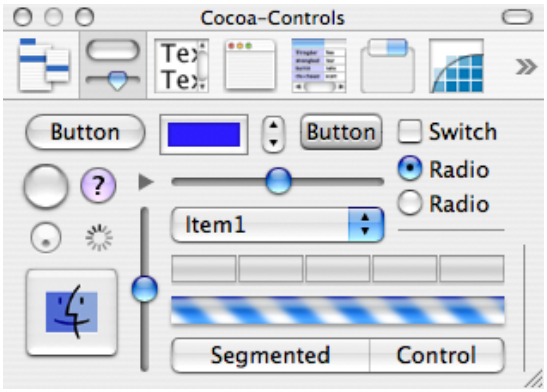
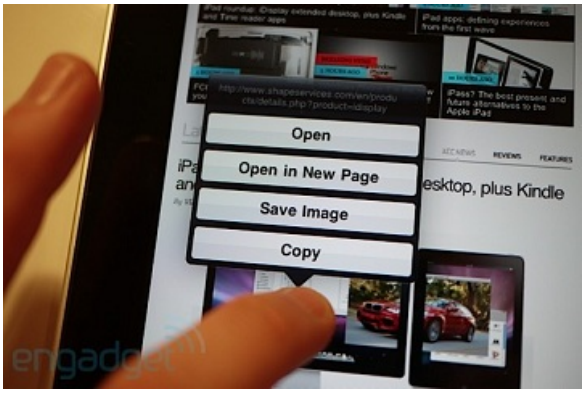
Assignment 3: MicroUI

CS 148 Autumn 2012-2013

Due Date: Monday, October 15th, 2012 at 7pm





Introduction The goal of this assignment is to learn techniques for synthesizing "raw" input events into more complex patterns of interaction and use these techniques to create a line editor. The GLUT programs we have built so far have used event callbacks to respond to user input. These user interactions have been extremely simple when compared to those found on PC operating systems, mobile phones, and game consoles.

In this project, we will be building up from the simple facilities provided by OpenGL and GLUT to make our own GUI (graphical user interface)"controls." While the UIs we will build are still relatively simple, these are the same techniques being used by major desktop OSes. Both the Windows and Mac UIs use graphics hardware interfaces like OpenGL and DirectX to draw themselves. We will touch on the three following UI topics for this assignment.

UI Widgets Most popular user interface toolkits include the concept of a widget, sometimes called a "control", "component", or "panel". A UI widget occupies a region of a screen or window, and has its own logic for display and event handling. Examples of widgets in common user interfaces are text labels, push buttons, sliders, scroll bars, and progress meters. For this project we will use a class `UIWidget` to represent UI widgets. We will define particular types of widgets like labels and buttons as subclasses of `UIWidget`.

Event Routing When we register a GLUT callback for mouse or keyboard events, the same callback responds to all events regardless of where they occurred in the window. In order to make a UI with widgets, we must make sure that each of these events gets routed to the widget or widgets that are appropriate for handling it. For example, when the user clicks a mouse button, we would usually like to send this event to the "closest" widget under the on-screen mouse cursor. In order to determine whether a widget was under the mouse cursor, we must have some way to perform hit testing between a point and a widget.

Editors When producing graphics for movies or games, we often need tools for editing shapes, images, animations and the other resources. While certain types of data can easily be edited with "off the shelf"

widgets, editing graphics objects often calls for the creation of custom widgets. After all, you wouldn't want to select a color in an image editor with text boxes.

One of the benefits of building your own UI toolkit in OpenGL is that you can always add new widgets when the need arises.

1 Part 1: Simple widgets

Step 1: Download and build the starter code

1. Download the assignment starter code. We have included all of the build files (Visual Studio project, Xcode project, and cross-platform Makefile).

2. Build the starter application. If you build and run the MicroUI application, it should start up and display a single blue rectangle on a background of Jake Sully from the movie Avatar (you'll understand why later in the assignment).

OS X users: If you did not do Assignment 2, you will need to install some required libraries with MacPorts. Please follow the instructions from Assignment 2 for this step. If you did these steps for Assignment 2 already, then you will not need to do them again.

