

GUI Scripter's Notes

for StarSiege: Tribes

These notes are intended to make it easier for a Tribes scripter to put together a set of gui options and use them with NewOpts. The descriptions and instructions here are inconsistent and rather informal, but so am I. I tried to write for scripters who know little or nothing about gui editing, but I may have taken some things for granted. I sincerely hope you find them useful. Should you have any comments, suggestions, ommision reports, or bug reports, please send them to me at <http://www.cetisp.com/~thammock/scripts>.

Thanks to (in alpha order) Adrenfreak, Cowboy, Crunchy, DS.Poker, GrimJack, GrymReaper, Magic, LabRat, Phoenix, Presto, Strange and Writer, all of whom, whether they knew it or not, contributed ideas and knowledge that resulted in NewOpts.

Enjoy,

Tim 'Zear' Hammock

PART 1: CREATING A GUI

Before you begin, I should let you know that creating your own options page requires the creation of a .gui file. When I complete DynOpts, you will be able to create your page with script alone, but that is still under development.

Note: I did try to automate some of these steps, but ran into many problems, most notably, a huge number of stack faults in Tribes. Hey, I tried, OK?

Also, I use the terms 'component' and 'object' interchangeably. Object is the correct term, but I'm a Java programmer, so I forget and use component a lot. Please be understanding.

The first step is to create another config directory. Trust me, this makes a big difference. Rename your regular config to config.play or something like that, then create a new config directory. If you do not have a <Tribes>\gui directory, you need to create one. You can then rename the config directory to config.dev (or something like that) and rename your original config directory to config again when you want to play.

Install the Presto Pack (v0.93) to the new config directory.

Install NewOpts - remember to unzip this to your <Tribes> directory with subdirectories, as NewOpts will need to put a file in your gui directory.

Open the <tribes>\config\NewOpts\NewPrefs.cs file and go to the very bottom. You will see three lines that call NewOpts::register(). Comment these out (put a '/' before each) and save the file.

Start Tribes and go to the scripts section of the options. You will see a combo box and a frame on the page. This is why you created a new config directory.

Note: you MUST be in the options gui at this point, or these instructions get too complicated.


Type the following in the console ():

```
editGui();  
GuiEditMode(MainWindow);  
cursorOn(MainWindow);  
tree();
```

That gives you the full compliment of gui editing tools, as provided by Dynamix. You will see three windows. There will be a tree-view window, a 'GuiEditBar' window, and a properties sheet. Let me explain them a little before you go messing around - you can crash Tribes pretty quick with this stuff.

The tree window shows all the objects currently loaded by the game. Almost any object can be a container (technically speaking) so the objects can be nested. If you click on an object then right click on it, you will get a popup menu for that object. Mostly you will choose edit. ALWAYS click left-click on an object before right clicking on it, or you may end up performing the operation on the wrong object - selection does NOT automatically get set to the component you right-clicked on and some operations will shift the selection on you. When deleting objects prepare for the occasional crash (more like Tribes just disappears). You will be saving your work ALL the time (I'll get to that too). You can also move a component from one container to another by dragging and dropping. This is also crash prone. I think most of the crashes are due to deleting or moving an object that Tribes is updating, but I can't prove it. One more thing about the tree - it does not update to reflect changes after most operations. If you need to see the change in the tree view, collapse the tree at a point above where the changes should show, then re-expand. The changes will show after that (but it can sometimes get tedious, so I fly 'blind' a lot).

In the tree view, each object is described by a line like:

 8345,"OptionsGui", SimGui::Control

The folder symbol could also be a green dot or a brownish-black cube. It doesn't really matter for this type of work - you will always deal with the folders. The number (8345 here) is the object's ID number. Almost any function call that takes an object as an argument will take the number. These numbers are not guaranteed to stay the same. They are auto-numbered in the order of creation, and since script additions and changes can change object creation order, they will change on you. We can work around this later. The next thing is the object's name in quotes ("OptionsGui"). While many functions will take the name instead of the ID, sometimes it is unreliable. Many of the objects in the tree do not have names (""). You will want to name many of the objects you create (more later). The last thing on the line is the object's type (SimGui::Control).

The gui edit bar has buttons for aligning objects, etc. The only one I found useful is the one labeled 'Edit'. You may have noticed that in the main Tribes window, you can now drag and resize objects with the mouse. If you click 'Edit' in the bar, you will toggle this mode on and off. You will do it a lot.

The last window is the property sheet. When in 'Edit' mode, any object you click on will display its properties here. I will cover the different properties later, but for now, you should also know that this window has a 'twin'. When you right-click an object in the tree view and choose edit, another identical window will appear (sometimes it will be at bottom right edge of the screen and you will have to collapse your Windows task bar to get to it). These have the same functionality, but one is for the main Tribes window, and the other is for the tree view. Don't confuse them - it is easy to do and leads to situations where you keep setting a property, but it never changes on the property sheet (duh, wrong sheet), because even when the two property sheets both display properties for the same object, they are not synchronized.

One last thing about all these windows – DON'T close any unless you won't need them until after a restart of Tribes. Some won't come back, others will, but with weird side effects. Of course if you accidentally close one, you can save your work and restart Tribes, but trust me, after the umpteenth time, you will start to avoid closing them.

These tools are provided by Tribes, not me, and they are probably irresistible about now if you've never seen them before. As long as you don't make a call to `storeObject()` or mess with anything in `PlayGui`, you can do anything a restart won't fix. So go ahead and play with the tools, then come back when you are bored.

