Ibero-American Symposium on Computer Graphics - SIACG (2006), pp. 1–4
P. Brunet, N. Correia, and G. Baranoski (Editors)

GLAZE: A flexible GUI engine for video games

Thijs Kruithof[1] and Rafael Bidarra[2]

[1]Coded Illusions B.V., Rotterdam, The Netherlands
[2]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, The Netherlands

Abstract

*In most current video games, a Graphical User Interface (GUI) engine is used that is tightly intertwined with the graphics engine. This paper presents GLAZE, a flexible GUI engine for games. As a game-oriented User Interface Management System (UIMS) [RHM*88], GLAZE provides a game engine independent solution for all games requiring improved GUI functionality. GLAZE is designed to be integrated with any game engine with relative ease, and does not restrict the use of any of the functionalities provided by an available game engine. By maintaining a clear separation between the state and the presentation of GUI elements, GLAZE is capable of handling a wide variety of GUI types.*

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [Information Interfaces and Presentation]: User Interfaces, H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems, I.3.6 [Methodology and Techniques]: Interaction techniques, D.2.2 [Software Engineering]: Design Tools and Techniques

1. Introduction

Through the years video games have become increasingly complex. The video games we see today take years to develop and demand ever increasing budgets. A direct result of the increasing complexity of video games is that the role modularization plays is more important than ever before. Games are nowadays built up out of several intertwined subsystems such as graphics engines, physics engines, sound engines, networking engines and also GUI engines, the subsystem responsible for all GUI management.

Most of the games currently on the market reuse existing game technology such as graphics engines of other games, or use commercially available engines that can be easily integrated such as physics systems. GUI engines, however, are mostly built specifically for one graphics engine and are therefore not easily usable with any other game system, unless it is based on the same graphics engine. [Jor04]

This paper describes the main features of GLAZE, a new GUI engine for games designed to be independent of the game engine. This is achieved by not enforcing any requirements concerning the presentation of the GUIs.

Another issue that GLAZE addresses is the fact that all present GUI engines used in games are applied to specific elements of the game. Mostly, only the menu structures and HUD (Heads-Up Display) elements, with which a player can interact, are handled by the GUI engine. In a typical video game in which a player can walk around in a three dimensional virtual world, the GUI engine is almost always responsible for rendering the status indicators and any present menu structures. However, any interaction between the player in the virtual world and the virtual world is handled by completely different systems. When for example we have a switch in our virtual world that can be pulled by our virtual player this will not be handled by the GUI engine, although it has a strong resemblance to toggling a switch in the settings menu of the game. GLAZE can be used for both of these types of interaction, due to its consistent separation between the behavior and the presentation of an interactive game element.

2. Engine Design

GLAZE was designed with the following goals in mind:

- The engine should be well suited for the use in video games.
- The engine should be reusable. One should be able to integrate it into different games using different technologies, with relative ease.
- The engine should not assume anything on the presenta-

tion part of the GUI. This enables the use of GLAZE for interaction with elements within the game world next to the usual GUI elements such as menus and HUD status indicators.

To get a clear picture of GLAZE's design we will first cover the core, the heart, of GLAZE, followed by a description of the overall system design.

2.1. Foundation

GLAZE is built around the idea that the user of a graphical user interface does not directly interact with something he observes. In GLAZE's world, a user interacts with a state, the *inner state*, of an object which in turn influences an object, or rather a *presentation* of an object, that the user observes.

An object with which the user can interact is therefore within GLAZE defined by a combination of a inner state and a presentation element.

Inner State

The inner state of an interactive object represents the current state an object is in. This state consists of a set of *attributes* with their current values and a set of behaviors that specify how the current state should be altered upon an external event. External events for which actions can be specified are typically the actions a user can perform with an input device.

A GUI button that will be pushed when the user presses the A button on the controller will have an attribute named **pressed** in its inner state. This attribute will initially have the value **false**. The inner state of the button will also possess a behavior, which sets the **pressed** attribute to **true** upon receiving the external event specifying that the user pressed the A button on the controller.

Inner states can also inherit from other inner states, giving the GUI designer the well-known benefits of generalization and specialization. This also simplifies the creation of separate libraries which can be used for the creation of new GUI's.

Presentation Element

A presentation element describes the entities the user observes and how the inner state influences this observation.

For example, for a typical WIMP style [vD97] button (as found in interfaces using Windows, Icons, Menus and Pointers), the presentation element consists of a drawing of a rectangle with a caption. And the button will contain a behavior specifying that the rectangle should be drawn sunk when the **pressed** attribute of the inner state changes to **true**.