Xcode users: See the troubleshooting section at the bottom of this page to set your working directory to the MicroUI directory.

Step 2: Review `UIWidget` and its Subclasses Much of the work for this assignment will be in defining subclasses of `UIWidget`. Your first step should therefore be to read the `UIWidget.h` header.

In order to display a widget on the screen, we will use the `UIWidget::Display()` function. This function is named after its relationship to the GLUT display event. You will notice that in the starter code we call `Display()` on all the widgets in the window. A subclass of `UIWidget` will need to override `Display()` to implement its own display logic.

Each widget in our system occupies a rectangle on the screen. This rectangle can be accessed with using the `UIWidget::GetRectangle()` function and set with the `UIWidget::SetRectangle()` function. Changing the rectangle of a widget should change how it displays itself, and how it responds to hit-test operations. The points in the rectangle are in OpenGL window coordinates: The origin is at the lower-left corner of the window and increases towards the upper-right corner. For our purposes, a rectangle can be best defined by its bottom-left corner (the point with minimum X and Y) and its top-right corner (the point with maximum X and Y). The `UIRectangle` class represents this design, with two `STPoint2` members, called `pMin` and `pMax` respectively. Please quickly read the `UIRectangle.h` header.

We can test whether a widget "covers" a point on the screen with the `UIWidget::HitTest()` function. This checks if the specified point is inside the rectangle of the widget by default. However, your widgets could implement special-case logic within this function; for example, to implement an oval-shaped button.

A widget can respond to mouse events by overriding the various `UIWidget::HandleMouse*()` functions. It will be your responsibility in this assignment to translate GLUT mouse events into calls on these functions, and to implement these functions for your new widgets as needed.

Step 3: Implement `UILabel` Widget The first widget we will implement is a label. This is a widget that simply displays a text string. In order to draw fonts (for your label widget, or as the label on a button) you will need to use the `STFont` class. Rather than give you detailed information here, we will instead ask you to read the `STFont.h` header included with `libST`.

Before implementing the `UILabel` class, you may want to update the `CreateWidgets()` function in `main.cpp` to create one or more labels for testing. You will then need to implement the `UILabel::Display()` function.

Every label stores both an `STFont*` and a `std::string`. You should investigate the functions declared in `STFont.h` to see how to use these for drawing.

Before moving on, make sure that your `UILabel` draws itself properly when given different rectangles in the window.

There is a pretty serious "gotcha" when using `STFont` – If the lower-left corner of your text string lands outside of the window, then the string won't draw *at all*. This is a side-effect of how `STFont` is implemented and not a bug in your code. Please make sure that when you test your widgets you place all your text so that it fits within the window.

Step 4: Routing Mouse Events The next widget we will implement is the `UIButton` but before we can get the button working we will need to route GLUT mouse events to the appropriate widget so our buttons can respond to clicks. The starter application does not have callbacks for the various kinds of GLUT mouse events so you will need to add these. When a mouse event occurs in the GLUT window you will need to make sure to call the appropriate `UIWidget::HandleMouse*()` on an appropriate widget in response.

Below are some GLUT functions you will need to call to set up the mouse callback functions. You should read about them before implementing buttons.

- `glutMouseFunc` sets up a callback for mouse clicking
- `glutPassiveMotionFunc` sets up a callback for mouse motion
- `glutMotionFunc` sets up a callback for dragging motion (mouse movement during mouse click and hold)

Here are some hints:

- It should be easy to find which widget is under the mouse pointer using `UIWidget::HitTest()`
- Remember, though, that if widgets overlap, the last one to be drawn will appear "on top" of the other.
- To generate mouse enter/leave events, you might need to remember what widget was under the mouse at the last event.
- *The OpenGL window coordinate space puts the origin in the lower left, but GLUT mouse events are reported as if the origin is in the upper left.*

This last point is an important one to deal with early in your event callbacks. Many GLUT callbacks receive the (X, Y) position of the mouse, and that position is measured in pixels, with $(0, 0)$ at the top-left of the window, and Y increasing as you move down. This "Y down" convention is typical of many GUI systems (because it matches the flow of text layout in most western scripts), but is not consistent with the OpenGL coordinate system.

