Generating SAW selection sets using code

Introduction

SAW is written in c# using the .net Framework. It is possible to use functions from the SAW.exe using any language which can interoperate with .net. This does not use the SAW application while running – rather it loads the SAW.exe into code you create as if it was a DLL.

These instructions use the c# language, but others are possible.

Getting started

You will require a development environment capable of writing .net code. Microsoft offer a free version of their development tools, currently available from

https://visualstudio.microsoft.com/vs/community/

or search for “Visual Studio Community”. These instructions are all based on Visual Studio. Don’t confuse it with “Visual Studio Code” which is a very different system (more focused on web development)

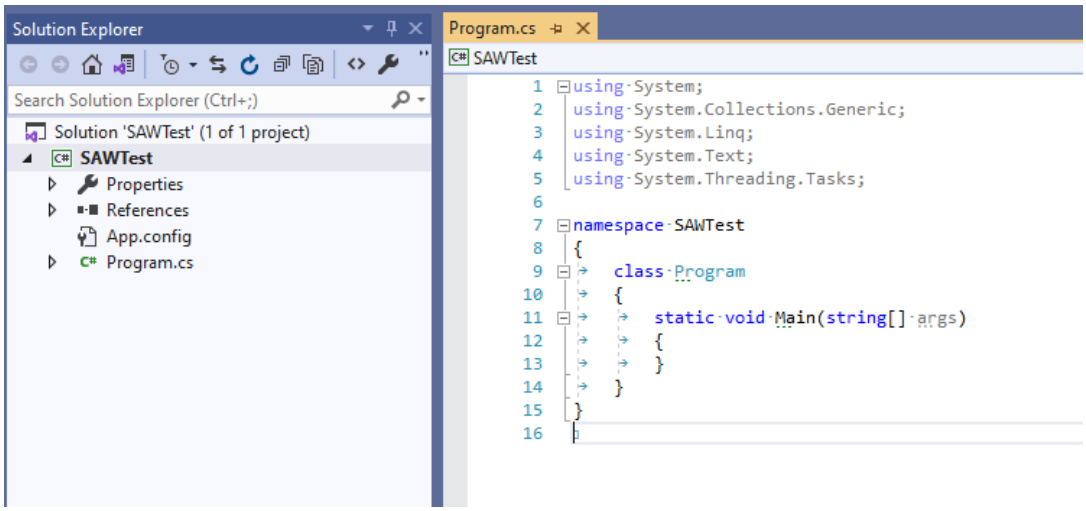
If installing this, select the “.Net desktop development” workload in the installer. The “Universal Windows platform development” option is not needed – despite the name this is not used for Windows software, rather only targets a specific virtual OS (UWP).

Creating a project

When you start Visual Studio it offers a choice between recent projects, on the left, and “Get started” on the right – select “Create a new project” on the right.

The next screen offers a huge selection of templates. Select “C#”, “Windows” and “Console” in the drop down options at the top. This should then offer a “Console App (.Net Framework)”. A console application has just text output and is the easiest option to run a single task. If you want an interactive UI, you can use one of the “Windows Forms App” options. Whichever is chosen be sure to select one ending “(.Net Framework)” not “(.Net)” or “(.Net core)” (which are all different things).

Once you click next you must give the project a name, and specify where to save it. Once created you have a project with one main code file “program.cs” which contains a main function that will run in the console window:



Referencing SAW

Before writing any code you must first tell VS that you will be using commands defined in the SAW executable. To do this right-click on the “References” header in the Solution Explorer, and select “Add Reference”. In the reference manager select “Browse” at the bottom of the screen and navigate to and select the SAW exe file in Program files.

Note that this doesn’t use the exe in-situ. Rather it will copy the exe (and some DLLs that it uses) to your project. However VS will detect if the original changes and update its copy to match.

In addition to SAW itself you must add a reference to one system assembly, System.Drawing (this may already be included if you used a different project template). To add this, again right-click on references, select “Add reference”, then in the “Assemblies” tab, tick “System.Drawing”. This contains definitions for graphics in .net – which includes some basic concepts such as Point or Rectangle.

It is also worth adding the line:

```
using System.Drawing;
```

at the top of the file. This allows, for example, use of the name “Point” without the full prefix (“System.Drawing.Point”).

