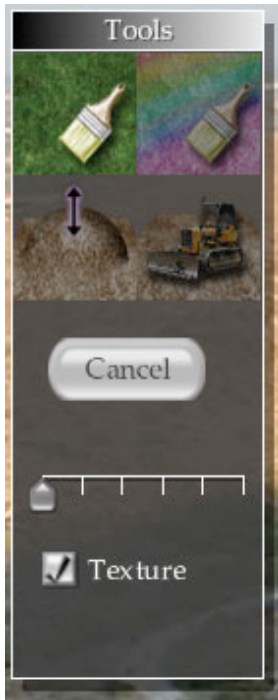


HOG: The Handy OpenGL GUI Toolkit

Project Description	2
Project Plan	2
<i>Main Tasks</i>	2
<i>Schedule</i>	3
<i>Risk Management</i>	4
Software Requirements	5
<i>Functional Requirements</i>	5
<i>Non-Functional Requirements</i>	5
Software Design	7
Software Testing	10
<i>Primary Objective</i>	10
<i>Schedule</i>	10
<i>Fuzz Testing Framework</i>	10
Lessons Learned	12
<i>Analysis of Task Times</i>	13
Documentation	14

The complete source code is over 3k lines, so instead of printing it I have included it on an Appendix CD.

Project Description



HOG: The Handy OpenGL GUI toolkit is a cross-platform C++ library for displaying and receiving input from user-interface widgets like scroll-bars, sliders, text fields, and more in an OpenGL window. It is designed primarily with games and game editors in mind, and is an extension to PIGlib, the Portable Independent Game library I wrote for Operating Systems class that serves as a cross-platform base for game development.

HOG supports Mac OS X and Windows. Developers can write their user interface (the locations of widgets, what functions are called when widgets are manipulated, etc.) by changing a simple text file interpreted by HOG when the application starts. Every graphics call that HOG makes to display the widgets is abstracted to a GUI renderer class for easy customizability of the appearance of the GUI. HOG comes with complete documentation and sample source code demonstrating its capabilities.

A small HOG window containing Tool Palette, Button, Slider, and Check Box widgets.

Project Plan

Main Tasks

This project is fairly large, and has required many hours of programming. Unlike a corporate project, I have not had eight hours a day to devote to the project. My free-time varies widely day to day and week to week. So rather than plan around a certain amount of development to occur each week, I have created a series of milestones with firm dates. A milestone is passed with SUCCESS if every feature in the description is completely implemented by the milestone date.

September 20: Basic GUI manager and GUI renderer classes created. Ability to create and drag blank virtual windows. SUCCESS.

October 1: Widget base class created. Button widget implemented, but must be specified manually in code rather than in interpreted text file. SUCCESS.

October 10: Tool palette, slider, and vertical scroll bar widgets implemented. Again, must be specified manually in code. SUCCESS.

October 25: Text field widget completed, with copy, paste, selections, and all other standard text field behaviors implemented. Color well widget implemented. SUCCESS.

NOTE: Before November 10th, I implemented two additional widgets that were not part of the original requirements: CheckBox and RadioButtonGroup. These widgets are a result of feature-creep and are partially responsible for the next two missed milestones. See the 'Lessons Learned' section.

November 10: Ability to parse text file and generate widgets from text file description. Ability to call developer-defined functions based on names in text file. FAILURE: Passed on November 15.