2.2. Architecture

GLAZE is decomposed into several components. When looked upon from a distance we have a hierarchy of GUI

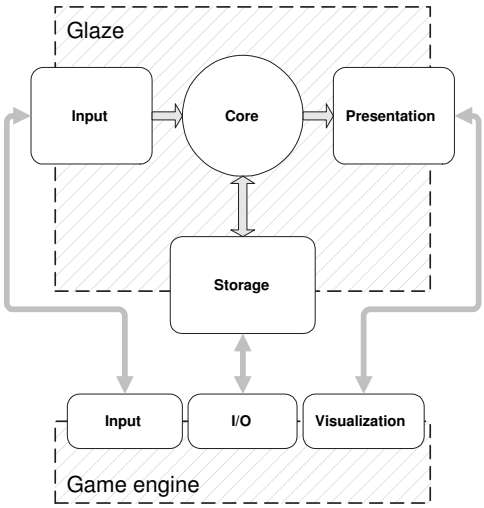

Figure 1: *Subsystem decomposition of* GLAZE

definitions, containing all inner states and presentation elements, as the heart of the system. This central kernel is connected to three other components that provide functionality for communication with the outside world, the game engine. Each of these three components, the **input** subsystem, the **presentation** subsystem and the **storage** subsystem, performs its own tasks. Figure 1 illustrates this architecture.

Within the total engine, the **input** subsystem is responsible for monitoring any user input and presenting this to the core system in the form of events. The information regarding the status of the input devices is provided by the game engine, since the game logic itself also relies on this status information of the input device. The input subsystem of GLAZE therefore has to communicate with any input subsystem present in the game engine.

Next to the input subsystem, GLAZE also has a **presentation** subsystem. The presentation subsystem is responsible for having the game engine *present* the GUIs. A GUI designer can define a presentation element in terms of elements, or primitives, known by the game engine. GLAZE however has no knowledge of, for example, what a button looks like. It is the presentation subsystem's task to have the game engine render the button as it was specified by the designer. This requires this subsystem to communicate with the game engine.

Finally the **storage** subsystem is responsible for all low-level data storage and retrieval functionality. Since the storage techniques used in a game are very much game dependent, it is the storage subsystem's task to delegate all storage and retrieval requests to the game engine.

Our core module uses these three subsystems to communicate with the game engine. Any input device status infor-

mation provided by the input subsystem is processed and will result in the execution of any actions defined in the inner state of an active GUI element. When this results in a change of one or more inner states within the GUI, then this can also lead to a presentation element being modified. This requires the core module to have the presentation subsystem update the presentation of one of the GUI elements.

The core module also relies on the functionality provided by the storage subsystem. This system is used for the storage and retrieval of any GUI definition that is requested at runtime. GLAZE demands a certain set of functionality needed for loading GUI definitions, but how this functionality is implemented is up to the game engine.

2.3. Implementation

To simplify the integration with existing game technologies, GLAZE is a library written in C++. The choice for C++ also enables GLAZE to be built and used on a variety of hardware platforms.

Both Inner States and Presentation Elements contain descriptions of behaviors i.e. responses to external events. These behaviors are implemented as small scripts written in the scripting language Lua [IdFF96]. Lua is a language designed and implemented at the PUC-Rio in Brazil that can easily be embedded in an application. Lua also comes with a simple and familiar syntax and fast runtime execution times for scripts: two features GLAZE benefits from.

Game engines, just as games, have the tendency to start up quite slowly and thus forcing the user to wait before being able to perform any work. In GLAZE, behaviors, in the form of scripts, are implemented in such a way that they can be modified and reloaded by GLAZE during execution. This decreases the duration of the typical design-implement-test cycles used for implementing GUI's in games, since the game engine only has to be started once.

3. Engine Functionality

In this section, the main distinguishing features of the GLAZE engine are briefly presented.

3.1. System Integration

One of the design goals for GLAZE was to easily be able to integrate it with different games using different technologies. The separation of our system into one core module and three communication subsystems greatly facilitates this.

Our core module is explicitly designed to be game engine independent. All three subsystems are tightly coupled with the game engine, each through its own interface. Integrating

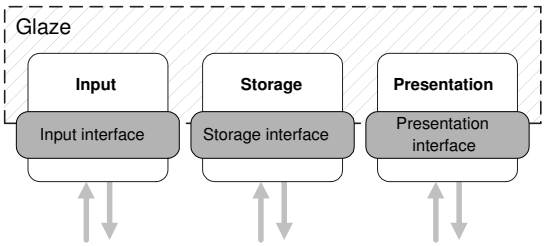

Figure 2: *External interfaces required for the integration of* GLAZE

GLAZE into a game engine requires three interfaces to be established, one for each subsystem, as depicted by figure 2. These interfaces are as it were the 'glue' between GLAZE and the game engine.