It is possible to add a using for SAW itself, although since the namespace it quite short, it is probably preferable not to: when typing “SAW.” VS will then offer to auto-complete with all the definitions in SAW. It can be more useful, however, to adding usings for some of the subsidiary namespaces within SAW such as “using SAW.Commands”

Getting started

A minimal program to generate a SAW file looks something like this:

```
namespace SAWTest
{
    class Program
    {
        static void Main(string[] args)
        {
            SAW.Generator.InitialiseExternal( exeFolder: @"c:\program files (x86)\SAW");
            SAW.Document doc = SAW.Generator.CreateNewDocument(new Size(500, 500));
            SAW.Page page = doc.Page(0);
            page.AddNew(new SAW.Shapes.Line(new PointF(100, -100), new PointF(200, -1200)));

            doc.Save( file: @"d:\temp\generated.saw7");
            Console.WriteLine("Done");
            Console.ReadKey();
        }
    }
}
```

This makes a set with just a single (inactive) line on the page.

There is one initial, compulsory step, before any other SAW commands are used, which is the command:

```
SAW.Generator.InitialiseExternal(@"c:\program files (x86)\SAW");
```

The *Generator* class contains a couple of functions to support access to SAW by external code. The function InitialiseExternal must be called before any other SAW classes are used. This performs some startup steps that are usually triggered by the UI. The parameter is the path name of the folder where SAW is installed. This is required so it can load some of its data (such as its translation file); these cannot be found by the normal means since the EXE itself will have been copied to your project folder.

The *CreateNewDocument* on the second line creates a document and does some initialisation of defaults within it and returns the document object. The parameter is the size, in pixels, of the selection set. SAW7

documents support multiple pages, although only 1 is normally used. Most editing is done through the page – so the next line stores a reference to the first (and only) page (numbered “0”).

At the end of the function the generated document is saved to a filename provided. The filename shows as an optional parameter, but it must be provided in this case: it can only be omitted if the document was loaded from an existing file.

And the last 2 lines are not SAW-related. By default the console application will quit as soon as it reaches the end of the Main function – these 2 lines wait for a key press, so that it’s obvious that it has run.

Concepts

The content of a SAW document mainly consists of *Shapes* added to the page. There are a number of different classes for different shapes. *Item* (class *SAW.Shapes.Item*) is the class for a standard button. Other classes in the SAW.Shapes are used for standard graphical shapes – these are listed in their own section below. There are a number of others classes in SAW.Shapes either used internally, or more commonly as base definitions for the usable shapes (eg Lined is the base definitions for all geometric shapes providing the line thickness and colour, etc)

All of these classes provide purely graphical content without any command or switch logic. The interactivity for anything which is scanned is all implemented within a single class *SAW.Shapes.Scriptable* which is a wrapper which links to exactly one other graphical shape. This object contains the scripts, and some of the settings from the property window in SAW such as switch-swapping; the colour changes when highlighted, the numeric ID used in scripts to refer to other buttons etc. Any graphical object added directly to the page without a *Scriptable* is ignored by the scanning and is purely decorative.

Therefore the usual organisation for a SAW selection set is a *Document* containing one *Page*. That page contains a number of *Scriptable* objects. And each *Scriptable* contains an *Item* defining the appearance of the button.

The *Item* shape, only, can contain other items. Where buttons are added within other buttons or popups in the SAW graphical editor that is reflected in the data structure. In this case the page contains a *Scriptable* for the container, which contains the outer Item. That *Item* then contains several *Scriptable* objects, each containing another *Item*. (And they can contain further items, if desired).

Transactions: a number of the SAW functions accept a *Transaction* parameter. This is used internally for change-tracking to support Undo. For external code this can be *null* in all cases.

Images and sounds: these must be added from files (not .net objects in memory). They are always added initially to the SAW document, which keeps a cache of the media used within it. When the file is added to the document it returns a reference object which can then be assigned to individual items. When a selection set is saved it automatically discards from its cache any media that is no longer used anywhere. If the same file is added more than once only one copy is retained (the document uses checksums to detect duplicates)

Coordinates and Geometry

All coordinates are expressed in pixels, and always use floating point rather than integer numbers (ie PointF and RectangleF classes rather than Point and Rectangle). This allows scaling without rounding errors.

The coordinate space does not, however, match the one presented to the user in the editor UI. In the editor the origin is top-left with increasing numbers going right and down. Internally the direction of the values is the same (as this matches Windows). However for historical reasons the internal origin is at the bottom left of the page. **Therefore all Y coordinates should be negative**. (0,0) is the bottom left origin. (10, -10) is the point diagonally in from this (ie up and right). (0, -pageHeight) is the top left.