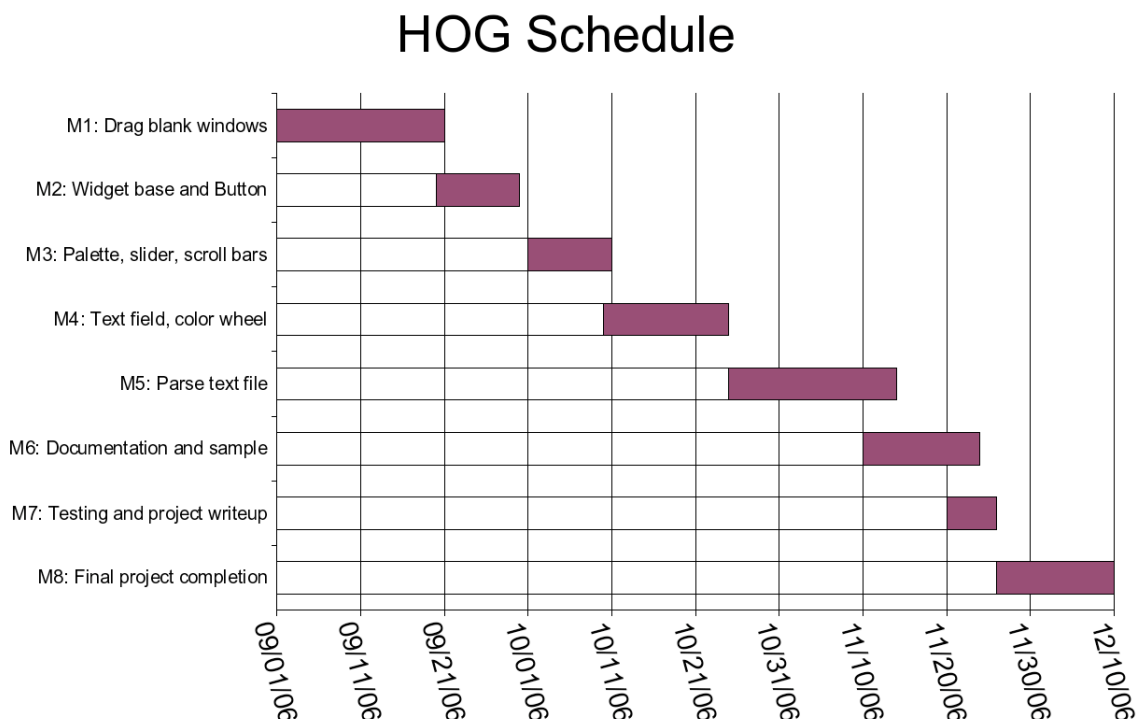
November 20: Documentation written and example project demonstrating HOG's capabilities created. FAILURE: Passed on Nov. 24.

November 26: Testing completed and bugs fixed. Draft of project write up completed. SUCCESS.

December 10: Final peanut project write up completed.

Schedule

On the left are each of the milestones, M1-M8. The chart demonstrates how I was on schedule, even ahead of schedule, for the first 5 milestones. Notice that M5 and M6 have significant overlap, as I was working hard on both to get them both completed before the next milestone was due. Similarly M6 and M7 overlap considerably. By M8 I was finally caught up with the plan.



Risk Management

Risk	Probability	Effect	Occurred
Don't make October 25th milestone	Low	Tolerable	No
Don't make November 10th milestone	Medium	Serious	Yes
Don't make November 20th milestone	Medium	Serious	Yes
Don't make November 26th milestone	Low	Catastrophic	No
Don't make December 10th milestone	Low	Serious	?
Testing doesn't reveal show-stopping bug until close to November 26th milestone	Low	Serious	No
Testing is inconclusive, bugs present in final version	Medium	Tolerable	No
Can't get fuzz-testing to work	Low	Serious	No
Can't get HOG working on Windows	Low	Catastrophic	No
Widget requirements leave out a useful widget	Medium	Tolerable	Yes
HOG runs slowly on old computers	Medium	Tolerable	No

Luckily only a few of the risks wound up happening, and none of them were catastrophic. I missed the November 10th and November 20th milestones, which made for a long Thanksgiving weekend of catching up. Notice the “Widget requirements leave out a useful widget” risk. Unfortunately in my initial draft I left out two very common and useful widgets, namely check boxes and radio buttons. Before the November 10th milestone I decided to implement these two widgets. Since it wasn’t part of the original plan, I wound up missing the next two deadlines because of it. A lesson learned: make sure your requirements don’t conflict! In this case I had a functional requirement that left out two widgets necessary to fulfill a non-functional requirement.

Software Requirements

Functional Requirements

Requirement: HOG will allow developers to create, display, and interact with, at the least, the following widgets:

Windows, Tool Palettes, Buttons, Vertical Scrollbars, Text Fields, Sliders, and Color Wells.

Discussion: All of the required widgets and their associated behaviors were successfully implemented, in addition to two extra widgets: Check boxes and radio buttons. SUCCESS.