Did you have fun? Cool, huh? OK, back to work.

Collapse the tree view all the way, then re-expand just one level. Look for an object named "OptionsGui". This object and its contents are actually the contents of the file `gui\Options.gui`. It's not important at this point but interesting to note. Expand OptionsGui, and you will see it has a child of the type `FearGui::FGShellBorder`. This is the component that provides that annoying border Cowboy and I couldn't deep-six in `CMDHud`. Expand it and look for an object of type `FearGui::FearGuiBox` (fifth child down). This component provides the green double-line box around the stuff inside the shell border (I like this component and use it in my own stuff). Expand it and look for an object named "NewOptsPage" (last child). This is transparent, but it covers the area with the cool b&w design to the right of the wide green bar.

Expand NewOptsPage and look for an object named "NewOptsFrame" (first child). This is the frame with the thin green border around it, and for the most part is your options playground. If you expand that, you will see it contains one object – "NewOptsChooser". You will learn to expand to this point pretty quickly, because this is where you do all your work (not that there isn't other interesting stuff in there).

When a page is registered with NewOpts (explained later), all the components are put on the page together, and only the current page's components shown. If I didn't do this, `CFGButtons` would not work correctly. If more than one page were loaded at a time, you would have possibly hundreds of components here. This is why you created a new config directory and why you commented out the lines in `NewPrefs.cs`. Since you are going to save the page to create yours, you don't want to have to delete the majority of them, nor do you want them cluttering up your 'workspace'.

NewOptsPage has dimensions of 294x306. I could have made it bigger, but if I did, some low-resolution players couldn't see the whole thing. You will notice that Dynamix did this all around. I thought of adding a scroll bar, but that adds an object layer that breaks `CFGButton` inter-communications. And making a gui is enough work without making a different set of pages for each screen resolution (and that would be your work to do not mine or the script's). You can just make another page if you run out of room – don't worry.

Unfortunately, part of the space you get to work with is taken up by the NewOptsChooser. I made it as small as possible, but it pretty much had to go in here. Leave it there for now, and then when you finish your page you can delete the chooser before saving to conserve on your file sizes. Not that it makes much difference - a few more bytes is all. The objects themselves are not saved, just descriptions, so they save pretty small. Also, my page loading code is smart enough to ignore this object if you load in a second copy with your page. For now you need it there so you don't accidentally place an object over it.

Now you add components for your options page. Go back up to the level of NewOptsPage and troll through its sibling's children and grandchildren, ad nauseum. Find the component type you want, click on it, right-click on it, and choose copy. Click on NewOptsFrame, right click on it, and choose paste. Your first component – aren't you proud? You can probably see it in the main Tribes window. Resize and reposition, fill in properties, and repeat until done (there's lots of info on what to set how in section 2). Of course there are a few things to know and do along the way.

Save a LOT. Here's how: at the console, type:

```
storeObject(<ObjID>, <filename>);
```

where <objID is the object number for NewOptsFrame (get it from the tree view), and <filename> is "temp\\YourGuiName.gui". You can save to config or to temp. I suggest temp. Any other place will result in your being told that your chosen directory cannot be written to. The helpful little dialog will let you retry or give up, but retrying doesn't let you change the directory, so it fails again, and again. Canceling will dump you unceremoniously out of Tribes, and there went any unsaved work, down the digital drain. It hurts on a physical level when you lot of work with no saves.

Once you have typed this once, you can just up-arrow back to the line and do it again later, saving you the typing (important when you work from a fast hunt-and-peck like me).

OK, now you want to know how to load your gui for testing and to re-edit it. First create a script file for your script, unless you already have one then just use that one). At the *end* of the file (I'll explain in the NewOpts::register() description in Part 3), add the line

```
NewOpts::register("YourCoolGui", "temp\\YourGuiName.gui", "", "", TRUE);
```

I'll explain what that does in the NewOpts.cs API section. For now, just put that in, save the file, and make sure it gets Include()'ed at startup like normal. Now when you restart Tribes (you *did* save your gui, right?), whatever you did in your gui will be there in the options. Woohoo!

A few words about the location of your gui file. I try to put it out of harm's way. For instance, the NewPrefs guis are in the config\\NewOpts\\Gui directory. Let's not clutter up the config directory, OK? Also, you have saved your gui to the temp directory so far. When you distribute your script, you can put it where you want (within reason) and you will just change the path to it you pass in the call to NewOpts::register();

Making a Dialog

The best way to make a dialog is to use DialogOpts::pushDialog() to push a dialog of the type you want, then modify and save it as your own. You will likely want to choose the LARGE type dialog. The dialog object will appear (after you refresh the tree) near the end of the list (of the children of the root) in the tree window. Look for a SimGui::Control that contains a FearGui::FGDlgBox.

Note: When editing a dialog, you will find that you cannot move or resize its components by dragging with the mouse in the main Tribes window. Dialogs are on a different plane, so to speak, so while you will be able to see it just fine, you will need to do all your editing through the tree view's property list. Clicking in the main Tribes window will only select the component behind the dialog. It's not so bad, just annoying.

