入时，构造函数和 setupScreen 两者都会被调用，但是构造函数通常只放一些与实例化相关的操作，setupScreen 函数负责执行与屏幕画面相关的工作。

## 在屏幕中使用动作和触发器

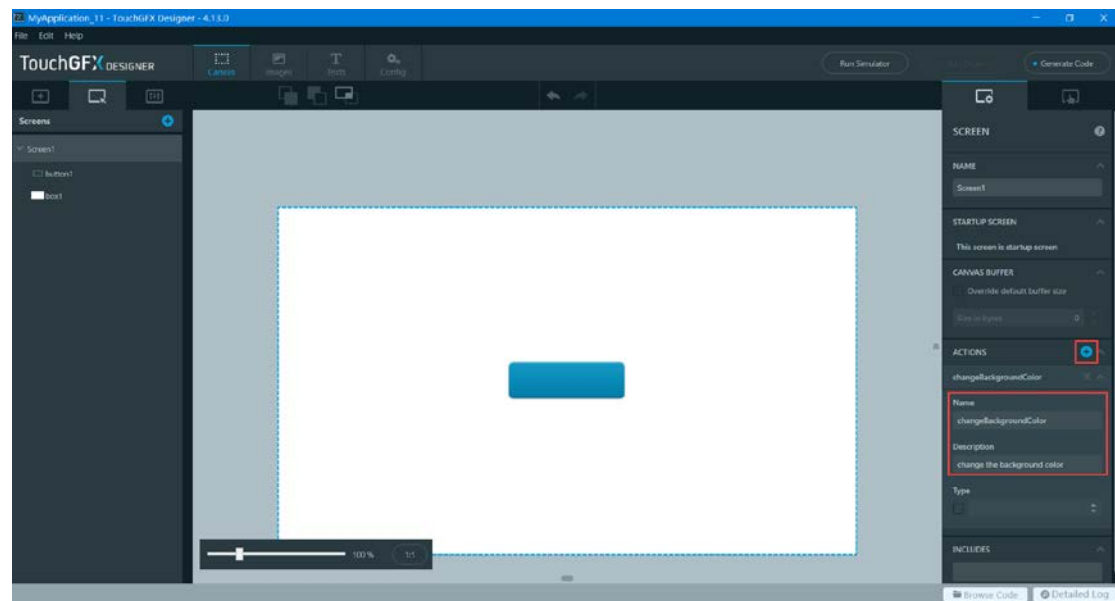
动作(Actions)和触发器(Triggers)是 Touchgfx Designer 里非常重要的两个概念，灵活运用这两个功能可以大幅提高界面功能的灵活性、降低编程难度，对不怎么熟悉 C++ 模板用户尤其友好。看过基础教程中自定义容器一节的同学就知道，给自定义注册一个回调函数是多么麻烦。使用这两个功能就能完全避免手动添加那些复杂度很高的代码。

**动作：**void 函数，可以在 TouchGFX Designer 的属性栏里调用，也可以在代码中调用。

**触发器：**相当于事件的回调函数，事件发生时自动执行，可以传递参数。

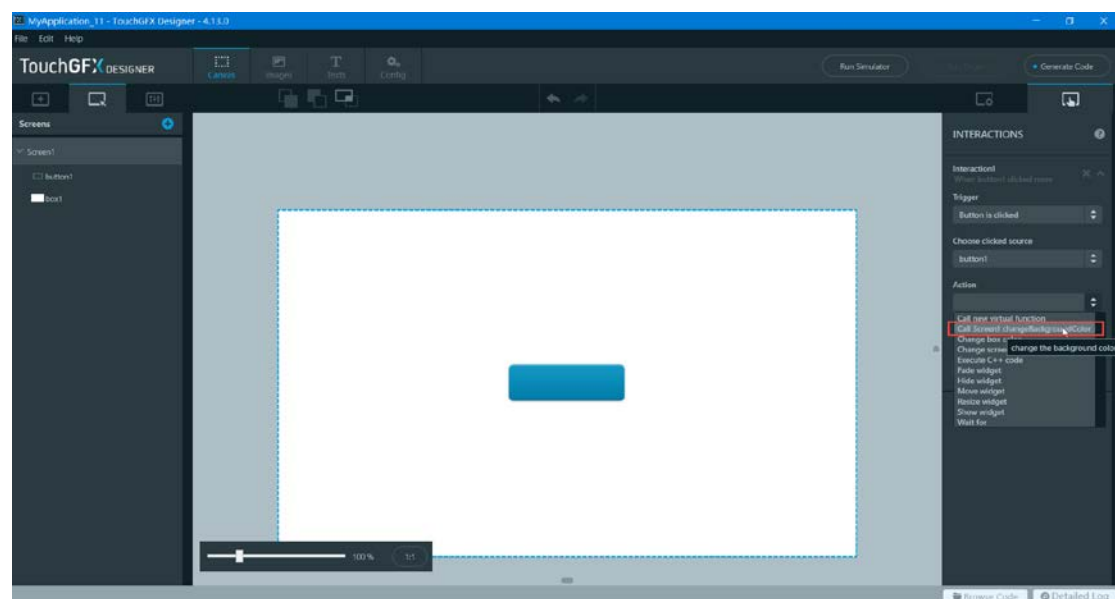
屏幕可以添加自定义动作，自定义容器可以添加自定义动作和触发器。下面通过实验演示如何用按键调用自定义动作来改变屏幕背景。

