guiliani.de

Guiliani Tilt Maze Demo Exposed

A short walkthrough to explain Guiliani Tilt Maze Demo and how it works

Product:	Guiliani Technical Showcase (Tilt Maze Demo)
Release version:	2.5
Release date:	December 1, 2023

Table of contents

guiliani.de

1 Introduction / Intended audience

This manual explains how the Guiliani tilt maze demo works. It lists all screens and the actions related to those screens. And it describes how everything is done on a very high level. When reading the code-parts of this tutorial you should be able to understand the basic principles of object-oriented-programming.

Note: We are explaining only how the methods connect to the GUI and the application and what parts of Guiliani are involved.

2 Helpful resources

There are basically two main resources which will help to enhance your knowledge about Guiliani and how are things done in the GSE:
- GSE Manual (optional: GSE Control Attributes)
- Guiliani API Documentation

If you encounter any questions which are not answered in this tutorial, please refer to any of these resources.

3 View the project

In order to examine the project in more detail you will need to open it in the GSE as well as the corresponding source-files in your text-editor or IDE. Please find more detailed description on how to do this in the documentation of the GSE and the used IDE.

4 Overview

Currently the tilt maze demo has multiple screens (screen-filling dialogs.) One menu and two levels. With the two buttons in the main menu either the tutorial or the actual game can be selected. The tutorial is an easy and explanatory game level which starts directly after switching the screen. The game screen has a checkbox and a start button. With the checkbox it is possible to select the difficulty by adding holes to make the game harder. Under the checkbox there's a start button which starts the game.

Playing the game works by tilting the whole device, the accelerometer/gyroscope is then read and the ball rolls according to the tilt. The game is lost when the ball "falls" into one of the holes. To win, the ball needs to enter the goal, which is the green field at the end of the maze.

At any time of the game, it's possible to enter the main menu again, by pressing the hardware button SW2. Similarly, the current level can be reset by pressing the hardware button SW1. Both options are displayed in the info overlay, which gets shown, when the game is either lost or won.

5 General Explanations

5.1 Parts of a Guiliani-Application

Every application using Guiliani will consist of the visual description (properties, dialogs and resources) created in the GSE and the code using the Guiliani-API to communicate with the GUI and create dynamic behaviour. The code will include the pre-built libraries, and some start up-code additionally to your business logic. The starting point for your application-code will be the file *MyGUI_SR.cpp* located in *<APP>/Source*-folder.

5.2 Dynamic behaviour

For many purposes there are built-in dynamics (*Behaviours* and *Commands*) you can use directly in the GSE-project without writing any line of source-code. That can be moving and/or resizing objects, changing visibility or transparency.

The most often used way in the Guiliani Tilt Maze Demo to process events from the GUI in the application-code is the *CallAPI*-command which can execute all sorts of things directly in the GUI-thread. Guiliani will call the *CMyGUI::DoCallAPI()* method with the strings API and Parameter you have specified in the GSE.

If you are unsure what is executed when interacting with an object, just click on that object in the GSE and examine the attached dynamics in the attribute-window.

For more information about the internals of the executed *Behaviour* and/or *Command* please refer to the Guiliani API Documentation.

5.3 Creating Applications with multiple screens

When your application consists of multiple screens which can be shown depending on various actions, you can use Dialog-Transitions to move from one dialog to another. There are several settings for the visual transition made during the dialog-switch as well.

In general the following procedure is done:
- Load the new dialog, it will be invisible by default
- Transition to the new dialog using blend, crossfade, dissolve, push, …
- Delete the old dialog

In the tilt maze demo this will be done in the background. Only a single *CallAPI*-command needs to be called to change to a different screen: *CallAPI*("switchDialog", "<file name of dialog>")

To prevent data-loss it is advised to save all relevant data from the former dialog **BEFORE** the transition is started. This can easily be done by using a *CallAPI*-command beforehand.

After the new dialog is shown there may be additional initialization needed for *TextFields*, *ComboBoxes*, etc. The new dialog will be loaded and shown as it was designed in the GSE. If you want to run initialization on any object in the new dialog, you can do this now by using *CallAPI* again.

Internally the tilt maze demo uses three commands which are executed when switching to a new dialog. These commands are executed from the menu located in *<APP>/Source/MyGUI_SR.cpp*.

- *DialogTransition*
- *CallAPI* ("*InitLevel*")

Deinitializing happens within the *SwitchDialog* API call.
After that the transition to the next dialog is performed.
When the transition is done, the second *CallAPI*-command will set up all things needed to interact with the controls according to the logic of the dialog.
In case a newly added dialog is not meant for playing the maze, special care must be taken, please orient yourself on the Menu dialog, as it also contains no playable level.