Inside the FGDlgBox, you will find a FearGui::FearGuiScrollCtrl – go ahead and delete that from the tree (unless you need it). Add your component using the tree or newObject(), and set them up as you wish. When you save the dialog, you need to save the SimGui::Control it is contained in. You can use GuiPushDialog() to make it appear (it will be modal), and GuiPopDialog() to toast it when you are done. Be sure and get the data from the dialog's components first though, because when you pop it, it will disappear from memory, and then it's too late.

Despite what I set up in DialogOpts to allow you to populate a dialog from script at runtime, you probably will not want to do this. At this time I have not figured out how to create a label (FearGui::FGSimpleText) using newObject() that has text other than "Test String". I'm working on it though.

For a good example of using a dialog this way, check out the file `NewPrefsIntervals.gui` and the functions `NewPrefs::pushIntervals()`, `NewPrefs::intervalValidate()`, and `NewPrefs::closeIntervalDialog()`.

PART 2: OBJECT INFORMATION

Now for some object information. First I'll discuss properties, then the objects themselves, and finally the list of general-purpose functions.

Property info:

Note: Not all components/objects actually use all of the properties (sometimes they are not shown in the property list, sometimes they just don't work). You will want to experiment a little.

Name: You will want to set this if you want to be able to manipulate the component from script. This will be just about everything. Pick a unique name, with your script's name at the head, like "NewPrefs::Playmode". That way if someone else picks the same name, they won't be likely to use the same script name too and you will avoid problems. Most labels will not need anything here. If you do not name the object, NewOpts will name it for you when it reloads it. The name will be "<pagename>::noname::<number>". Annoying as hell to see, but we can't have duplicate names in the game, can we?

Owns Objects: OK, I just always check it (actually it always seems to be checked already, so I ignore it). What's up here? I can't claim to know.

Pos TopLeft (x, y): This is where the upper left corner of the component will be place relative to the upper left corner of its containing object. Two number separated by a comma, spaces optional.

Extent (width, height): Like the previous property only this one is the size (duh).

Control ID: Set to <NONE>. This is used to reference the object internally by Tribes. We will use the object's name and/or ID. Same problems as Help Tag (below).

Horizontal Sizing: I leave this one at 'Resize Left'. You can change it (does some cool things), but the information can be lost in the save, and has limited use anyway, because we aren't in a resizing container anyway.

Vertical Sizing: I leave it at 'Resize Bottom', for reasons that match Horizontal Sizing.

Visible: Do you want to be able to see it?

Delete On Lose Content: Not really sure – I just leave it checked.

Console Variable: Put the name of a console variable in here (\$foo) and the variable will be set to match the state of the object when the user interacts with it. Really only useful with a checkbox.

Console Command: This is a biggie. When the user interacts with the object (presses a button, moves a slider, etc.) whatever is typed here will be executed. Use it to call a function of yours where you deal with updates to component states. Example – "NewPrefs::onPlayModeSelect();". Don't forget the semicolon.

Alt Console Command: This is like Console Command, but will be triggered on the alternate events, like reasing a click on a button. Not used a lot, but very handy when you need it

Opaque: Not all components support this, but for those that do, it makes the difference between a transparent (and invisible) component and one you can see. If opaque is checked some components allow you to set the colors for fill, selected fill, and disabled fill.

Border: Much the same as Opaque. Again, not all objects support it.

Help Tag: Set this to <NONE>. We could use it, but it would require someone to dole out help ID's. There are only so many, and EVERY help popup needs its own. Collisions mean only the first registered under that ID will work (other popups using the same ID will show the same, wrong, help popup), plus you will get console error messages. This is why I created help dialogs.

Active: Will it respond to user activity or is it just for looks?

Message Tag: Used internally by Tribes. I haven't devined how to use it, but it would have the same limitations as Help Tag, if not more. Set to <NONE>.

FontName Tag: Leave this and its next two siblings alone. You can get some cool fonts, but when you save and reload, it'll revert back to the default anyway. Bummer.

Text tag: If there is a label in the list appropriate for your use, use it. Most of the time there isn't. Same problem as Help Tag – set to <NONE>.

Text: If Text Tag is set to none (blank), you can enter a label here. Limited usefulness except with label objects. For instance, you can set it in a combobox, but you can only set one, and when you load in more choices from script, the one you enter here is toast.

Is an ON/OFF Switch: Use this for checkboxes, otherwise they will come unchecked every time.

Radio Set: I used this once in SnipeHud, but it has the same setup and problems as Help Tag. I probably should have done it in script. A radio button group manager is slated for DynOpts, but for now, just use Console Command to set off this type of behavior in script.

Lo-Res Position: Who uses low res? Seriously, I don't know about this one.

ACTIVE: This is different from Active, above. It's all caps and indicates whether to draw the component like it has been clicked on but not released yet. More importantly, it defines whether a checkbox should start out checked.

Mirror Console Var: This one rocks. If this is a checkbox and you have Console Variable, above, filled in correctly, the checkbox and variable will always be in sync. Checked == TRUE and unchecked == FALSE. I used the hell out of these two.

Alignment: No clue. This one never made any noticeable difference for me. Let me know if you learn something new here.

Text V Pos Delta: You can use this to offset the text label vertically by number of pixels. I don't see the need, but you might find one.

BMP Root Name: Some components, like buttons, can have three images: normal, selected, and inactive. By entering the right name here, you component will have them too. I haven't used it because I couldn't figure it out for my own bitmaps, and I haven't really needed it. Let me know if you learn something new here.