1、创建新工程，背景设置为白色、添加一个按键，并给屏幕添加自定义动作(点击画布以外的地方即是选中当前屏幕)：



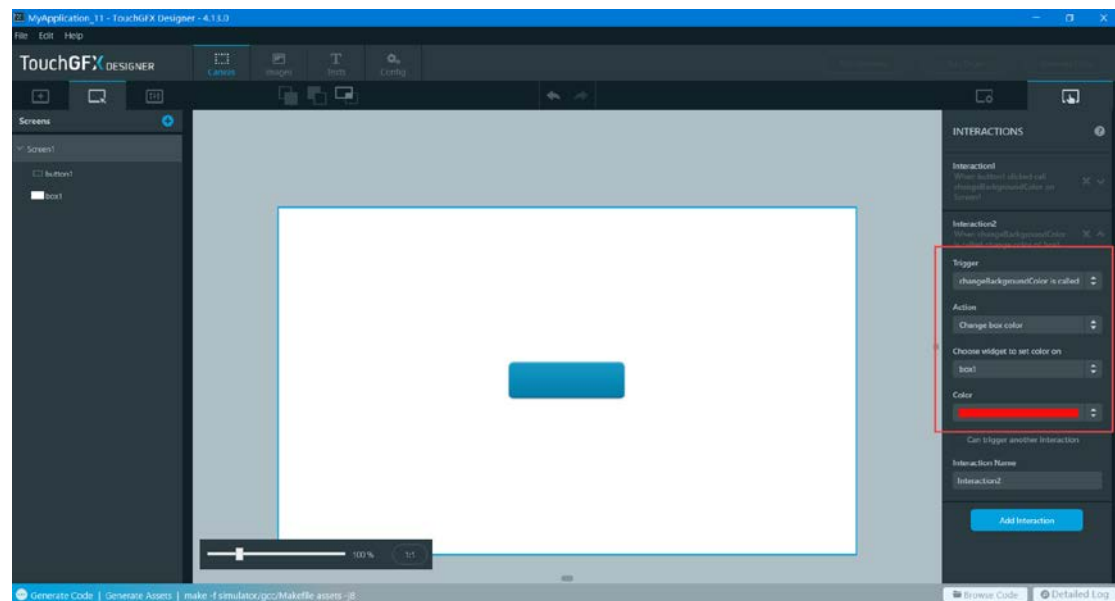
这一步操作将会在 ScreenView1Base.hpp 里生成名为 changeBackgroundColor 的空函数。

2、给按键添加交互，在 Action 下来选项里选择创建好的动作：



因为 changeBackgroundColor 函数是空的，具体要实现什么功能需要我们在代码里实

现，但 TouchGFX Designer 提供了一种更简便的方法：添加另一个交互，交互的触发器设置为 `changeBackgroundColor` is called，动作设置为修改 `box` 颜色，如图：