The starter application sets up an OpenGL projection aligned with the pixels of the window such that $(0, 0)$ is in the lower-left of the window, and Y increases up. To translate the coordinates (X, Y) from the GLUT mouse event into "OpenGL coordinates" you will thus need to compute $(X, H - Y)$ where H is the height of your window.

There are some more complexities to mouse event routing that we will get to shortly, but you should at least be able to implement the basics with these hints.

Step 5: Implement `UIButton` Widget We are now ready to add in the `UIButton` widget. You will probably want to update the starter application to create some buttons before you start implementing their behavior.

Your `UIButton` class will need to display itself with both an outline or background and a text label. To make for a nicer image, you should draw the text for the button centered within the button's rectangle.

Your button class will need one or more member variables to hold its state. The state should change in response to mouse events, and the way that the button is drawn should vary based on the state. Every state

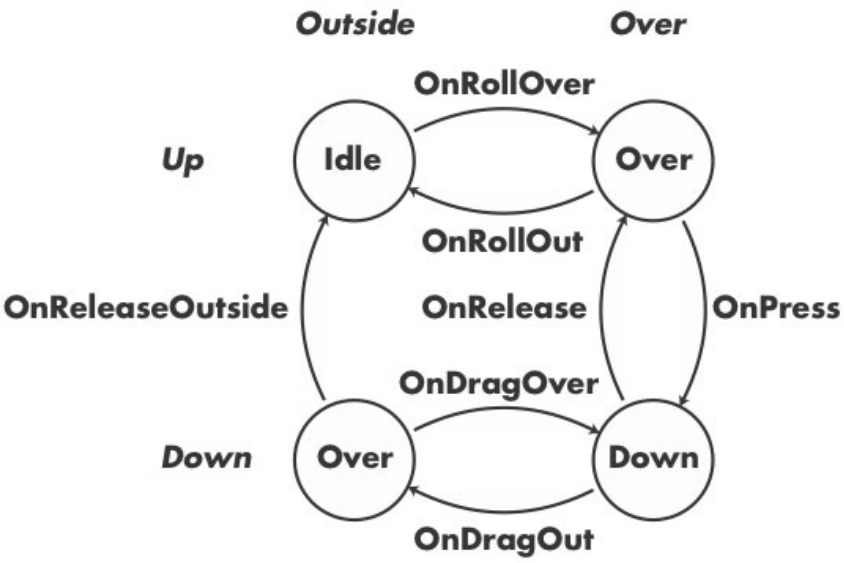


Figure 1: State transitions for the Flash "push button" widget

should lead to a different appearance for the button. Certain state transitions should cause the button to "fire" and call the callback function pointer stored in the button.

In lectures, we covered two different models for how buttons behave in Adobe Flash: push buttons and menu buttons. These two types of buttons share the same states, but have different state transitions. For this assignment, you will only need to implement push buttons.

Make sure to read the next two sections, **push button** and **mouse capture** before you jump into implementing buttons.

Push Button: The state-transition for the Flash "push button" widget was shown in lecture and is replicated here in Fig. 1. The important thing to note about push button behavior is that the button should "fire" only under a particular sequence of events. The exact sequence of events is:

- A mouse-enter event
- A mouse-down event
- Zero or more pairs of mouse-leave/mouse-enter events
- A mouse-up event

Note in particular that the button does *not* fire if the final mouse-up event occurs outside of the area of the button, and it should also not fire if the mouse-down event occurs outside the button, followed by a mouse-enter and mouse-up event. You will likely see this behavior in the push buttons (e.g. "OK" and "Cancel" in a dialog box) in your preferred desktop OS.

We have provided a link to a Flash animation so that you can use to try out the push-button behavior. Note the exact conditions under which this button does and does not generate an `SWFBUTTON_MOUSEUP` event.