Requirement: HOG will parse a simple GUI description language that will allow developers to specify widget properties in a text file. The description language must, at a minimum, allow developers to specify the following properties of widgets: Position, dimension, initial value, and function name to be called upon user interaction with the widget.

Discussion: HOG successfully parses text files and all of the listed properties, with the exception of dimension, are available for each widget. For some widgets, such as the Button widget, it doesn't make sense to allow the developer to set the dimensions because the dimensions depend on how big the Button's title is. This requirement should have omitted the 'dimension' property. SUCCESS.

Requirement: HOG will run on both Macintosh and Windows with no code changes.

Discussion: HOG now compiles and runs on either platform with the same code, however the Windows version has not been tested as thoroughly. SUCCESS.

Requirement: HOG must come with at least one example widget renderer.

Discussion: HOG comes with LineLooks, a simple widget rendering class with a clean, futuristic look. SUCCESS.

Non-Functional Requirements

Requirement: HOG must be reliable. Specifically, an application with two of each kind of widget must be able to survive fuzz-testing for at least two hours without crashing.

Discussion: HOG not only survived two hours of fuzz-testing, but sixteen. SUCCESS.

Requirement: HOG must have complete documentation, describing the API in detail. As part of this documentation, HOG must come with source code for an example program, demonstrating how to create and interact with the widgets.

Discussion: The documentation has a short tutorial on how to use HOG and a basic description of each widget with their public access functions. The documentation is not as complete as I would like, but I believe it fulfills the vague requirement that the documentation describe the API 'in detail'. Additionally, HOG comes with a simple example program showing how to load and interact with widgets. SUCCESS.

Requirement: All GUI activities within reason should be supported in some way, i.e. given some sort of input the developer wants to collect there should be at least one way to accomplish it with the default widgets.

Discussion: Unfortunately, I forgot about checkboxes and radio buttons until near the end of development. These widgets are required, because before I added them there was no good way to get similar input from the user. So I wound up adding them near the end(see Risk Management), and it cost me two milestones. Luckily now this requirement is complete, because all types of input can be gathered, even if it requires a little ingenuity. SUCCESS.

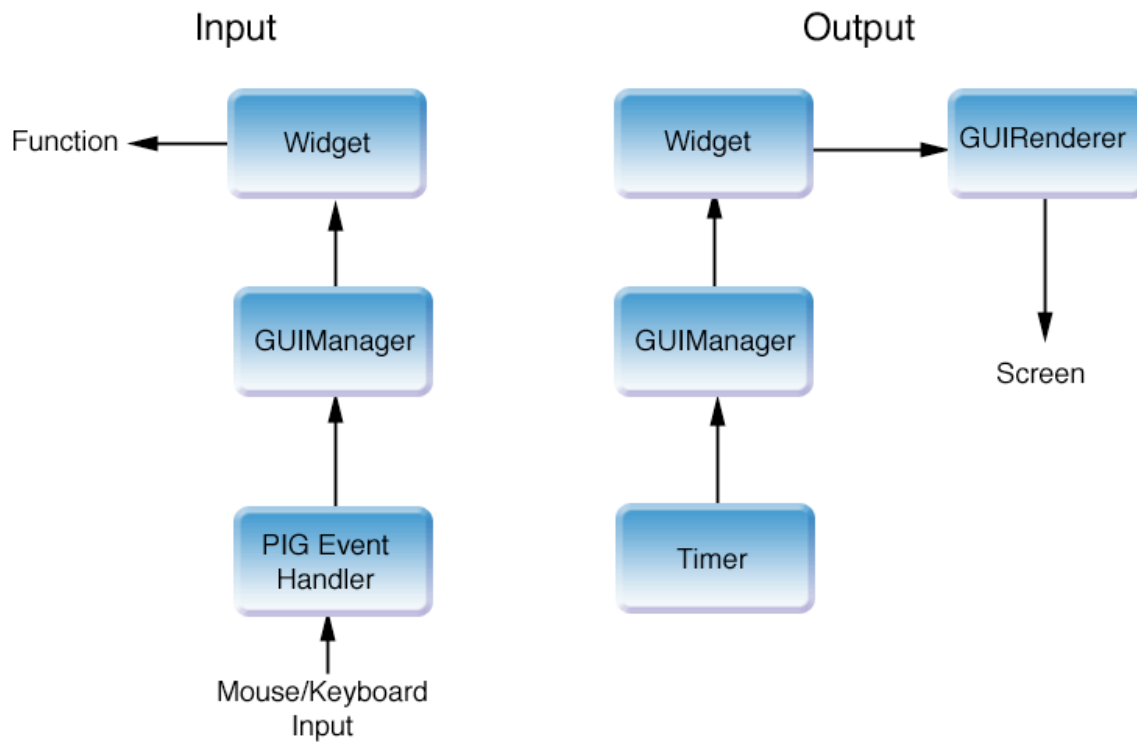
Requirement: HOG will be easily extendable in the following ways:

- 1) All rendering will be done via an abstract base class that can be implemented in whatever manner the user desires.
- 2) New widgets can be added by changing at most two lines of existing code.

Discussion: Rendering is handled by the abstract base class GUIRenderer which is easily subclassable. To add a new widget to the system, you need only add a line to the "makeWidget" function in "hog/Controls.cpp". SUCCESS.

Software Design

Any GUI library must focus on input and output. If the developer didn't care about input and output, they'd be using a text interface. I started with an abstract model of how ideally input and output would flow in HOG.



Notice that the input and output are largely unrelated. The output draws widgets to the screen tens or hundreds of times a second, and the input changes the properties of a widget and calls a user-defined function. For instance, if the user clicks a check box, its state is changed to 'on' and the program is notified through the calling of a developer-defined function. The new state is not reflected on-screen until the next time the check box is drawn.

There are three fundamental classes in HOG:

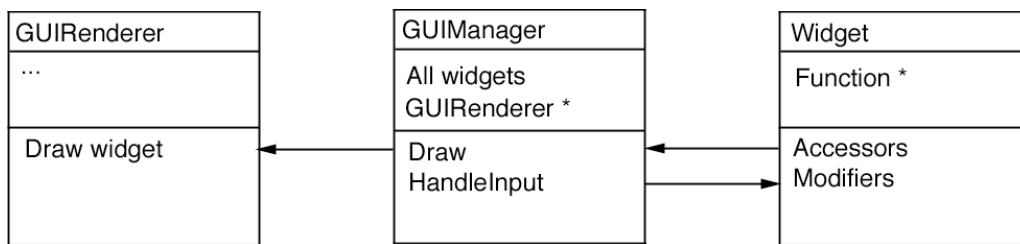
GUIManager - The "owner" of all other HOG-related objects. Maintains a list of pointers to widgets, has a "draw" function that draws all of the widgets, and redirects input to the correct widget.

Widget - An abstract base class with virtual methods for handling input and drawing. All widgets, from text fields to sliders to radio buttons subclass from the *Widget* class.

GUIRenderer - An abstract base class with virtual methods for drawing widgets and determining text size. The idea is that the developer subclasses from *GUIRenderer* and writes all of their own code for drawing widgets, thus allowing them to customize the look of the GUI however they want. HOG comes with a sample *GUIRenderer* called "LineLooks".

I had two designs in mind for how these three fundamental classes would interact.

Design #1: Widgets manage themselves(mostly)

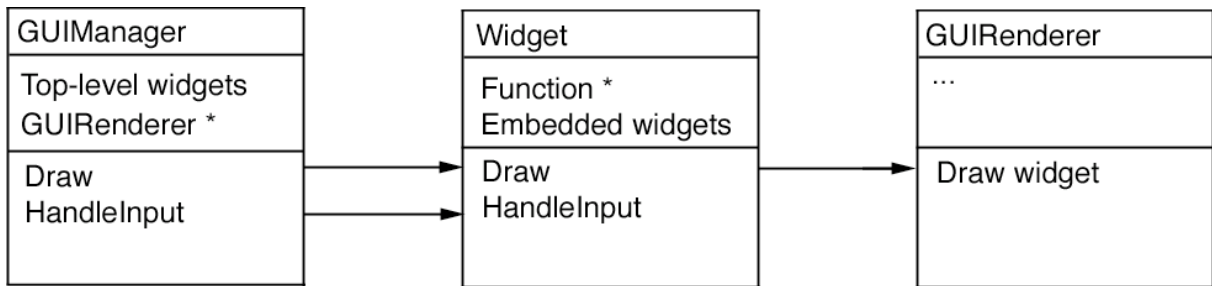


One way to design the classes would be to have *GUIManager* hold pointers to the top level of widgets, for instance, the on-screen windows. The windows themselves would then hold pointers to widgets embedded inside them, which could then hold other widgets, and so on. When *GUIManager::draw()* is called, the *GUIManager* would call each of the widget's draw commands, which would in turn call *GUIRenderer's* draw widget command on themselves and call draw on their embedded widgets. Similarly, input would be passed along from the *GUIManager* to each widget, allowing the widget to decide what to do with the input.