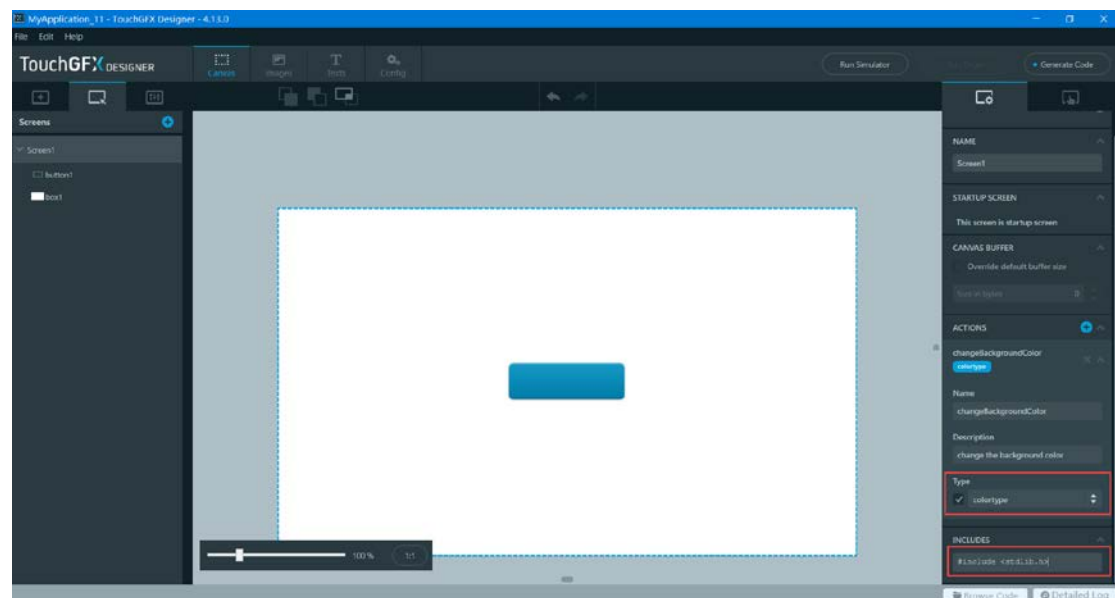


Touchgfx 后台工作流程如下：



至此，当按键按下时，屏幕的背景色就会变成红色。

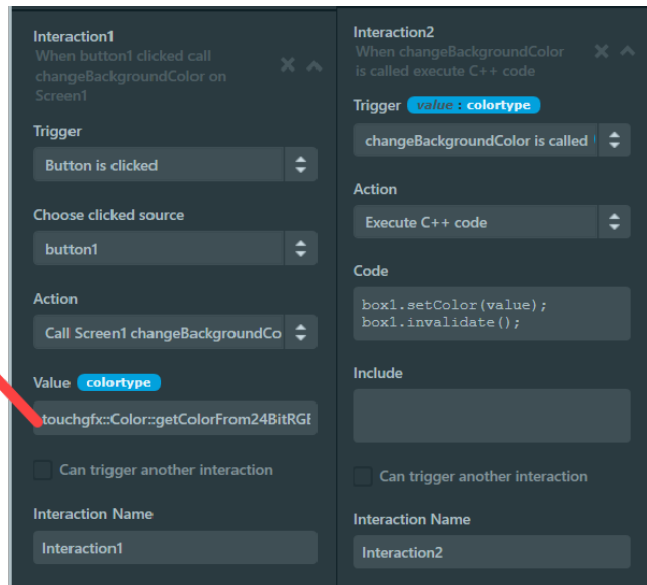
3、假如需要将背景修改为任意颜色，那么自定义动作就需要传递颜色参数，让最终执行的代码能根据这个参数来改变屏幕背景颜色。实现这个功能也很简单，直接在 **ACTIONS** 栏勾选 **Type**，然后填上参数类型 `colortype`，形参的名字默认为 `value`，用户修改不了 (`colortype` 是 Touchgfx 的内置颜色类型)。后面会用到 `stdlib.h` 中随机函数 `rand()`，所以在 **INCLUDES** 栏里填上 `#include <stdlib.h>`：





## 4、把刚才创建的两个交互改成如下：

touchgfx::Color::getColorFrom24BitRGB(  
rand()%256, rand()%256, rand()%256)