5.4   Shared functionality

To avoid duplicate code and the additional complexity and code size, functionality that's shared across multiple screens is handled within one place.

5.4.1   Game

The *CGame* class contains the collision detection code for the obstacles, holes and the goal.
There is only one instance of this class, and it gets its information from the *CLevel* class.
This class gets populated depending on which dialog is loaded.

5.4.2   Overlay

The dialog *Overlay* is used not as a screen (screen-filling dialog,) but rather an overlay for the current screen. It is managed by the *CMyGUI* as it is used in all playable screens. It displays the win and lose overlays, as well as an information overlay, which shows that the hardware switches can be pressed to interact with the application.

5.4.3   Hardware

The application has a *CHardware* class, it's used to manage the buttons, as well as the accelerometer of the used board. To enable easy porting to new platforms, the *CHardware* class should be inherited into a new class where hardware specific operations can occur.

For example, the *CHardwareMock* class is made to mock the hardware so the application can be run on a platform without an accelerometer. There it's simulated with the arrow keys.

6 The application explained

Now every part of the application will be described, looking at the GSE-project as well as the code. After explaining the code that controls the complete GUI, we look at each part, first we will have a look at how the screens look like. After that we will go through the code to look at the Guiliani-specific parts used.

If any function is unclear, please refer to the official Guiliani API Documentation.

6.1 CMyGUI

Source-File: *<APP>/Source/MyGUI_SR.cpp*

The *CMyGUI* class will exist as long as the application runs, as it controls almost everything. It manages the currently shown screen by creating them and forwarding calls from the GUI created in the GSE to it.
It also handles the switching of the dialogs/levels. So, it initializes and deinitializes them.
Changes of the level or game is also handled inside this class.
The method *DoCallAPI()* is used as entry point for the communication from the GUI to the code.

6.2 Screens

The screens are the dialogs that fill the whole screen handled by Guiliani.
In the tilt maze demo are two kinds of screens, a menu screen and two game screens.

6.2.1 Menu screen

Source-File: *<APP>/Source/CDialogMenu.cpp*

This dialog only contains two buttons, which are used to change to the appropriate screen.
As explained in 5.3 to change the screen in the tilt maze demo, only a single *CallAPI-*Command needs to be called. This is done in the GSE, so the *CDialogMenu* class is quite barebones.

GUICallAPICmd	
CommandClassID	CMD_CALLAPI ▼
ApplicationAPI	switchDialog
Parameter	Level_Tutorial

Because the *switchDialog* command is designed to change levels additionally to the dialogs, special care must be taken when switching to this dialog. This is done in the API call *InitLevel*, when the dialog is *Menu* it does not call the *InitLevel* function, but only clears the Level.

6.2.2 Game screens

The game screens have almost identical function and therefore they use the same class *CGame*.
To prepare the maze, the *GenerateLevel* function is used, it repopulates the *CLevel* instance.
The *OnHardwareChange* function is used to handle input from the hardware using the *CHardware* class further explained in 6.5. There the current game will be reset if the hardware button SW1 will be pressed and when the hardware button SW2 is pressed the game will stop and the screen will be switched back to the menu screen.
Additionally, the *CHardware* class is also used to access the accelerometer of the board, to control the ball.
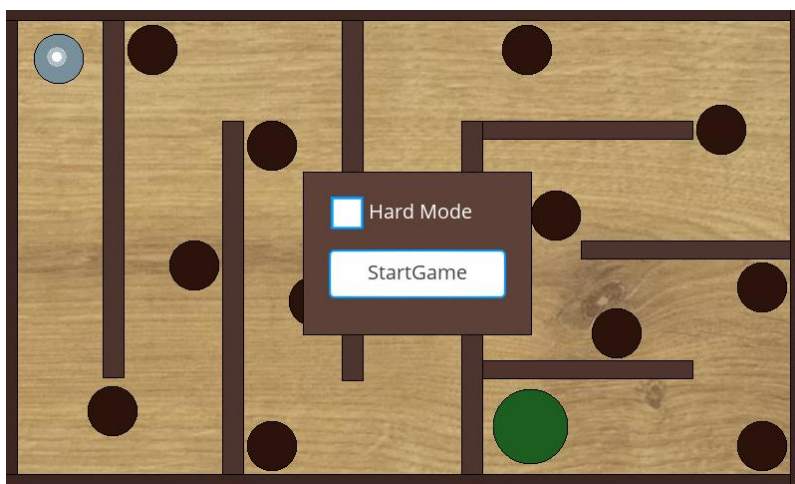
6.2.2.1 Tutorial
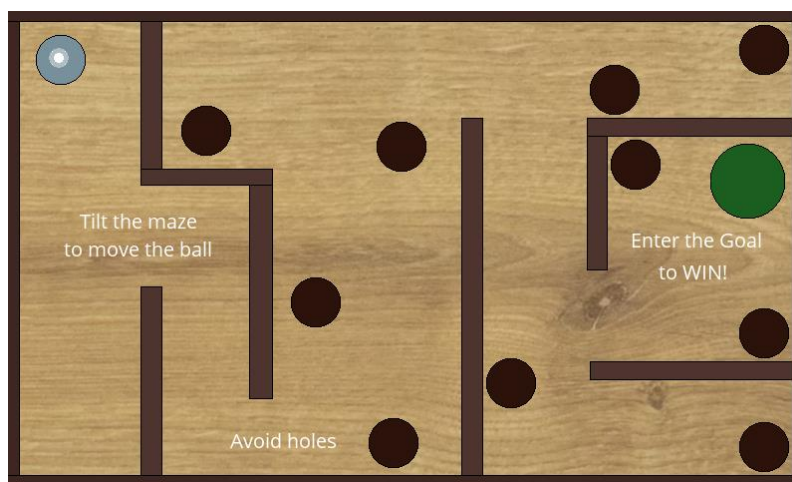
Source-File: *<APP>/Source/CDialogTutorial.cpp*