Min Value (float): Sets the low end of the scale's value for a slider. Problem was it acted flakey, when it worked at all. I usually left this and the next two alone. That gives you values from 0.0 to 1.0 (continuous) from the slider, and I just do the math conversions to the range I really need (and back) in

script. We really don't need to save cpu cycles in the options gui (a slider on the play gui!?) and this way worked reliably.

Max Value (float): See Min Value.

Stepping Value: How many stops on the slider do you want? Same problem as Min Value though.

Numeric field only: This is used by the FearGui::TestEdit (an editable text field). Will allow only numeric characters to be typed in if checked. Oddly enough, numeric seems to be defined as digits and the negative sign, but decimal point is not allowed. Bah!

Max Str Len: Use this to limit the number of chars that may be entered into a FearGui::TestEdit. Maximum usable value is 255.

Bitmap Array Tag: In a scroll control, this determines the images used for the scrollbars themselves. Leave it at IDBPA_SCROLL2_SHELL unless you really like ugly scrollbars.

Handle Arrow Keys: I assume this means the scrollbar will respond to the arrow keys on the keyboard. I just leave it checked and have not tested it.

Constant Sized Thumb: Determines whether the scrollbar thumb is a constant size or varies in size to indicate the percentage of the total area currently displayed. I personally can't think of a reason to check this.

Horizontal Scrollbar: Determines whether the scrollbar is always on, always off, or on only when the content exceeds the size of the viewing area.

Vertical Scrollbar: Same as the Horizontal Scrollbar property, except this determines behavior for the vertical scrollbar.

Header Dimensions: Unknown purpose, but I haven't needed it.

Border width: The width of the border around the component, in pixels (more or less).

Action Map: When setting up a CFGButton, this determines which action map to add the keybinding to.

Make Action: With the CFGButton, used to specify the IDACTION to associate with the make action of the bound key.

Make Value: With the CFGButton, the value to pass with the make IDACTION.

Break Action: With the CFGButton, used to specify the IDACTION to associate with the break action of the bound key.

Break Value: With the CFGButton, the value to pass with the break IDACTION.

Action: Unknown.

There are more properties for other components, but that pretty much covers what you will need for options gui's.

Component Info:

CFGButton:

Use a FearGui::CFGButton. If you want the bound key to trigger an IDACTION (the equivalent of a call to bindAction()), set the Make Action, Make Value, Break Action and Break Value properties accordingly. If you are wanting the equivalent of a bindCommand() call, set Make Action and Break Action to <NONE>, set Make Value and Break Value to 0 (zero), set Console Command to the name of the function you want called on key make (complete with arguments and the semicolon), and Alt Console Command to the name of the function you want called on key break (also complete with arguments and the semicolon). Either way you go, set Text to the text you want for a label, and make sure Message Tag is set to IDCTG_OPTS_CTRL_CFG_SELECT.

Important: The text label you use **MUST** be unique among all CFGButtons. The alert to the user that a CFGButton has been unbound (due to defining the same key/button in another CFGButton) is dependent on these labels being unique.

Label:

Use a FearGui::FGSimpleText. It will resize automatically depending on the text it displays. Set the text with 'Text'; if you check 'Active', the text will be white, otherwise it will be beige. To set the text from script, use:

```
Control::setValue(<objname>, <text>);
```

To get the text, use:

```
Control::getValue(<objname>);
```

Button:

Use a FearGui::FGUniversalButton. This one will be transparent if you don't supply an image. Using the in-built images is limiting, so I've been trying a few ideas, but for now, I just put an invisible button over the top of an appropriate white label. ConsoleCommand is executed on click, and Alt Console Command on release. Set it visible (not opaque) and Active, but not ACTIVE (confusing having two properties the same name, huh?).

Checkbox:

Use a FearGui::FGUniversalButton. This is a specialized button. Check Is an ON/OFF Button, and make sure BMP Root Name (the text box, not the combobox) is set to BTN_CheckBox (the underscore won't show, don't worry if you can't see it). If you set Console Variable to a global (boolean) variable name and check Mirror Console Var, the checkbox will always match the variable, automatically (checked == true, unchecked == false), even when the user changes the checkbox's state – very cool and no script needed at all for that. If you want to have more control, Console Command's contents are exec'ed on check, and Alt Console Command's on uncheck. You could set up id tags for radio button behavior, but that's really problematic as explained above (in Help Tag property). I suggest going with full control of the check/uncheck and implementing your own radio behavior. DynOpts should provide that framework for you later. Oh yeah, set the state of the checkbox with

```
Control::setValue(<objname>, <boolean>);
```

And get the state with

```
Control::getValue(<objname>);
```

Slider:

Use a FearGui::FGSlider. Whatever function you set in Console Command will get called every time the slider is adjusted. Use

`Control::setValue(<objname>, <value>);`

to set the slider's position (range of 0.0 to 1.0 by default) and use

`Control::getValue(<objname>);`

To read it back after the user changes it. Then you can multiply and add, etc. to convert that to the range you really want. I like to put a label above the slider that gives the current value in brackets – I hate making slider adjustments without knowing exactly what I'm setting it to. Also, the wider the slider, the finer the usable increments, so if you are using a range of, say, 1 to 1000, you'll want the slider as wide as possible.