All angles used internally in SAW are in radians, counting clockwise from north.

There is a static class SAW.Geometry which contains a number of functions and definitions to support manipulation of points, rectangles, angles etc. In particular Radians(angle) returns the radian value for an angle expressed in degrees.

Adding normal buttons

The UI item

As described above a normal SAW item is in 2 parts. The UI object is created specifying its bounding rectangle and, optionally, the text:

```
SAW.Shapes.Item item = new SAW.Shapes.Item(new RectangleF(50, -200, 100, 100),
       "Example");
```

All other features are optional. An image can be added this: (the sound is part of the interaction, however – see next section)

```
item.Image = doc.AddImageFromFile(@"d:\temp\example.jpg");
```

(technical note: AddImageFromFile returns an object of class SharedImage. The item.Image property requires a SharedReference<SharedImage> object. The SharedReference<> supports the linkage while loading and saving the document. However a SharedImage can be assigned directly where the SharedReference is required – SAW defines implicit conversions so that the SharedReference part is created automatically and can be largely ignored externally)

There are a number of fields which can be changed to affect the presentation of the Item:

Title	The text on the button. Can also be provided when the Item is created.
TextRatio	The proportion of the button which is occupied by the text. The remainder is allocated to the image – which will scale to fit, but without changing the aspect:ratio, so will not fill its area unless the shape happens to match perfectly. This and Arrangement have an effect if both are provided and both TextShown and GraphicShown == true.
Arrangement	The relative positioning of the image relative to the text. The Superimpose value should not be used alone, but only ORed with one of Below/Above/Right/Left – it makes the image fill the button, with the text still placed as normal according to TextRatio.
TextAlign *GraphicAlign*	The positioning of each element within its area. Ie TextRatio and Arrangement are used to calculate the rectangle allocated to the text; and then TextAlign is used to draw the text within those bounds. Values should always be a combination of a horizontal and vertical component, eg: `item.TextAlign = Alignment.Centre | Alignment.Middle;`
GraphicShown *GraphicOnlyOnHighlight*	Whether the graphic is used. The latter only applies if GraphicShown == true.
TextShown	
StyleType	Which of the styles defined in the document is used for this, if styles are enabled.
BorderShape	One of the values Rectangle/Rounded/Ellipse which defines the shape the border is drawn (and the shape of the filled area even if no border). Note that for non-rectangular shapes, especially the ellipse, the software does not prevent content from spilling outside the area – it will be contained within a rectangular border, but does not follow curved parts of the border inside this.

The non-selected colours can also be defined here, but see *colours and selection highlighting* below.

The *Item* created is not usually added directly to the page...

The functional part

The scripting and logic is kept in a separate object, which also needs to be created. The UI element to be scripted must be specified when creating this:

```
SAW.Shapes.Scriptable scriptable = new SAW.Shapes.Scriptable(item);
```

(where item is the object created in the previous part). For a normal selection set, it would usually be *Items* which are specified here; however other graphical elements can be used instead – see below.

The main scripts are described in the next section, but the *Scriptable* has a number of other fields which control the behaviour of the item. Many of these relate to elements in the property editor in SAW designer.

SAWID	The numeric ID which can be used to refer to the item from scripts on other items. This defaults to 0 when created, but *Page.AddNew* will reassign this to a unique number if it is still 0, and if the document being worked on is the SAW global current document (which is the case for the last document returned by SAW.Generator.CreateNewDocument)
AutoRepeat	The repeat from older versions of SAW. If *true* the main Visit script will repeat if the switch is held
RepeatTimeout	The repeat added to later versions of SAW6. Defaults to -1. If >=0 then the 3 custom repeat scripts are used. The value is the repeat time in milliseconds
Shown *Popup* *AutoRepeat* *ResetSwap*	All these match the checkboxes in the editor
OutputText *SpeechText*	Text for output to the keyboard or speech. If not defined the label/title on the UI element is used.
OutputAsDisplay *SpeechAsDisplay*	These represent the same checkboxes in the editor. Note, however, that this is mainly for use by the editor. If generating a set, you should assign OutputText/SpeechText to the correct value – assigning true to these will have little effect on generated content.
Sound	The recorded sound, if any. Like any image, this must be added to the document cache: `scriptable.Sound = doc.AddSoundFromFile(@"d:\temp\example.wav");`

Scripts

The most important part of the *Scriptable* are the scripts – potentially 6 of them. There is a field for each:

- VisitScript
- SelectScript
- NextScript
- PreRepeatScript – as in the UI the last 3 are only used if RepeatTimeout >=0
- RepeatScript
- PostRepeatScript

All of these are initially null; in which case the default in the document, if any, is executed. If defined they contain Script objects, and the entire script can be constructed using objects:

```
scriptable.VisitScript = new SAW.Shapes.Script();
scriptable.VisitScript.CommandList.Add(new SAW.Commands.CmdBeep());
scriptable.VisitScript.CommandList.Add(new SAW.Commands.CmdFlashItem(3));
```

which beeps and flashes item 3 on the page. The command objects can be provided directly when creating the script object as any array or List:

```
scriptable.VisitScript = new SAW.Shapes.Script(new SAW.Command [] {new
       SAW.Commands.CmdBeep(), new SAW.Commands.CmdFlashItem(3) });
```

However the script object itself can also be constructed using the text version of the script:

```
scriptable.VisitScript = new SAW.Shapes.Script("beep\r\nvisit 3");
```

Technical note: the \r\n is the newline in c# which is required between separate commands in the same script.

There is also a shortcut defined in SAW which allows the text to be assigned directly where the Script object is expected, and the object will be created automatically. So this:

```
scriptable.VisitScript = "beep\r\nvisit 3";
```

also creates exactly the same result as the previous versions. This does create script and command objects still, so to read the text form back from SAW you would use:

```
… = scriptable.VisitScript.GenerateScript();
```

If the script is fixed the text version is easiest; but if the parameters in the script are constructed from variables in the generator code the objects may be easier.

The Script object also has a Visit property which specifies the VisitType, and if needed, the ID – which works exactly as in the SAW application. There is also a RunDefault field which controls whether the page's default scripts, if any, are run before the script specified here.

All the SAW command objects are in the SAW.Commands namespace. Any classes in this namespace starting with an underscore are base classes providing shared functionality between commands and cannot be created themselves. These are not individually listed here as there are a lot of them, and many are used only rarely. The complete list can be seen in the development environment prompt after typing "new SAW.Commands.", or in the SAW code where it constructs the meta data (currently line 70 onwards):

https://github.com/stuart2w/SAW/blob/master/SAW/SAW/CommandList.cs

Some commands have constructors which define any values needed, such as the SAW.Commands.CmdFlashItem(3) above. Other commands only have a base constructor and parameters need to be added individually, for example:

```
var move = new SAW.Commands.CmdMouseMove();
move.ParamList.Add(new SAW.Commands.IntegerParam(1));
move.ParamList.Add(new SAW.Commands.IntegerParam(0));
scriptable.VisitScript.CommandList.Add(move);
```

The ParamList field on the command contains generic parameters where custom fields are not used. Individual parameters are of type IntegerParam, StringParam, FloatParam (non-integer numbers) and BoolParam. However, again, writing it out in full, as in the above example, is unnecessary as implicit conversions exist, so normal c# integer, string, float and bool values can be used directly:

```
var move = new SAW.Commands.CmdMouseMove();
move.ParamList.Add(1);
move.ParamList.Add(0);
scriptable.VisitScript.CommandList.Add(move);
```

Other possible graphical elements

As well as the SAW Item class, there are a number of other classes representing simple shapes which can be added to the page. They can be added to the page directly:

page.AddNew(new SAW.Shapes.Line(new PointF(50, -50), new PointF(100, -50)));

or can be added to a Scriptable object and then become scannable and have scripts attached to them.

All of these classes are in the SAW.Shapes namespace, like the Item, so must either be named in full (SAW.Shapes.Line), or the “SAW.Shapes.” can be omitted if “using SAW.Shapes;” was included at the top of the file.