The tutorial keeps things simple and just starts the maze right after opening the dialog. Therefore there is not much to do other than initializing the game.

6.2.2.2 Level 1

Source-File: *<APP>/Source/CDialogLevel1.cpp*

This screen has a simple menu before starting the maze. There it is possible to activate a harder version of the level. It manages this by simply adding more holes to the playing field. This can be done by toggling the checkmark. When doing so, the holes will be added or removed accordingly, so it's possible to see what it will look like. To observe the checkbox and know when to change the holes the *AddSelectionObserver* method of the checkbox object is called in the *CMyGUI::InitLevel* function and then handling a change in the *OnNotification* function.

The menu also has a button to start the maze. When starting the maze, the menu will be hidden. Starting the maze is done via a *CallAPI*-Command with the API call *Game* and the corresponding parameter *start*. This call will be handled by the *CMyGUI* class, simply hiding the screen and starting the game.

6.2.2.3 Customizing Levels

It is quite easy to modify a level. For example, the layout of the holes and obstacles, as well as the position and size of the goal, and even the ball, can just be changed in the GSE. As long as they are put in the correct *CGUICompositeObjects*, new obstacles and holes can just be added. The only thing to look out for is, holes are round, and obstacles are rectangular. Decorations – like the instructions on how to play in the tutorial level – can just be added, and the ball will ignore them. After the level was modified it just has to be exported, and the resource file needs to be loaded onto board.

6.2.2.4 Adding Levels

Adding levels is also quite easy but need a bit new code. First of the level needs to be created in the GSE. To do so, open the project in GSE and create a new dialog, by pressing the ⊟ - button (New Dialog…) in the toolbar or in the *File* menu from the menu bar. After naming it something like *Level_MyCustomLevel* and setting a fitting ID to the root element like DLG_LEVEL_MY_LEVEL. We'll need a ball, it can be any *CGUIObject* but its physics in game will always behave as if it would be round, so it is recommended to use a *CGUIGeometryObject* with its shape set to ellipse, or an *CGUIImageObject* with a round image. Its height and width need to be the same, as to not have weird behaviour when playing. Set the ID of the ball to BALL.

Next the game needs a goal, this again can be any *CGUIObject* and its collision is checked as if it were round. Here it is not as important to have a round looking object, but it should still have the same height as width.Set the ID of the goal to GOAL.

And because a game without obstacles would be quite boring, a *CGUICompositeObject* with the ID OBSTACLES can be added, within this container multiple obstacles can be added. The obstacles don't need to have an ID and can be any *CGUIObject*. Their collision is checked as if they are rectangular, therefore it is recommended they also look like solid rectangles. It is also recommended to add four obstacles on the edges of the screen, so the ball won't roll out of view.

To add a bit of risk to the game, holes can be added. This works similarly to the obstacles. Just add a *CGUICompositeObject* with the ID HOLES and add any *CGUIObject* into it. Their collision will be checked as if they are round, so it makes sense to also make them visually round. Again, the height and width should be the same. After a ball falls into a hole it gets stuck in the middle of it, and it will loose some opacity. This has the visual effect of a shadow that's cast onto the ball.

To be able to get to our level, add a button in the main menu and labeling it something like *MyLevel*. In the attributes window of the button under *GUICommand* choose the *CommandClassID* "CMD_CALLAPI" then write "switchDialog*"* into the *ApplicationAPI* field. The Parameter needs to be the name of the level as it is shown in the Dialogs window. If you followed these instructions, it should be *Level_MyCustomLevel*. To clean up the menu a bit and avoid confusion, change the label of the *Game* button to *Level 1*.