这样，每次点击按键时，形参 value 就被传入一个随机的颜色值，Interaction2 根据参数值修改背景的颜色。现在编译、仿真运行已经能看到结果了。

以下是 TouchGFX 生成的代码，可以看出：回调函数中的具体实现就是我们在 Touchgfx Designer 中填写的代码：

```
Screen1ViewBase.cpp # x
Application -> Screen1ViewBase -> setupScreen()
22
23 void Screen1ViewBase::setupScreen()
24 {
25 }
26
27
28 void Screen1ViewBase::changeBackgroundColor(colortype value)
29 {
30     //Interaction2
31     //When changeBackgroundColor is called execute C++ code
32     //Execute C++ code
33     box1.setColor(value);
34     box1.invalidate();
35 }
36
37 void Screen1ViewBase::buttonCallbackHandler(const touchgfx::AbstractButton& src)
38 {
39     if (&src == &button1)
40     {
41         //Interaction1
42         //When button1 clicked call changeBackgroundColor on Screen1
43         //Call changeBackgroundColor
44         changeBackgroundColor(touchgfx::Color::getColorFrom24BitRGB(rand()%256, rand()%256, rand()%256));
45     }
46 }
```

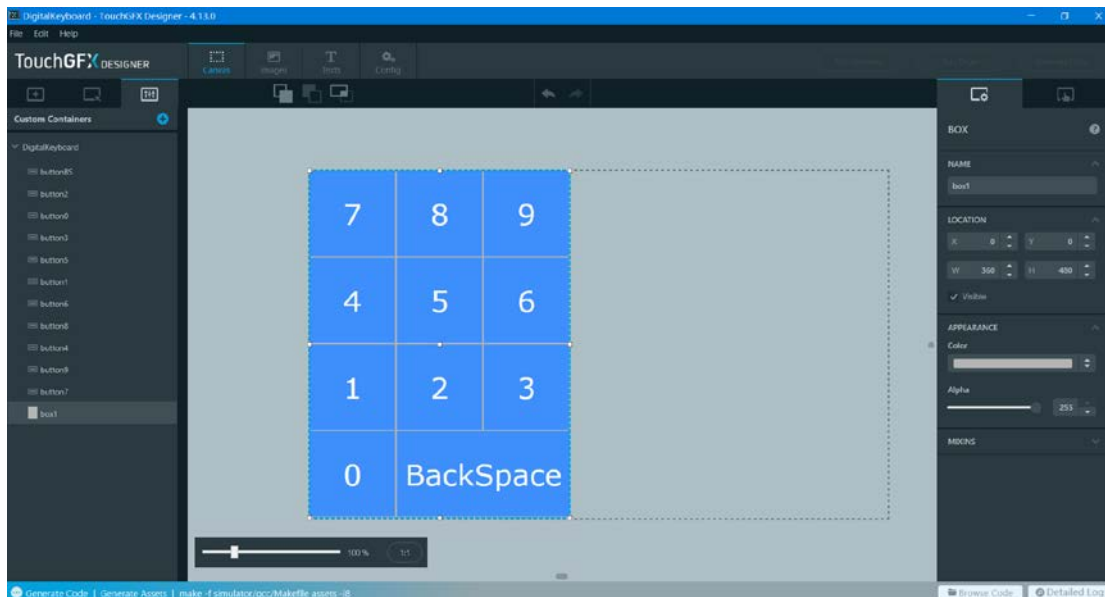
总结：屏幕中定义的动作在 Interaction 中被当做触发器或回调函数使用，Touchgfx Designer 会自动注册回调函数，我们只需在回调函数中补全代码就行了。