You can see on the state diagram that a push-button has four states: "Idle," "Down," and two states labelled "Over." This diagram is labeled with two "Over" because both states are drawn the same in flash. *Make sure you draw "Idle," "Down," and "Over" differently.* Look at the Flash animation for an example of how to draw the states. A button will be drawn the same if the mouse is over the button, or if the user clicked inside the button but moved outside with the mouse still down. You could display the different states by using different colors, textures, animations, or whatever you desire.

Mouse Capture: You may find that your initial mouse event routing is insufficient for implementing these buttons. For example, the mouse-up event that would change a push button from "Over" (mouse down outside of the button) to "Idle" occurs outside of its rectangle. If you apply a simple strategy of sending all mouse events to the widget that the mouse is directly over, then your button would miss this event and be left in the "Over" state incorrectly.

Most UI toolkits resolve this using the idea of mouse "capture." In its event handler for a mouse-down event, a widget can signal that it would like to intercept all mouse events until it subsequently "releases" the mouse.

You will need to implement some mechanism like this, whereby a widget can temporarily intercept all mouse events - even those that occur outside of its area - for a limited time.

A simple approach to this would be to define two functions in your main.cpp called `CaptureMouse(UIWidget*)` and `ReleaseMouse(UIWidget*)` and call these from within your push button implementation. You will then need to update your GLUT mouse event callbacks in `main.cpp` so that if any widget has "captured" the mouse it will intercept all the mouse events.

Step 6: Implementing a Dragging Behavior In implementing the push-button behavior, we have seen that some interaction techniques require us to "intercept" events that would normally be routed elsewhere. We will now explore another interaction technique that involves intercepting and re-routing events: dragging.

In the Flash UI, a widget can initiate dragging behavior by calling `beginDrag()` in its mouse-down event handler, and `endDrag()` in its mouse-up handler. The call to `beginDrag()` registers a specific widget (or a "movie clip" in Flash terminology) to be the "drag object." Whenever the mouse moves, the object will also be moved, until `endDrag()` is called.

Implementing dragging should be a lot like implementing mouse capture. You might need functions like `BeginDrag(UIWidget*)` and `EndDrag(UIWidget*)` to set or unset the "drag object", and you will need to update your GLUT mouse callbacks to handle the case where a drag object has been set and properly move it around the screen.

To test that your dragging works, you may want to edit a subclass of the `UIBox` widget we have provided so that users can drag boxes around the window.

2 Part 2: Line Editor UI

At this point you should have many useful widgets and interaction techniques to play with. Now you will use these tools, and create some new ones, to implement a simple editing interface. Your line editor will allow users to draw and edit lines.

The next assignment,"Morphing", will be a class project where you will write the software to compute a morph from a picture of you to a picture of one of your classmates. By combining all the morphs, we will be able to make a video of everyone in the class morphing into each other!

In order to compute a morph between two people, correspondences between facial features must be established manually by the graphics technician or artist, ideally using a well-crafted user interface. These correspondences are represented by directed line pairs, and your MicroUI application can be used to help you create them for the morphing algorithm you will code next week. An example of two images with correspondence lines indicated (in white) is shown in Fig. 2. A good line editor for this task would have the following properties:

1. The ability to reposition lines after drawing them
2. The ability to delete lines after drawing them
3. Lines should be directed (i.e. line endpoints should be visually distinguishable)
4. Each line should be labeled or colored, so that...
5. ...it is easy to see which lines match between the two different image
6. Adding a line to one image automatically adds a line to the other, so that the number of lines is always the same between the two images