Advantages: New widgets can be added without modifying *GUIManager*. Fewer accessor and modifier functions must be written per widget, because the widget handles its own modification. Since widgets can be embedded, detecting where an event should be passed becomes a simple hierarchical process.

Disadvantages: It is easy to implement non-standard behavior in a widget, for instance, having the widget draw itself instead of letting *GUIRenderer* draw it. Ownership of widget pointers is more complicated, since widgets may own widgets. Care must be taken that every widget is disposed of upon destruction.

Design #2: GUIManager controls everything



A second way to design the classes would be to let GUIManager rule over the other classes absolutely. GUIManager would maintain a list of all widgets in the program, and would automatically call GUIRenderer's draw widget function on each widget. Similarly, given input, it would dictate how a widget should be modified.

Advantages: Centralized design may decrease bugs. Ensures standard behavior on every widget, even new ones. Having all widgets stored in GUIManager reduces allocation/deallocation headaches.

Disadvantages: Writing accessors and modifiers for every new widget is inconvenient. GUIManager would have to be modified slightly to add radically different new widgets. Detecting where to send an event may require a sophisticated algorithm.

Decision: I decided to implement Design #1. It was slightly less clean than Design #2, but it fulfills requirements better. Specifically, the requirement about easy customization with no less than two lines changed would be difficult to accomplish with Design #2. Adding new widgets would be a pain, because the developer would have to carefully add accessors and modifiers to their new widgets. The hierarchal nature of having widgets store embedded widgets was appealing to my aesthetic sense. Hence HOG has been written using Design #1.

The widgets are drawn to the screen whenever "GUIManager::draw()" is called, which is typically drawn every frame or hooked up to a timer. Whenever a mouse or keyboard event occurs, it is collected by PIG and the program's PIG event handler is notified. It is the event handler's job to notify the GUIManager, which in turn determines which widget should receive the event. This widget then either processes the event, or, if it contains embedded widgets, passes the event on to them. The widget can then call a developer defined function.

Software Testing

Primary Objective

My primary goal was to ensure that once a developer has HOG up and running successfully, HOG does not crash the program. There is nothing more annoying than writing clean, bug-free code, and then using somebody else's library and have it crash constantly. Hence I wanted to run HOG through extensive fuzz testing.

Schedule

Testing occurred between November 23rd and November 24th. I had missed the previous two milestones(see the planning section), and so I had less time to complete testing than I would have liked. I decided to focus on fuzz testing to guarantee the primary objective, but also tested HOG's text parsing abilities to ensure malicious users couldn't crash a program by giving it bogus widgets.

Fuzz Testing Framework

Since PIG lets you create mouse and keyboard "handler" objects, which have member functions like "mouseDown(float x, float y)", it was easy to just randomly flood these functions with fake mouse clicks and keyboard taps. I tried to make the input realistic, but spastic, incorporating mouse drags and key combinations like Control-C. However, the fuzz tester is not allowed to press the ESC key, since that closes the test program. The complete code for the fuzz testing framework can be found in the source code under "hog/FuzzTester.h" and "hog/FuzzTester.cpp".

NOTE: All tests were performed on a MacBook Pro 2.16 Ghz with 1 gig of RAM and an ATI Radeon 1600XT graphics card. I had hoped to test the Windows version as well, but ran out of time. Since most of the test cases are behavioral and the code is the same on both platforms, passing on the Mac likely means passing on Windows. However the fuzz testing of HOG should ultimately be performed under both Mac and Windows environments.

Test Requirement: HOG, once properly configured, does not crash the program.

Test Method: Fuzz testing

Known Input: All possible mouse clicks, mouse drags, scroll-wheel input, key presses, and key combinations.