## 自定义容器使用动作和触发器

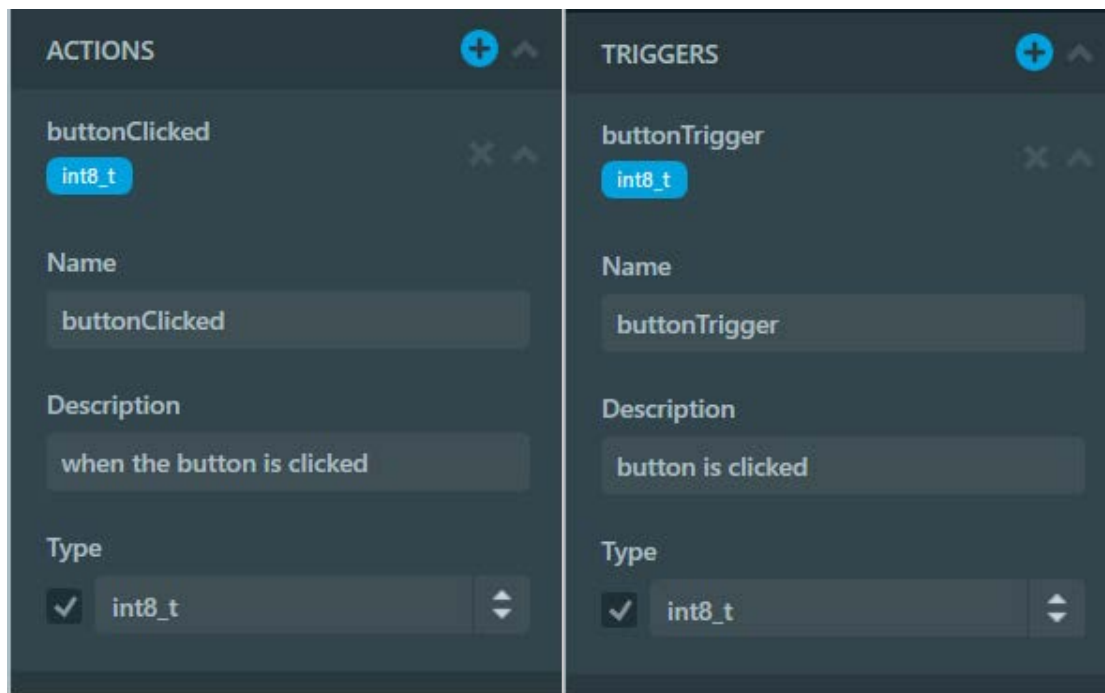
自定义容器可以在属性栏添加动作和触发器，用户无需手动注册回调函数就能将自定义容器的事件和数据传递出来，让屏幕捕捉到并进行处理，降低了编程的难度。

下面以一个数字键盘为例，说明如何使用动作和触发器。Touchgfx 自带 Keyboard 类，但是这里没有使用，这部分内容将在后面的教程中介绍。

1、新建空白项目，搭建一个自定义容器，一共用了 11 个 ButtonWithLabel 控件，如图：



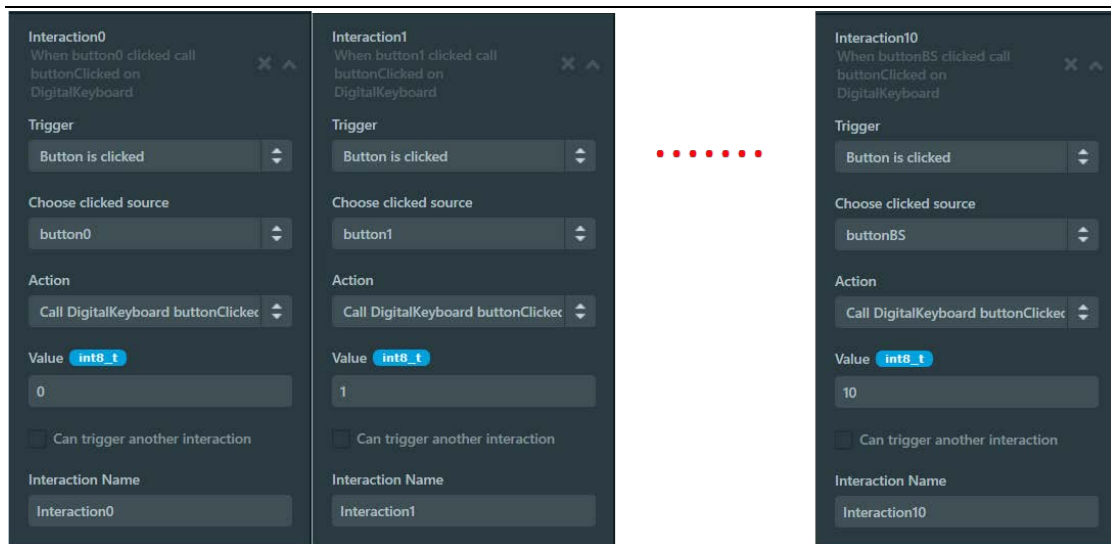
2、给自定义容器添加一个动作和一个触发器，如图：



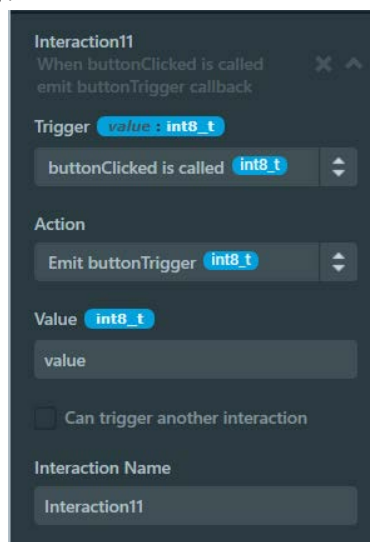
3、给每一个按键都添加一个交互，在被按下的时候调用 buttonClicked 动作，并把相应的键码传递给形参，这里我们定义的键码如下表所示：