Combobox:

Use a `FearGui::FGStandardComboBox`. At initialization (I explain that later in the API) do a

`FGCombo::clear(<objname>);`

Then load your list in by doing

`FGCombo::addEntry(<objname>, <string entry>, <index>);`

repeatedly, where <index> is a number. Just start at zero and increment that one. Keep track of entry/index pairs though, cause you use

`FGCombo::setSelected(<objname>, <index>);`

to set the selection, but you use

`FGCombo::getSelected(<objname>);`

to get the index number of the selected entry. Alternatively, you could use

`FGCombo::getSelectedText(<objname>)`

to get the text of the selection. The entries will be alphabetized automatically and I haven't found a way to turn that off. Console Command's code gets executed whenever the user makes a selection, even if they re-chose the previous selection.

Textfield:

Use a `FearGui::TestEdit`. Checking Numeric Field Only causes the textfield to only accept numeric characters. Interestingly, digits and the negative sign are considered numeric, but the decimal point is not (Zear scratches his head in confusion)! Max Str Len can be set to an integer value to limit the number of chars the textfield will accept (maximum is 255). If you supply a variable in Console Variable, the variable will match what is entered. Supplying a function to Console Command will result in the function being called EVERY time the textfield is updated, clicked in, or typed in. This is handy because the edits are not actually mirrored to the textfield itself until after the Console Command is called, allowing you to filter the text. Just get the text at that point by using

`Control::getText(<objname>);`

filter it removing or changing text you consider invalid, then send the modified text back using

`Control::setText(<objname>, <text>);`

The user will never even see the intermediate text – if they hit the x key and you edit out the x this way, it will seem as though the text field doesn't accept the x char at all. This is good for validating the info so as not to cause a problem when your script tries to use it.

Important: If you try to validate a FearGui::TestEdit while it is on your options page, you are almost *guaranteed* a stack fault. I have no idea why. The solution is to put this kind of stuff in a dialog. You will be able to validate safely and you can force the DONE button to be inactive until the FearGui::TestEdit contains valid text, making the user enter only information in the form you want it before continuing.

Formatted Text:

Use a FearGuiFormattedText. Notice no colons in that object's name. Weird huh? These are display only. The valid tags that can be embedded in the text (that I know of) are

<f0>	brown text
<f1>	beige text
<f2>	white text
<jl>	left justify
<jc>	center justify
<jr>	right justify
<bx,y:bitmap>	display a bitmap
<l#> and <r#>	indent # columns from left or right
\n	carriage return
\t	tab

To set the text use

```
Control::setValue(<objname>, <text>);
```

I assume that

```
Control::getValue(<objname>);
```

would get the text back, but I haven't tried cause, well, what would be the point? It's display only.

Textlist:

Use a FearGui::FGTextList. This is essentially the same as the Combobox, without the popup thing – it is 'popped up' all the time. Use

```
TextList::clear(<objname>);
```

to clear it. Use

```
TextList::addLine(<objname>, <text>);
```

to add lines to the list, and

```
Control::getValue(<objname>);
```

will return the currently selected text. I'm not sure how to set the selection (there is no obvious function), nor do I know about sorting, although I would assume

```
FGTextList::sort(<objname>);
```

would work, since the function exists (I haven't tried it – the one I used sorted automatically). Of course that is no guarantee – lots of obvious ones like this come back as unknown function, or worse yet, they appear to execute but have no discernable effect. How the heck are we supposed to figure out garbage like that? There is also a Don't Allow Duplicates checkbox in the property list for this one – does just what it says.

Blank panel:

Use a `SimGui::Control`. This is so simple – set the position and size, play with Opaque, Border and the associated color specifying properties, and well, that's about it. A useful feature of this one is that objects placed inside will not show any portion that falls outside the rectangle defined by the panel. I used this in `SnipeHud` to mask out the parts of the `TSCtrl` (the 3D view part of the hud) I didn't want to see. I've also used this to cover up ugly stuff I didn't want the user to see by making it opaque and setting the color to the same as the background color behind it. I used it for the header backgrounds for the dialogs in `DialogOpts` too. I'm still coming up with devious uses for this one.

Scrollpanel:

Use a `FearGui::FearGuiScrollCtrl`. This one is an odd duck, because you need the `SimGui::ScrollContentCtrl` inside it. As a matter of fact, if you create the `ScrollCtrl` using `newObject()`, the `ContentCtrl` will be added automatically. Set the Vertical and Horizontal Scrollbar properties the way you want and then drop a `SimGui::Control` into the `ScrollContentCtrl` (or a `Textlist`, or a `FearGuiFormattedText`, or whatever). If the component you drop in is larger than the `FearGuiScrollCtrl`, you will be able to scroll it around. I covered Bitmap Array Tag, Handle Arrow Keys and Constant Sized Thumb in the properties, above. I don't have any information on Header Dimensions – I haven't got to experiment with that yet. I'm sure I've either missed something cool or it doesn't work.

Frame:

Use a `SimGui::Control`, above if you want a one pixel frame, but if you want that cool green thin-line/thick-line frame you see in `Tribes`, use a `FearGui::FearGuiBox`. This thing works pretty much like the `SimGui::Control`, except it has the cooler border. 'Nuff said.

OK, that's as far as I'm going with the descriptions. That should give you what you need for gui options anyway. I want to finish this document this year, and the object types go on forever. Too bad half of the rest are hardwired and thus fairly useless. Feel free to experiment though – you might find something cool.

General Purpose Functions:

Here is a list of generic functions you may find useful. There are a lot more, but they get pretty esoteric, and I don't get a lot of them yet.