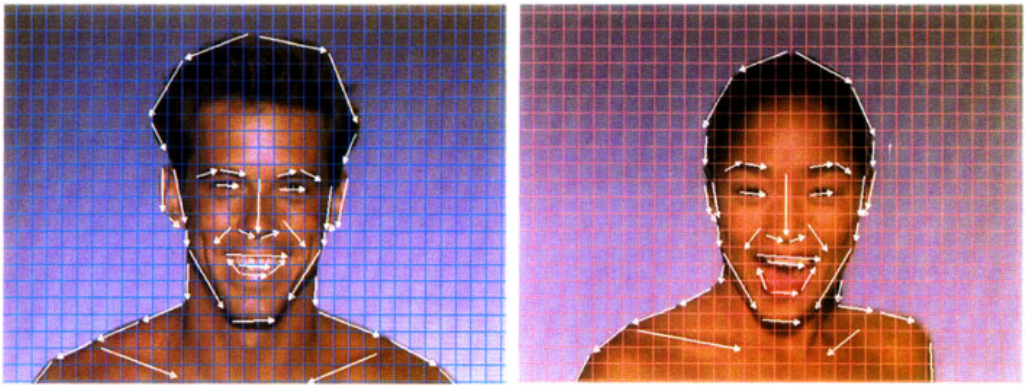


Figure 2: Two images with correspondence lines shown in white

7. Users should be able to save lines they have drawn to a file, which can be loaded later (this is mostly implemented for you)

Your line editor will only require properties 1, 2, 3, and 4, and will only require editing a single image.

Step 7: Add a Quit Button and Label Your Line Editor Include a `UILabel` widget that gives the name of your application and add a "quit" push button to your line editor that terminates the application

Step 8: Allow User to Add Lines The user should be able to add lines by left-clicking on any position in the image, dragging to another position, and releasing the mouse. This draws a line from the position where the user clicked to the position where the mouse was released. You should fulfill the following requirements for drawing lines.

- After a user has clicked on the image to begin drawing a line, but before releasing the mouse, there should be some visual indication that a line is about to be added between the point clicked and the current mouse pointer position, for example, by drawing a line between the two points using `GL_LINES`.
- Line endpoints should be distinguishable from one another. One way to do this would be to color each endpoint differently. Another way would be to label each endpoint differently, for example, an arrow could be drawn at one end.
- Line endpoints must be draggable so the user can edit lines after drawing them. One way to do this would be to make the line endpoints draggable widgets like those you created in part 6.
- Lines should be distinguishable from one another. One way to satisfy this is to give each line its own label. Another way would be to color different lines differently, in which case it would be OK to just loop through at least three colors (e.g. first line drawn is red, second green, third blue, etc.). It is not absolutely necessary that each line is distinguishable from every other line, but ideally lines should look different from neighboring lines. The reason for this requirement is if you were editing two images, this would allow you to tell which lines correspond between the two images by their label or color.

Step 9: Deleting Lines You must come up with some way to delete lines. At the very minimum, you could include a button which simply deletes the most recently drawn line. Better ways to delete lines would allow users to delete any line they choose. For example, you could allow the user to click on a line to select it and then click on a button to delete the selected line. Another option would be to right click on a line to bring up a menu of different options, one of which would be 'delete'. Or, come up with your own way!

Step 10: Submit Your Picture for the "Grand Metamorphosis" You'll be morphing yourself into another student in the next assignment, so you will be needing a picture of the "target" student. Thus, you

can really help the previous student out by providing a picture of yourself! To make the morphing effect more convincing, the photos should be as similar as possible. Here are some suggestions:

1. Take a picture of your face in front of a plain white background.
2. Crop and/or resize the photo to 512×512 pixels. The photo should be tightly cropped on your face, so that your face takes up most of the image.
3. Upload the picture to the WWW directory in your AFS home directory.

If, for whatever reason (privacy, etc.), you are really opposed to posting a photo of yourself, you may instead upload a picture of an avatar to represent you. Of course, make sure your avatar has distinct facial features so that the morph can be completed – humans/humanoids are preferred. Try your best to find or create a picture that meets the suggestions above (plain background, same size).

Questions

1. Suppose you have a list of `UIWidgets` denoted by `(W0, W1, W2, ... ,WN)`. When drawing, you move through the widgets from "low to high": `for(i = 0 ... N-1) W[i]->Display()`. This means that widgets later in the list will overlap those earlier in the list visually. Write pseudo-code for a routine `FindWidget(P)` which finds the widget the user clicked on for a mouse event at point P. You can freely use functions defined in `UIWidget` in your answer. Make sure that when widgets overlap the one you return is consistent with what the user would see on the screen!
2. Suppose you have an `STFont*` called `F` and you want to know the width and height (W, H) of the smallest rectangle that can contain the string "Hello, World!". How would you compute W and H using the functions provided by `STFont`?
3. We used the push button as an example of when you might want to have a widget "capture" future mouse events for a period of time. Imagine instead we only gave the push button mouse events that occurred inside its rectangle. Describe the exact sequence of mouse events you could use to "fool" the button into firing when it had not been properly pushed.