按键	0	1	2	3	4	5	6	7	8	9	BackSpace
键码	0	1	2	3	4	5	6	7	8	9	10

交互截图如下：

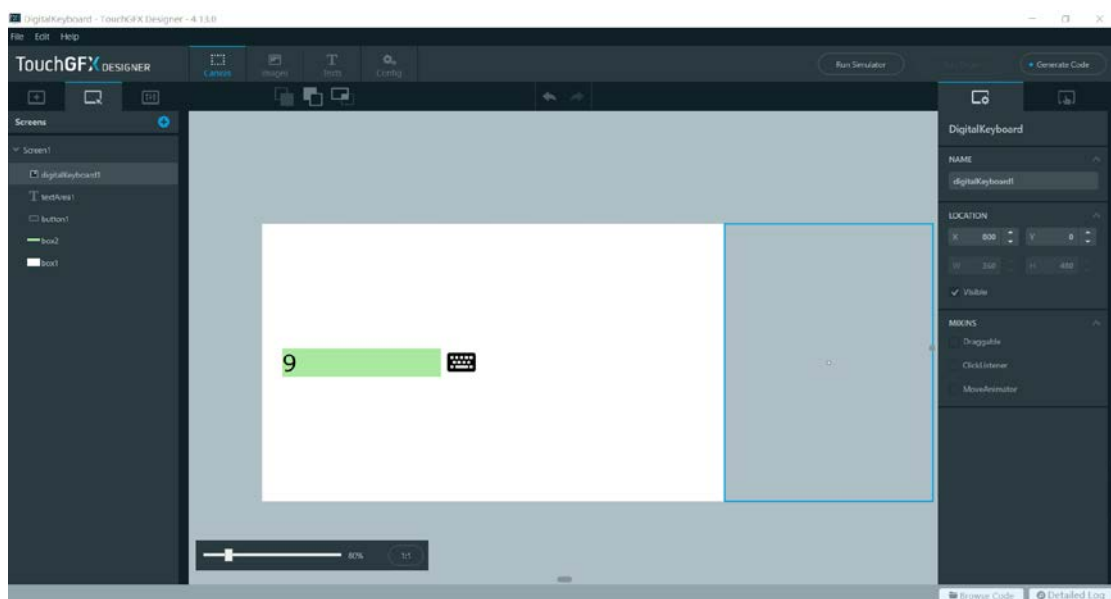


4、再添加一个交互，当 buttonClicked 动作被调用时，自定义容器向外发送 buttonTrigger 事件，键码被当做参数传了出去：

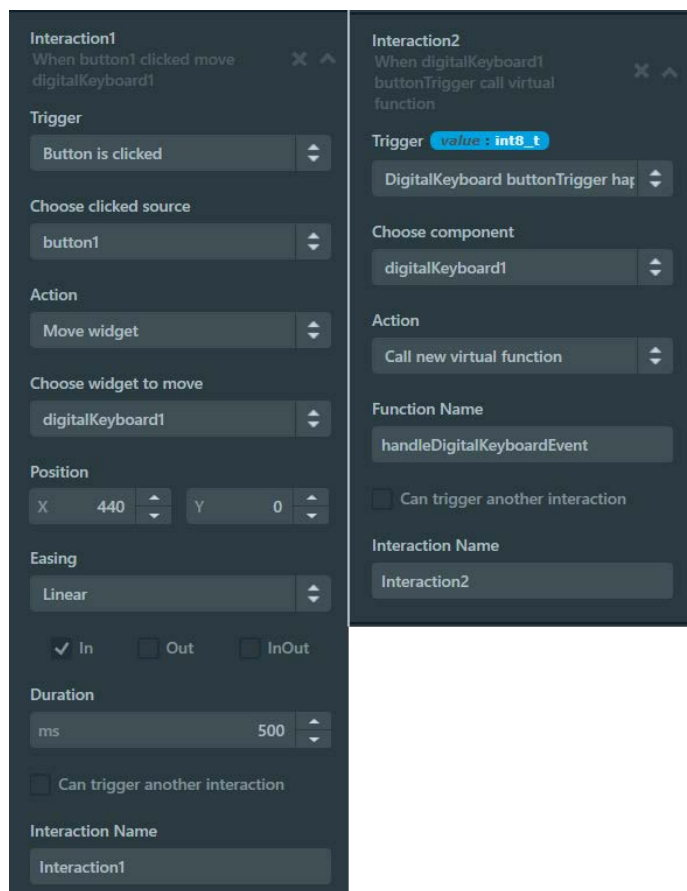


这里 value 是 buttonClicked 的形参，Touchgfx 默认的命名，用户不能修改。

5、在画布中构建界面如图所示，给 textArea1 启用 12 字节的通配符，把刚才做好的键盘放在屏幕右侧，当点击黑色小键盘按钮的时候，键盘从右侧滑入屏幕：



在屏幕中添加两个交互，一个让键盘滑入屏幕，另一个用来监测是否有按键按下，如果有，就用 `handleKeyboardEvent` 函数处理：



至此，自定义数字键盘就已经能够向屏幕发送点击事件和键码，屏幕也能捕捉到这个事件并进行处理。

6、接下来在 VS 中实现 `handleKeyboardEvent` 回调函数：

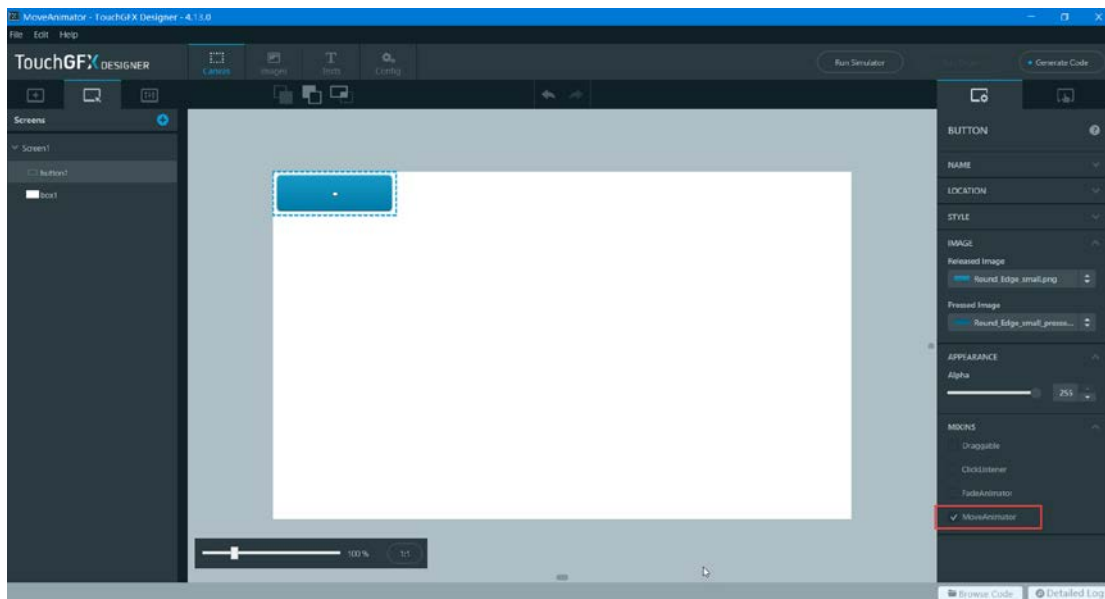
```
Screen1View.hpp | Screen1View.cpp | Application | Screen1View
16 | }
17 |
18 | void Screen1View::handleDigitalKeyboardEvent(int8_t value)
19 | {
20 |     static int index = 0; //指向结束符的索引
21 |     if (value != 10)
22 |     {
23 |         if (index < TEXTAREA1_SIZE - 1)
24 |         {
25 |             textArea1Buffer[index++] = value + 48;
26 |             textArea1Buffer[index] = 0;
27 |         }
28 |     }
29 |     else
30 |     {
31 |         if(index>0)
32 |             textArea1Buffer[--index] = 0;
33 |     }
34 |
35 |     textArea1.resizeToCurrentText();
36 |     textArea1.invalidate();
37 |     box2.invalidate();
38 |
39 | }
40 |
```