`newObject(<name>, <class>, <x>, <y>, <w>, <h>, <convar>, <concmd>);`

This can vary with some of the more unusual objects, but this is the basic form. I include this mostly because it is so basic, and sometimes I just want to create the object myself instead of hunting one up. The new Object will appear at or near the end of the list of children of the root object in the tree. You can then (usually) drag it to the place in the tree where you want it.

`<name>` can be empty quotes ("") if you don't care about the name

`<class>` the object type to create – I specified them in the last section.

`<x>, <y>` position of the top left corner relative to the top left corner of the containing object. Make it negative if you want, but you will not see some or all of it in most cases due to masking.

`<w>, <h>` width & height. No negatives, OK?

<convar> puts this string into the Console Variable property field

<concmd> puts this string into the Console Command property field

deleteObject(<objID>);

<objID> the object ID of the object you wish to delete. This is prone to crashing under some circumstances – SAVE FIRST!

addToSet(<set>, <obj>);

This is how you nest objects from script.

<set> the object ID of the set (a set is any object that can be a container) to add the object to
<obj> the ID of the object to be added to the set

Control::getActive(<objID>);

<objID> the object ID of the object you want the state of

Control::getText(<objID>);

<objID> the object ID of the object you want the text of

Control::getValue(<objID>);

<objID> the object ID of the object you want the value of

Control::setVisible(<objID>);

<objID> the object ID of the object you want to determine visibility for

Control::performClick(<objID>);

Control::performClick(<objID>, <val>);

Couldn't coax this one into giving me anything. Let me know if you figure it out.

Control::setActive(<objID>, <boolean>);

<objID> the object ID of the object you want to set the state for

<boolean> TRUE or FALSE

Control::setText(<objID>, <text>);

<objID> the object ID of the object you want set the text for

<text> the text the object should display

Control::setValue(<objID>, <val>);

<objID> the object ID of the object you want to set the value for

<val> the value to assign to this object; varies as to type

Control::setVisible(<objID>, <val>);

<objID> the object ID of the object you want to set the visibility state for

<boolean> TRUE or FALSE

Group::getObject(<set>, <index>);

Returns the Nth object from a set (container object). Returns -1 if the Nth object doesn't exist.

<set> the ID of the container you are looking in

<index> the zero-based index at which you expect to find an object

Group::objectCount(<set>);

Returns the number of objects in a set (container object).

<set> the ID of the container you are looking in

Object::getName(<objID>);

Returns the name of the specified object.

<objID> the object ID of the object you want to retrieve the name of

nameToId(<objname>);

Get the object ID of the object with the specified name. I have no idea what would happen if there were multiple objects with the same name. Let's assume that would be bad.

<objname> the object name of the object you want to retrieve the ID of

getGroup(<objID>);

Gets the ID of the parent, set or container this object is contained in.

<objID> the object ID of the object you want to get the parent of

setButtonHelp(<WindowName>, <ButtonName>, <HelpString>);

OK, I wanted to use this, but I can't figure out what <WindowName> should be. It isn't MainWindow like everywhere else. It isn't the name of the container the object is in, nor did it seem to be a Control ID. I'm stumped, but if anyone makes a breakthrough here, I'll surely add this capability to NewOpts.

PART 3: THE NEWOPTS API (SUCH AS IT IS)

NewOpts.cs

This is the heart of NewOpts. This file provides registration services as well as run-time functionality that will allow you to add gui options pages for your scripts.

NewOpts::register(%pagename, %file, %initfunc, %closefunc, %visall)

This is the function that registers your opts page so that it will be displayed. Due to a quirk in the way Tribes works, this line should not be executed until after *all* functions referenced in the Console Command and Alt Console Command properties of your gui have been defined. If you load the gui (which will happen almost immediately after this call) when the referenced functions have not yet been defined, Tribes does some really, *really* strange things. I don't have a debug version of Tribes of course, but objects and functions can get shuffled and deleted. I think Tribes is probably resolving the references and there is little or no error checking, so the tables get hosed and Tribes starts making the wrong connections among objects and functions. Of course, I'm just guessing...

Example: NewOpts.cs, lines 301-303

%pagename The name you want to appear to represent this page in the script combobox. Pick something unique, because only the first page registered under this name will be registered.

%file The name of the file your gui is contained in, with relative path.

%initfunc The function to be called to initialize your gui page when it is selected for display. This is optional (you can specify "" if you like).

%closefunc The function to be called to close out your gui page when the user moves on. This is where you would do any last minute updates to your script based on the user's choices. This is optional (you can specify "" if you like).

%visall TRUE or FALSE. If true, the NewOpts system will make all components in your gui visible when the page is displayed. If you need some to be left invisible, you will need to make the appropriate ones visible in your %initfunc, otherwise your page will be blank. All components on your page will be made non-visible automatically when the page is closed.

NewOpts::registerHelp(%pagename, %topic, %text)

This function registers help topics associated with your options page. The page itself should already be registered before calling this function. Currently all topic for a given page are concatenated into one huge string, but in a future version, I intend to do help a different way. For that, the topics need to be separate, so for compatibility, I'm starting that now. Registering at least one topic for a page will activate the help button for that page.

Example: NewOptsHelp.cs, line 7 (and many others in the same file)

%pagename The name of the page to associate with this help topic. It needs to be identical to the argument of the same name passed to NewOpts::register() for this page.

%topic The name of the help topic.