By implementing these interfaces, GLAZE can be embedded in any game engine, making it virtually possible to present one single GUI definition within any game. However, GUIs defined for GLAZE are very much game engine dependent. A GUI defined in GLAZE relies on a set of input events and on some available presentation functionality. The set of possible input events is a result of the implementation of the interface between the input subsystem of GLAZE and the game engine. The functionality provided by the presentation subsystem of GLAZE is also a result of the implementation of the interface between that same subsystem and the game engine. A different game engine requires different interfaces which results in a different set of functionality available within a GUI definition. Therefore, although a particular GUI defined within GLAZE can possibly be game engine dependent, GLAZE itself is not.

3.2. Users

As a GUI engine, GLAZE has to deal with two quite different types of users.

The first type of users are the GUI implementers. GUI implementers typically use GLAZE to "run" their GUIs. They deliver GUI definitions in the form of XML documents to GLAZE, which GLAZE processes and executes. GUI definitions consist of descriptions of Inner States and Presentation Elements which are made up out of property definitions and simple behavior scripts, which are written in Lua. To be able to create a GUI by hand some basic knowledge of programming is needed. External tools (e.g. with visual features) can overcome this requirement, which otherwise might form an obstacle for GUI designers and implementers.

The second type of users are the engine programmers. Engine programmers have the task to integrate GLAZE into the existing game architecture. This also includes maintaining the connection between the game engine and GLAZE's subsystems.

3.3. Capabilities

The design of GLAZE offers several unique capabilities to its users, including the following:

- GLAZE can be easily integrated into an existing game engine. Most available complete GUI solutions are tightly coupled to other technologies, such as graphics engines. When developing a game the decision of which GUI technology to use is often only a matter of finding out which GUI system is available for the used graphics engine. Thus in fact limiting the available GUI functionality to the set provided by the included GUI system.
- Due to the fact that almost all available GUI systems focus on providing a fixed set of WIMP GUI elements, there is a technical separation in most games between what is handled by the GUI system and what is categorized as game logic. A switch that is placed in a menu that can be toggled by the player to activate or deactivate a certain game configuration setting is typically an interactive element controlled by the GUI system. But, a switch placed next to a door in the virtual game world, which, when activated, will open the door, is often regarded as a game logic element. Game logic elements are controlled by the game logic subsystem, which is a completely different subsystem than the GUI subsystem. GLAZE is designed to be able to control both types of these interactions. Figure 3 shows an example of the usage of GLAZE for this second type of interaction.
- GUIs for GLAZE are defined by GUI definition documents. These GUI definitions can be hand written by GUI designers. Alternatively, a game engine-specific GUI editor can be created to aid the development of GUIs for complex games. GUIs for GLAZE can also be defined using existing GUI device independent definition language such as the User Interface Markup Language (UIML [APB*99]).
- Due to the hierarchical nature of the inner states and presentation elements that form a GUI, it is possible to create GUI libraries, consisting of definitions of several GUI elements. The creation of libraries simplifies the creation of large or complex GUI structures.

Figure 3: *An in-game soda vending machine powered by* GLAZE

4. Conclusions

The GUI engine presented in this paper, GLAZE, is a flexible UIMS for games. It provides a uniform solution for the management of GUIs within a video game, independently of the used game technology and targeted platform.

The present version of the GLAZE engine is already being integrated in a video game currently in development, targeted at the next generation video game consoles.

References

[APB*99] ABRAMS M., PHANOURIOU C., BATONGBACAL A., WILLIAMS S., SHUSTER J.: UIML: an appliance-independent XML user interface language. *Computer Networks (Amsterdam, Netherlands: 1999) 31*, 11–16 (May 1999), 1695–1708.

[IdFF96] IERUSALIMSCHY R., DE FIGUEIREDO L. H., FILHO W. C.: Lua — an extensible extension language. *Software Practice and Experience 26*, 6 (1996), 635–652.

[Jor04] JORGENSEN A. H.: Marrying HCI/Usability and computer games: a preliminary look. In *NordiCHI '04: Proceedings of the third Nordic conference on Human-computer interaction* (New York, NY, USA, 2004), ACM Press, pp. 393–396.

[RHM*88] ROSENBERG J., HILL R., MILLER J., SCHULERT A., SHEWMAKE D.: Uimss: threat or menace? In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1988), ACM Press, pp. 197–200.

[vD97] VAN DAM A.: Post-WIMP user interfaces. *Communications of the ACM 40*, 2 (1997), 63–67.