After exporting, we are done in the GSE.

Now we need to add a bit of code to get to the new level and get it running. For this go into the CMyGUI::DoAPICall function and look for the API call InitLevel, within this code block there is comment "Add new level/dialogs here!" just above this command add the following code:

```
else if ("Level_MyCustomLevel" == kParam)
{
    InitLevel(DLG_LEVEL_MY_LEVEL);
}
```

If you choose a different dialog name or ID, you have to change them here accordingly.

Now you can compile the code, flash it to your board and play your new maze.

If you added more than those basics to your dialog and don't want your game to start directly after the dialog loaded, then add the parameter false to the *InitLevel* function call after the ID. This will stop the application from starting the game automatically. You then can edit the *CMyGUI::InitLevel* function, so you'll can add your specific initializing procedures there, just like DLG_LEVEL_1 does. Your custom deinitializing can be done in the *CMyGUI::SwitchDialog* function.

6.3   Game Logic

The game logic is separated into two classes *CLevel* and *CGame*. *CLevel* contains the data that is relevant to the current level whereas *CMaze* uses this data to move the ball accordingly.

6.3.1   Level

Source-File: *<APP>/Source/CLevel.cpp*

The *CLevel* class contains the *CGUIObjects* needed for the maze. Those objects are the goal, the obstacles and the holes.
Holes of one set are within a *CGUICompositeObject*, those sets are then within a list. This way it is possible to set different holes for the same level, for example to change the difficulty as in Level 1. Switching between different hole sets is done through the *SetActiveHolesIndex* function. This function takes the parameter *iHoleIndex* which defines which set should be used and the optional parameter *bUpdateVisuals* which defaults to true and will define if the function should update the hole sets, so only the currently active is set to visible.

Additionally, the class contains a starting point for the level, there the ball will start on a new or restarted game, regardless of where it was placed in the GSE. By default the place set in the GSE is used, this can be circumvented by simply setting the starting point to another position in the *CMyGUI::InitLevel* function (see 6.2.2.4 for more information about dialog specific initializations.)

6.3.2   Game

Source-File: *<APP>/Source/CGame.cpp*

The maze works by utilizing the *CGUIAnimatable* class and the *DoAnimate* function that gets called by the internal Guiliani timer in a specified period. This way the *DoAnimate* function can be used as a game loop. Telling the timer to call this function is done with the following code:

```
GETTIMER.AddAnimationCallback(m_iSampleRate, this);
```

The *m_iSampleRate* is a member variable of *CGame* with the purpose of telling the timer how many milliseconds should be waited between each call. This needs to be passed, so the timer knows it should call the *DoAnimate* of this instance of *CGame*. The *Start* function of the class calls this function to start the game. Similarly, the *Stop* function calls the following function to tell the timer to stop calling *DoAnimate*:

```
GETTIMER.RemoveAnimationCallback(this);
```

In the *DoAnimate* function the ball gets moved according to the rotation of the device, as if the ball would be lying on top of the display. If the ball hits an obstacle through this movement, the position of the ball will be calculated as if it bounced off the obstacle. If the ball falls into a hole or rolls into the goal the game will notify the *CMyGUI* class with this info. The class then decides how to continue from there.

6.4   Overlay

The dialog *Overlay* is different to the *Menu*, *Tutorial* and *Level 1* dialogs as it is not a screen, because it is not the main dialog shown at any time, and it does not have its own class in the code. As the name suggests this dialog is shown on top of other screens. As there is no dialog transition when the overlay should be shown, the overlay does not contain any functionality besides displaying information and it is used in multiple screens, there is no need for a dedicated class and the displaying and hiding of the dialog is done in the *CMyGUI* class.

To use the overlay from any screen, the visibility can be set via the API calls *showOverlay* and *hideOverlay*. Where the *showOverlay* needs an additional parameter ("won", "lost" or "info") to know which part of the overlay should be shown. The *hideOverlay* does not need an additional parameter. If it is given, only the corresponding part of the overlay will be hidden, otherwise the whole dialog will be hidden.

www.guiliani.de

6.5 Hardware

Source-File: *<APP>/Source/CHardware.cpp*

The necessary hardware to play the maze, namely an accelerometer and two hardware buttons, are handled by the *CHardware* class. More specifically, the classes that inherit from the *CHardware* class. The inheritance is made, so it is more easily to port the tilt maze demo to other platforms. To use it on a platform without the necessary hardware, there is the *CHardwareMock* class. This class together with the *DoKeyDown* and *DoKeyUp* functions in *CMyGUI* is used to substitute the hardware and its values with key presses. In this mockup the accelerometer values are modified with the arrow keys, the SW1 button is substituted by the A key and the SW2 button by the Z key.