%text The text of the help topic

NewOpts::isValidString(%val, %charsAllowed)

This function allows you to determine whether a string is valid, as defined by %charsAllowed. Any characters in %charsAllowed are considered valid characters. A string passed in which is comprised entirely of characters in %charsAllowed will be considered valid and TRUE will be returned. Otherwise the function will return false. This allows you to filter user input for validity.

Example: if(NewOpts::isValidString(%userinput, "x0123456789abcdef") // valid for hexadecimal entries

%val The string to be checked for validity.

%charsAllowed A string containing the characters considered valid.

NewOpts::validateAlphaText(%val, %charsAllowed)

This function is also for string validation. This function takes %val and compares the characters in it to the characters in %charsAllowed. Any character not found in %charsAllowed is discarded, but valid characters are kept. The result of this comparison is returned.

Example: NewOpts::validateAlphaText("test", "st"); // returns "tst", since 'e' is not a valid character

%val The string to be checked for validity.

%charsAllowed A string containing the characters considered valid.

NewOpts::validateNumericText(%val, %decimalAllowed, %negAllowed)

This function is also for string validation. This is a specialized version of NewOpts::validateAlphaText(). It assumes that the valid characters are “0123456789”. If %decimalAllowed == TRUE then the character ‘.’ is valid. If %negAllowed == TRUE then the character ‘-’ is valid. The string %val, minus any invalid characters, is returned.

Example: NewOpts::validateNumericText(“-0.125x”, TRUE, FALSE); // returns “0.125”

%val The string to be checked for validity.

%decimalAllowed TRUE or FALSE; Is a decimal point a valid character?

%negAllowed TRUE or FALSE; Is a negative sign a valid character?

NewOpts::getIdFromSet(%set, %name)

This function searches %set for an object named %name. If the object %name is found, this function returns its object ID. If not, it returns a -1. I know this seems like an obscure function, but I needed it, and you might find it useful.

Example: DialogOpts.cs. line 164

%set The set (container) to search in

%name The name of the object to search for.

DialogOpts.cs

DialogOpts provides default dialogs for other scripts. These are modal dialogs, and only one may be shown at a time. There are three pre-designed dialogs and a general purpose dialog. To manually dispose of any of these dialogs, call

```
GuiPopDialog(MainWindow, 0);
```

DialogOpts::pushDialog(%title, %text, %color, %size, %type, %result)

This creates a general purpose dialog and displays it.

Example: DialogOpts.cs, line 142

%title The title string for the dialog. This text will be put into a FearGuiFormattedText component, so you may use the following tags:

<f0>	brown text	<bx,y:bitmap>	display a bitmap
<f1>	beige text	<l#>	indent # columns from left
<f2>	white text	<r#>	indent # columns from right
<jl>	left justify	\n	carriage return
<jc>	center justify	\t	tab
<jr>	right justify		

While you may use a \n character, I recommend against it as the title field is sized for one line of text only.

%text The body text of the dialog. This text is also put into a `FearGuiFormattedText`, so the above tags apply here too. The text area is scrollable.

%color I have provided 24 colors for the title bar, plus a black-with-white-border version for a total of 25. The colors are:

B_&_W (black with white frame)	BLUE
BLACK	CYAN
DK_GRAY	AQUA
MED_GRAY	BLUE_GRAY
LT_GRAY	CORN_BLUE
WHITE	DK_BLUE
RED	YELLOW
PALE_RED	PALE_YELLOW
DK_RED	PURPLE
GREEN	DK_PURPLE
LT_GREEN	BROWN
DK_GREEN	BEIGE
DLG_GREEN (dialog green)	

To specify the color, pass in one of the above all-caps color names. If you do not specify a color, it will default to B_&_W. You may want to consider altering the color of your title text above to contrast better in some cases.

%size `DialogOpts` dialogs come in two sizes: `SMALL` and `LARGE`. Specify one of those two all-caps strings to choose a size. If you do not specify a size, it will default to `SMALL`. Both dialogs are identical except for the height of the body text area. A `SMALL` dialog provides about 4 2/3 lines. A `LARGE` dialog provides about 13 lines. There is, of course, a vertical scroll bar in case your text exceeds the number of displayable lines.

%type This parameter can be one of **OK**, **DONE**, or **DONECHECK**. **OK** is the default where no type is specified. There will be a single, active OK button at the bottom. Clicking the button will close the dialog. **DONE** provides two buttons, **DONE** and **CANCEL**, at the bottom of the dialog, and both buttons will be active. **DONECHECK** provides both **DONE** and **CANCEL** buttons, but the **DONE** button is inactive (grayed out) until a call is made to `DialogOpts::activateDialogDone()`. This allows you to add components to the dialog, and by setting the Console Command field of your components to call a function you create, you can validate the data the user has entered before enabling the **DONE** button. When the call to `DialogOpts::pushDialog()` completes, it will return a handle to the dialog panel. If you do not pass text for the dialog body, you can use this handle to construct a custom dialog. Note that you should get any values from the components in a custom dialog immediately and then make a call to `DialogOpts::closeDialog()`, as the dialog will remain open until you do. This is not necessary with the **OK** type dialog, which will be immediately destroyed when the OK button is pressed.

%result By passing the name of a function in this parameter, you can receive a token indicating which button was pressed to close the dialog. The three possible values returned are **OK**, **CANCEL**, and **DONE**. The name of the function should be passed without parens or semicolon - when called, the function will be passed the result token as a single argument.