Expected Output: Proper behavior for given input. Possible crashes that must be fixed. But the program must run for at least 12 hours without crashing to pass the test.

Results:

1: The first fuzz testing run ran for 12 seconds before crashing. It turned out there was a bug with the 'RadioButtonWidget' that caused the widget to use a non-existent texture if two radio buttons were selected in a short period of time. The bug was fixed.

2: The next run ran for 4 seconds before crashing. The 'TextField' widget was accepting input for characters it could not print. 'TextField' was changed to only accept valid characters it is capable of drawing.

3: The next run ran for 16 hours, and was closed manually. Considering the intensive nature of test, with 10-50 mouse clicks per seconds alone, I consider this run to be a success. It demonstrates that HOG will not likely not crash the program no matter what the user does.

Test Requirement: When parsing text files, HOG either generates the correct widgets or reports an error.

Test Method: Structural Testing in 3 cases

Case #1: Trying to declare a non-existent widget type.

Known Input: Writing 'RadioNonExistent' or 'blahblahblah' instead of 'RadioButtonGroup'.

Expected Output: HOG should report an error.

Results:

I tried substituting 3 widget names for other, random names. HOG reported the error: "Non-existent widget declared." Due to the way the code is structured, even if HOG handles one non-existent widget correctly it must pass the test.

Case #2: Random text.

Known Input: Garbage text containing unicode characters.

Expected Output: HOG should report an error and not generate any widgets.

Results:

I wrote a program a few months ago to randomly generate garbage unicode text. I gave HOG three different garbage text files and each time HOG reported the error: "Parsing error." Success.

Case #3: Adding a non-existent parameter.

Known Input: A parameter that is not recognized by the widget type.

Expected Output: HOG should ignore the parameter.

Results:

Added bogus parameters to many widgets, and HOG ignored all of them. Success.

Lessons Learned

Problem: One of the non-functional requirements dictated that all reasonable methods of input should be available to the developer, but I left two very important widgets out of the functional requirements. The check box and radio button widgets were added very late in the development process, and I wound up missing two milestones because of them. By adding these two widgets I completed the non-functional requirement, but I wound up having to work very hard to get everything done in time.

Cost: Two milestones.

Lesson Learned: Check your requirements carefully for conflicts. I should have looked over my requirements carefully for dependencies. In this case a non-functional requirement made certain demands on a functional requirement. Had I scheduled those two widgets from the beginning, it would have been much easier to include them. Alternately, I could have eliminated that non-functional requirement and left those two widgets out.

Problem: I underestimated the difficulty of testing the program and writing the documentation. As a programmer who has rarely had to thoroughly test and document my work, I wasn't expecting these two parts to take as long as they did. As such, the documentation is a little more sparse than I was hoping.

Cost: Weak documentation.

Lesson Learned: Testing and documentation can take a very long time, and should definitely not be treated as an afterthought. I think devoting an extra week to these activities would have given me enough time to complete them successfully.

Analysis of Task Times

I had a specific set of time in which I was going to complete each milestone. Some took less time than was scheduled, some took more.

Milestone	Scheduled	Completed	Error
1: Drag blank windows.	20 days	19 days	-5%
2: Widget base and button.	10 days	8 days	-20%
3: Palette, slider, scroll bars.	10 days	10 days	
4: Text field, color wheel.	15 days	15 days	
5: Parse text file.	15 days	20 days	+33%
6: Documentation and sample project.	10 days	14 days	+40%
7: Testing and project write up.	6 days	6 days	
8: Finished project.	14 days		

Some of the early milestones were a little too easy, which cost me later. Had I scheduled the two extra widgets early, I could have moved back the early milestones to give myself more time for the later ones. Clearly milestones 5 and 6 hurt a great deal, being 33% and 40% late, respectively.