Extensions: If you want to create a line editor that can be used for the next assignment, you'll want to look at properties 5-7 listed at the beginning of Part 2.

- Edit two different images: In the file `config.txt`, you will see two image filenames. These images have already been loaded as `gBgIm1` and `gBgIm2` global variables by the call to `ParseConfig`. For your line editor, users must be able to edit lines for both image files. To support this, you may either add a widget which allows users to toggle which image is being displayed, or you may display both images side by side. Drawing a line on one image draws the corresponding line on the other image. These lines should have the same relative position in both images. If images are shown side-by-side, this means that two lines will be drawn and displayed at the same time, one in each image, with the same coordinates relative to the lower left corner of the corresponding image. If you choose to toggle between two images, this means that you will add two lines at the exact same window coordinates, but one of the lines will be hidden depending on which image is being edited (it should also not receive mouse events). Corresponding lines should have the same label or color. You may want to move your line drawing code so that it uses the `AddNewLine` function, to prepare for the next step of loading lines from a file.
- Loading a line editor file: Add a way for the user to load line sets from a file. To do this, you won't actually have to do any parsing of files, as we already do this for you. For loading, you should call `loadLineEditorFile`, which is provided in `parseConfig.h`. The file we are loading from is specified in `config.txt`, and the filename specified there is already read into the variable `gLoadFName`. `loadLineEditorFile` will parse this file and make calls to `AddNewLine`, so if you have implemented `AddNewLine` correctly, setting up this callback should automatically load the lines correctly. `loadLineEditorFile` will load the lines in a different order than when you manually add lines. Instead of adding identical copies to each image, it will add all the lines for one image, then all the lines for the other image. You'll want to make sure that your `AddNewLines` function works correctly when the

`imageChoice` parameter is something other than `BOTH_IMAGES`, and you'll also want to check that labels and colors for corresponding lines still correspond after loading lines from a file. The default load file is `lineEditorFile.txt`, which gives correspondences between Jake Sully in human and avatar form. This should give you an idea of how this line editor will be used for the next assignment.

- Saving a line editor file: Add a way to allow users to save the currently displayed lines to a file. To do this, you should make a call to `saveLineEditorFile`. Read the comments about `saveLineEditorFile` in `parseConfig.h` so you understand the parameters it expects. The filename of the file to be saved to is specified in `config.txt`, and is already loaded as the variable `gSaveFName`. Edit `config.txt` if you would like to change the names of the files to be loaded and saved. You should definitely check that the file you saved can be loaded correctly.

Grading: Grading will be based on a four-point scale: zero points for no credit, four points for full credit. To receive full credit, you must answer all the questions correctly and implement a line editor that meets all the requirements stated in the steps above. You may freely implement additional new UI elements that improve the editing experience, but we ask that you also include the required widgets in your demo. Credit will be given in the following fashion:

- Correct answers to the questions and submitted picture, plus correct `UILabel` implementation. **(1 point)**
- A correct implementation of `UIButton`. This includes correct drawing and correct state transitions. The three button states ("idle," "over," and "down") must display differently.**(1 point)**
- A correct implementation dragging behavior and editing lines by dragging.**(1 point)**
- Drawing and deleting lines.**(1 point)** Your project must distinguish between lines and line endpoints.