The three pre-designed dialogs are the Error dialog, the Info dialog, and the Help dialog. All three are `OK`-style dialogs. The Error and Info dialogs are `SMALL` and the Help dialog is `LARGE`.

ErrorDlg(%text)

The Error dialog has a red title area with the title "Error" centered in white.

Example: `ErrorDlg("Array out of bounds");`

%text The text to be displayed in the body of the dialog.

InfoDlg(%text)

The Info dialog has a blue title area with the title "Attention" centered in white.

Example: InfoDlg("There are no more enemies to crush!");

%text The text to be displayed in the body of the dialog.

HelpDlg(%topic, %text)

The Help dialog has a dialog-green title area with the supplied title centered in white. As mentioned above, this dialog is LARGE size. If you are providing help for gui options, please use NewOpts::registerHelp() instead.

Example: HelpDlg("Explaining General Relativity", "Sure thing. Just a sec...");

%topic The topic title

%text The text to be displayed in the body of the dialog.

MoreString.cs

Most of these functions were originally contained in MoreString. They are all string related.

String::left(%string, %len)

Returns the leftmost %len characters from %string. For example

String::left("What is at the left?", 7)

will return the string

"What is"

String::right(%string, %len)

Returns the rightmost %len characters from %string. For example

String::right("What is at the right?", 7)

will return the string

" right?"

String::starts(%string, %search)

Returns true if the beginning of %string exactly (including case) matches %search, character for character, otherwise returns false. For example

String::starts("What is at the left?", "What")

will return true, but

String::starts("What is at the left?", "what")

will return false.

String::ends(%string, %search)

Returns true if the end of %string exactly (including case) matches %search, character for character, otherwise returns false. For example

String::starts("What is at the right?", "right?")
will return true, but
String::starts("What is at the right?", "right")
will return false (no "?").

String::insert(%string, %insert, %idx)

Returns a string where %insert has been inserted into %string and position %idx. For example
String::insert("Insert into this", "Tribes ", 7)
will return the string
"Insert Tribes into this"

String::replace(%string, %search, %replace)

Returns a string where all instances of %search in %string have been replaced with an instance of %replace. For example
String::replace("Insert into this", "Insert into", "Replace")
will return the string
"Replace this"

String::setString(%chars, %num)

Returns a string consisting of %num repetitions of %chars. For example
String::setString("Four", 4)
will return the string
"FourFourFourFour"

String::trim(%string)

Returns %string after stripping off any leading or trailing spaces. For example
String::trim(" too many spaces ")
will return the string
"too many spaces"

String::pad(%string, %char, %width, %justify)

Returns %string after padding it with enough %char's to give it a length of %width. If %char is more than one character in length, or if %width is less than the initial length of %string, then %string will be returned unmodified. Justify can be "left", "right", or "center" (actually only the first character of %justify is significant and case doesn't matter). For example

String::pad("XXX", "o", 10, "center")
will return the string
"oooXXXoooo"

The most common use will likely be one where %char is passed as " " (single space) to allow text to be lined up in columns.

String::indexOf(%string, %search, %idx)

Returns an index indicating the position of %search in %string, much like the Dynamix function String::findSubStr(). The difference here is that indexOf() begins searching %string after %idx characters, allowing you to skip over instances of %search already found earlier in %string. For example

String::indexOf("A string that repeats. String repeats.", "repeats", 24)
will return 30.

String::len(%string)

Returns the length of %string in chars. For example
String::len("A string with length.")
will return 21.

String::getWordCount(%string)

Returns the number of words in %string, using spaces as delimiters. For example
String::getWordCount("A string of words to count.")
will return 6. Note that the period at the end is considered a part of the word 'count' as there is no space between them.

String::ascii(%string, %idx)

Returns the ascii value of the %idx'th character of %string. For example:
String::ascii("ABC", 0)
will return 65. %idx is zero-based, so the first character is at %idx == 0. Only ascii numbers 32 through 126 are returned. If the character falls outside that range, if %idx < 0, or if %idx >= String::len(%string), a -1 will be returned.

String::char(%ascii)

Returns the character having the ascii value %ascii. For example:
String::char(65)
will return "A". Only ascii numbers 32 through 126 are returned. If the value falls outside that range a "" will be returned.

String::pixels(%string)

Returns the width of %string in pixels. Only ascii numbers 32 through 126 are considered. If the value falls outside that range its width will not be added.

SenseStack.cs

I'm hoping this script catches on because it would clear up some real problems. If all scripts that modified sensitivity use SenseStack, it makes sure that sensitivity is always returned to a reasonable value and that axis flip is preserved. The last script to request a certain sensitivity determines the current sensitivity. When that script releases sensitivity, the previous script's sensitivity is reinstated. When all scripts have released sensitivity, it will revert to the value set in the gui options. If scripts release out of order, there is no foul up where base sensitivity ends up being set to one of the script's values.

To turn on the script specific sensitivity for your script, call

```
SenseStack::request(%script, %sense);
```

passing in your script's name and the desired sensitivity. You may call this repeatedly with the same %script parameter and it will change the sensitivity accordingly. When your script no longer needs to modify the mouse sensitivity, call

```
SenseStack::release(%script)
```

passing the same script name used above and the sensitivity will revert to the setting requested by the script before you (or the most recent not released) or to the base sensitivity as set by the user (whichever is appropriate). Repeated calls to this function after the first call will have no effect.