- Arc
- Circle
- Ellipse
- FloatingLabel: a text label attached to another shape which will move with the other shape
- Group: a set of shapes which are grouped together to act as one. The individual shapes must be created first. No coordinates are provided for the group – its bounds are the bounds of the contained shapes. This should contain only simple graphical shapes. The behaviour if *Item* or *Scriptable* objects are included is undefined.
- ImportedImage: an external image such as a bitmap loaded from a file. This supports any image type supported natively by .net (bmp, png, jpeg, wmf – although with limits) plus svg. Note that the extension on the file must be correct as this is used to determine the image type. The document object must be provided when creating this, as the entire image file is added to a cache in the document (so no link to the original file is kept). Adding the same image twice will only keep one copy of the byte data for the image. The image can be retrieved later using myImage.GetImage().GetNetImage() (the first GetImage() returns an internal SAW class storing any memory-buffered image). If the image was SVG this will render it to a bitmap first and return that.
- IrregularPolygon (formed from straight lines joining a series of points, always closed)
- Line: a single straight line between 2 points
- Parallelogram
- PolyLine: a series of straight lines between points – the difference to IrregularPolygon is that this shape is not closed and not filled
- OrthogonalRectangle: a rectangle that cannot be rotated
- RectangleShape (note the name: it is not called “Rectangle” as that would conflict with System.Drawing.Rectangle). This version can be rotated, and can be specified using a rectangle, or for a rotated one, any 3 consecutive points.
- Rhombus
- Square
- TextLine: plain text placed on the page. It is placed below a baseline, and will wrap if wider than the specified line. The normal constructor takes a rectangle. Note that it is unbounded on the bottom – the text will flow down indefinitely. A second constructor takes 2 points, which need not be horizontal, and can be used for rotated text.
- Triangle

Each of these has a default constructor

```
new SAW.Shapes.Line()
```

and one with parameters:

```
new SAW.Shapes.Line(new PointF(50, -50), new PointF(100, -50))
```

When generating content externally, the parameterised one must be used. The first is used only within the UI when the user draws the shape. The Intellisense within c# should show the parameters.

Styles and selection highlighting

The standard *Item* has 3 fields which define the style when in its non-selected state:

- FillStyle - colour of the fill (and also potentially a pattern)
- LineStyle - colour and thickness of the (border) line
- TextStyle – font name, size and colour

The other graphical shapes use the same fields, but not all shapes have all 3. So a simple Line shape has no FillStyle since it is not filled.

The main property of the first 2 is the colour. This can be assigned using a .net Color (note the spelling), either using a named one, or by specifying (A)RGB components. These 3 are all identical:

```
item.FillStyle.Colour = Color.Red;
item.FillStyle.Colour = Color.FromArgb(255, 255, 0, 0);
item.FillStyle.Colour = Color.FromArgb(255, 0, 0);
```

FromArgb requires 3 or 4 values. They all range from 0-255. If 4 values the first is the transparency, with 255 (the default) for a solid colour and 0 being transparent where the RGB values have no effect. This can be omitted. The remaining 3 are the red, green, blue components.

The line colour works just the same way. The line thickness is assigned as a number of pixels:

```
item.LineStyle.Width = 1.5f;
```

This doesn't need to be integral, but the SAW editor will round back to an integer any item that is edited within SAW.

Technical note: The "f" is required in this code since SAW generally uses float values and c# defaults literal values to double precision, which cannot be assigned to single-precision floats. The 'f' suffix indicates it is a single-precision float value

This styling on the graphical object is for the non-selected state. The style when selected is set on the *Scriptable* object no matter what the content element is. (And therefore inactive, graphical-only elements which are not contained in a *Scriptable* have no selected style). The *Scriptable* has a HighlightStyle field with an object containing the 4 values that can change when highlighted. Eg:

```
scriptable.HighlightStyle.LineColour = Color.Red;
scriptable.HighlightStyle.FillColour = Color.Black;
scriptable.HighlightStyle.TextColour = Color.White;
scriptable.HighlightStyle.LineWidth = 2;
```

When the item is scanned these will be used to update whichever fields are possible in the graphical item; any that don't apply are ignored. So, for example, the HighlightStyle.FillColour will be disregarded if the content is just a simple Line.

Nested items

In the SAW editor you can place buttons within other buttons to make groups which can be scanned. The sub-items are added to the *Contents* of the containing SAW item:

```
Item container = new Item(new RectangleF(50, -450, 400, 400), "");
Scriptable containerScript = new Scriptable(container);
page.AddNew(containerScript);

Item item = new Item(new RectangleF(100,-400,100,100), "One");
Scriptable scriptable = new Scriptable(item);
container.Contents.Add(scriptable);

item = new Item(new RectangleF(250,-400,100,100), "Two");
scriptable = new Scriptable(item);
container.Contents.Add(scriptable);
```

The SAW.Shapes has been omitted here – assuming a “using SAW.Shapes” command was at the top of the page. The sub-items could be written on a single line if they are this simple:

```
container.Contents.Add(new Scriptable(new Item(new RectangleF(250, -400, 100,
       100), "Two")));
```

however in most cases in order to add scripts etc, it would be organised as in the earlier example.

Note that the sub-items are added to the *Item* of the container, not the *Scriptable*.