Tips/Troubleshooting

- Transitions that are not drawn in the state machine diagram are, by convention, assumed to be loops. In other words, they do not alter the state.
- For this assignment, it is permissible to have text just outside the rectangle in `UILabel`. (In practice, many label implementations only display portions of the text that fit in the bounding box, or a layout manager will dynamically adjust the bounding box to fit the text.)
- The `FireCallback` type is defined earlier in `UIButton.h` as a C function-pointer type. When creating a button you should pass the address of a C function, like so:

```
void FunctionToCallWhenButtonFires(UIButton* whichButton){
    printf("The button fired!\n");
}
int main(int argc, char **argv){
    //...
    UIButton* button = new UIButton(gFont, "Press Me!", &FunctionToCallWhenButtonFires);
    //...
}
```

- The starter code assumes that the resource files (input text files) are put in the project directory.
- Xcode 4.x: Please set the working directory to Custom Directory. Product → Edit Scheme... → Run MicroUI → Options. Set the Working Directory to the one containing your Xcode project on your computer (See Fig. 3).
- Xcode 3.x: Please set the working directory to Project Directory. To do this in Xcode: Project → Edit Active Executable "MicroUI" → General → Set the working directory to → Choose Project Directory (See Fig. 4).

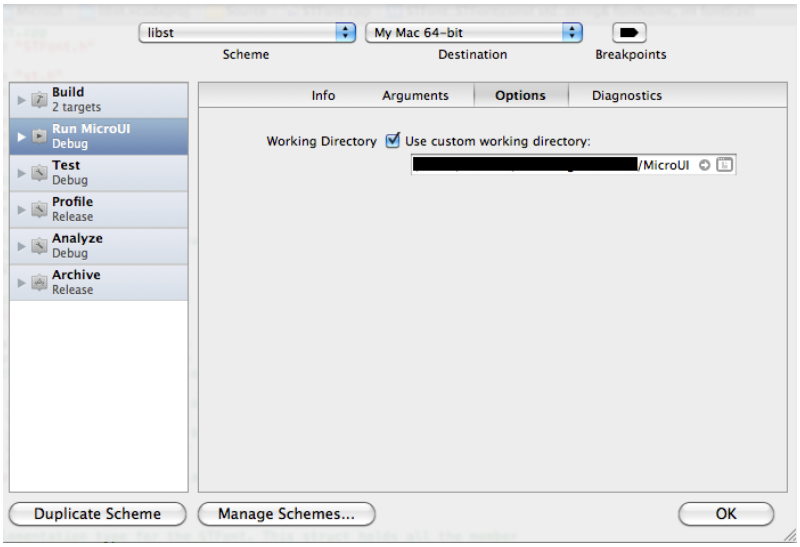


Figure 3: Xcode 4.x: Use custom working directory

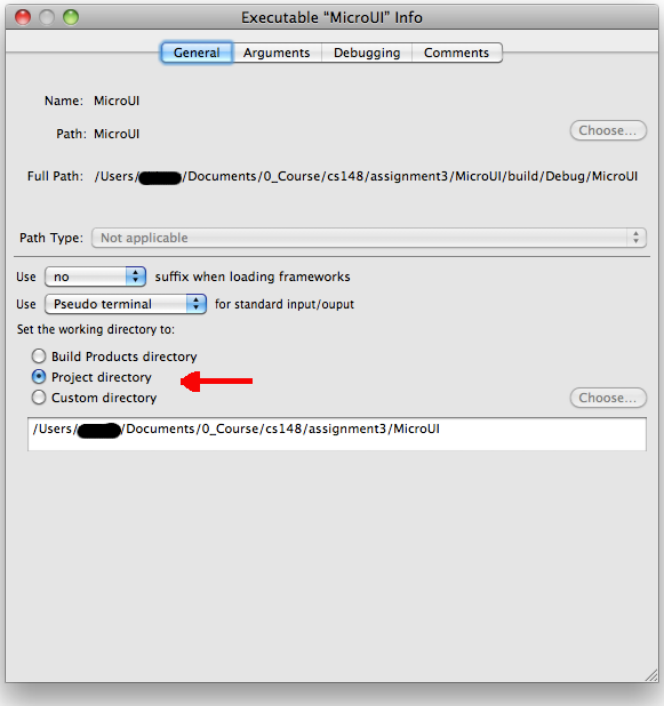


Figure 4: Xcode 3.x: Set working directory to Project directory