Documentation

Tutorial

Make sure your program is using PIGlib, the Portable Independent Game library for creating an OpenGL window and receiving input. Include the following headers:

```
#include "hog/GUI.h"
#include "hog/Controls.h"
```

GUIManager class

Create an instance of the GUIManager class, and create an instance of some subclass of GUIRenderer. HOG comes default with 'LineLooks', a simple transparent theme.

```
hog::LineLooks myGUITheme;
hog::GUIManager myGUIManager(&myGUITheme);
```

Now you'll want to set up input for myGUIManager. Since you are using PIG, you probably have already subclassed the pig::MouseHandler and pig::KeyHandler classes. The GUIManager class has the same functions for handling key and mouse input data, so just pass the message along like this:

```
void MyMouseHandler::mouseDown(unsigned int button, float p_x, float p_y) {
    if(!myGUIManager.mouseDown(p_x, p_y))
        doMyStuff();
}
```

If the GUIManager uses the event, the function will return true. Otherwise, the event missed all of the widgets in the GUI and you should deal with the input in your own way.

Adding Widgets

The best way to add widgets is to define them in a text file and have HOG read them in. Take a look at the 'widgets.txt' file included with the sample code. You can load these widgets like this:

```
myGUIManager.load("widgets.txt");
```

To see how to declare widgets in the text file, read the **Widget Documentation** section. Alternately, you can manually add widgets like this:

```
myGUIManager.addWidget(new Button("Do it!", 0.1, 0.1, NULL));
```

Take a look at Controls.h to see what widgets are available and what their constructors are.

Widget Documentation

Window Widget

```
@Window myWindow {
    title "Window Title Goes Here..."
    size 0.5 0.5
    position 0.5 0.5

    @OtherWidget myEmbeddedWidget {
        ...
    }
    ...
}
```

Access Functions:

None

Button Widget

```
@Button myButton {
    title "Do it!"
    position 0.5 0.5
    function "myButtonFunc"
}
```

"myButtonFunc" called when button is clicked.

Access Functions:

None

Slider Widget

```
@Slider mySlider {
    size 0.5 0.1
    position 0.5 0.5
    value 5 100 # minValue = 5, maxValue = 100
```

```

    continuous 1          # if 0, slider stops only at ticks
    ticks 30             # number of ticks
    function "mySliderFunc"
}

```

"mySliderFunc" called when slider value is changed.

Access Functions:

float getValue() - returns current slider value

ToolPalette Widget

```

@ToolPalette myPalette {
    size 0.5 0.5
    position 0.1 0.1
    dimensions 2 5          # 2 rows, 5 columns
    image "toolicons.tga"
    function "myPaletteFunc"
}

```

"myPaletteFunc" called when a new palette tool is selected

Access Functions:

int activeTool() - returns current tool number

ScrollablePage Widget

```

@ScrollablePage myScrollablePage {
    size 0.5 0.5
    position 0.1 0.1
    height 1.0             # although size is 0.5, can scroll to 1.0

    @OtherWidget myEmbeddedWidget {
        ...
    }
    @OtherWidget myOtherEmbeddedWidget {
        ...
    }
}

```

Access Functions:

None

HSBColorChooser Widget

```
@HSBColorChooser myColorChooser {
    size 0.5 0.5
    position 0.1 0.1
    function "myColorFunc"
}
```

"myColorFunc" is called when the user selects a different color

Access Functions:

gfx::Color getColor() - returns current color selection

RadioButtonGroup Widget

```
@RadioButtonGroup myRadioButtonGroup {
    size 0.5 0.5
    position 0.1 0.1
    columns 2 # number of columns of buttons, default is 1
    buttons "First one!" "second..." "tres" "fourth button!"
    function "myFunc"
}
```

"myFunc" is called when the user selects a different radio button

Access Functions:

int getSelection() - returns id of current selected radio button

CheckBox Widget

```
@CheckBox myCheckBox {
    size 0.3 0.1
    position 0.1 0.1
    state 1 # initial state on/off
    function "myFunc"
}
```

"myFunc" is called when the user toggles the checkbox

Access Functions:

bool getState() - returns true if checkbox is selected, false otherwise

TextField Widget

```
@TextField myTextField {
    size 0.3 0.3
    position 0.1 0.1
    text "Initial text goes here..."
    editable 1 # is text editable?
    function "myFunc"
}
```

"myFunc" is called when text is edited

Access Functions

std::string getText() - returns current text in text field

ScrollableTextField Widget

```
@ScrollableTextField myScrollableTextField {
    size 0.3 0.3
    position 0.1 0.1
    text "Initial text goes here..."
    editable 1 # is text editable?
    function "myFunc"
}
```

"myFunc" is called when text is edited

Access Functions

std::string getText() - returns current text in text field