现在，文本区域最多可以显示按键输入的 11 个字符了。尽管通配符有 12 字节，但最后一个字节必须用来存放结束符。

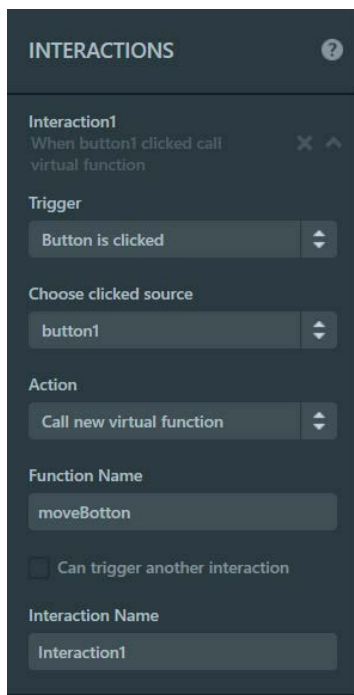
## 移动动画

在屏幕中移动控件是很常用的动画效果，在第三节已经用到过移动动画了一键盘从右往左滑动静茹屏幕，但是例程却没有实现键盘从左往右移出屏幕的动画。Touchgfx 定义了一个 MoveAnimator 类模板，任何需要使用移动动画的控件都需要先声明为 MoveAnimator 类。这一节将演示如何将控件移动到任意坐标点。

1、构建如图屏幕，在 button 的属性栏里勾选 MoveAnimator，这样，Touchgfx Designer 在生成代码的时候会自动把 button 控件声明为 MoveAnimator 对象，然后就可以使用 MoveAnimator 类提供的控制动画的方法了：



2、给按键添加一个点击回调函数，用来触发动画：



3、生成代码，可以在 Screen1ViewBase.hpp 里看到这样的代码：

```

16 {
17 public:
18     Screen1ViewBase();
19     virtual ~Screen1ViewBase() {}
20     virtual void setupScreen();
21
22     /*
23      * Virtual Action Handlers
24      */
25     virtual void moveBotton()
26     {
27         // Override and implement this function in Screen1
28     }
29
30 protected:
31     FrontendApplication& application() {
32         return *static_cast<FrontendApplication*>(touchgfx::Application::getInstance());
33     }
34
35     /*
36     * Member Declarations
37     */
38     touchgfx::Box box1;
39     touchgfx::MoveAnimator< touchgfx::Button > button1;

```

button 控件被声明为了 MoveAnimator 对象，但是 box1 因为没有勾选 MoveAnimator 选项，还是被声明为普通的 Box 对象。

#### 4、在 view 中实现 moveButton 回调函数：

```

13 void Screen1View::tearDownScreen()
14 {
15     Screen1ViewBase::tearDownScreen();
16 }
17
18 void Screen1View::moveButton()
19 {
20     static int counter = 0;
21     if ((counter++) % 2 == 0)
22     {
23         button1.clearMoveAnimationEndedAction();
24         button1.startMoveAnimation(630, 420, 18,
25             touchgfx::EasingEquations::linearEaseIn,
26             touchgfx::EasingEquations::linearEaseIn);
27     }
28     else
29     {
30         button1.clearMoveAnimationEndedAction();
31         button1.startMoveAnimation(0, 0, 18,
32             touchgfx::EasingEquations::linearEaseIn,
33             touchgfx::EasingEquations::linearEaseIn);
34     }
35 }
36

```

这样，点一下按键，按键就会从左上角移动到右下角，再点一次就会再次回到左上角。

这里用到了 MoveAnimator 两个重要的方法，其作用解释如下：

**clearMoveAnimationEndedAction():** MoveAnimator 类在动画结束时执行回调函数，这个函数由 setMoveAnimationEndedAction() 注册。我们希望在点击按键的时候直接执行我们期望的动画，取消执行回调函数，以免出现一些逻辑上的漏洞。如果没有实现回调函数，也可以不使用 clearMoveAnimationEndedAction()。

startMoveAnimation(): 顾名思义，让控件动起来。其参数含义如下：

<div><div><div>startMoveAnimation(int16_t endX, int16_t endY, uint16_t duration, EasingEquation void xProgressionEquation = &amp;EasingEquations::linearEaseNone, EasingEquation yProgressionEquation = &amp;EasingEquations::linearEaseNone)</div><div>Starts the move animation from the current position to the specified end position.</div></div></div>	
endX	目标点的 x 坐标
endY	目标点的 y 坐标
duration	动画持续时间，值越小，动画越快。单位是 Touchgfx 的 tic，约为 16.67ms。
xProgressionEquation	x 方向的动画效果
yProgressionEquation	y 方向的动画效果

这些动画效果都建议大家自己都试一试。