B-026

A GUI GRAPHICS LIBRARY FOR EMBEDDED DEVICES

Armand GIRIER, Yusuke FUKAI*, Akira NAKANISHI*, Takeo HORIGUCHI**

Abstract

This paper presents the architecture of a 3D GUI framework realized by extending an already existent 2D framework. By taking advantage of the object oriented architecture of the framework, we modified it to draw its widget using OpenGLES 2.0. As a result, it became able to benefit from hardware accelerated graphics on recent embedded graphic systems. Another extension was to allow manipulating widgets as 3D objects while maintaining their original features.

Keywords: GUI, 3D, OpenGLES, hardware accelerated graphics, GPU, embedded systems.

1 INTRODUCTION

Animated user interfaces have become ubiquitous on general use embedded devices, the most obvious examples being smart phones and tablets. It finds applications in user friendly interfaces and intuitive use of touch panels. The enjoyable user experience it provides makes it a highly desirable feature.

Although graphics animated at acceptable rate require huge amounts of computing power, two factors made it possible even for embedded devices. One is the dramatic reduction of hardware costs along time, which allowed developing reasonably powerful graphic processing units, fit for battery devices. The second is the apparition in 2003 of OpenGLES[1], a subset of the OpenGL 3D graphics API designed for embedded devices, and its generalization as a de facto standard, which made it possible to benefit from said embedded GPU with a small footprint while maintaining portability.

Several GUI frameworks designed for embedded devices offer support for OpenGLES. This paper describes the extension of an already existent GUI framework, to not only support 3D drawings through OpenGLES but become a fully animated 3D GUI framework. We made it use OpenGLES for all of its graphics operations, including the drawing of widgets themselves. This combined with a simple 3D engine and a monitoring framework for the original widgets, resulted in a highly portable 3D animated GUI framework.

2 GENERAL ARCHITECTURE

The GUI toolkit we have extended was designed for portability between different types of operating systems, including embedded types of operating systems. For this, it has to adapt to several graphics middleware and API. This is usually done by providing a porting layer set of classes implementing a draw toolkit interface which API provides routines for drawing primitives like lines, rectangles, text and bitmap images. Classes for Windows, X11 or other systems are provided, and which class is instantiated at runtime is up to the building process.

By providing an implementation that performs drawing using OpenGLES, we can redirect the drawing instructions to the GPU, thus making hardware accelerated graphics available for any platform providing an OpenGLES implementation. The point of this is not that the framework can merely draw

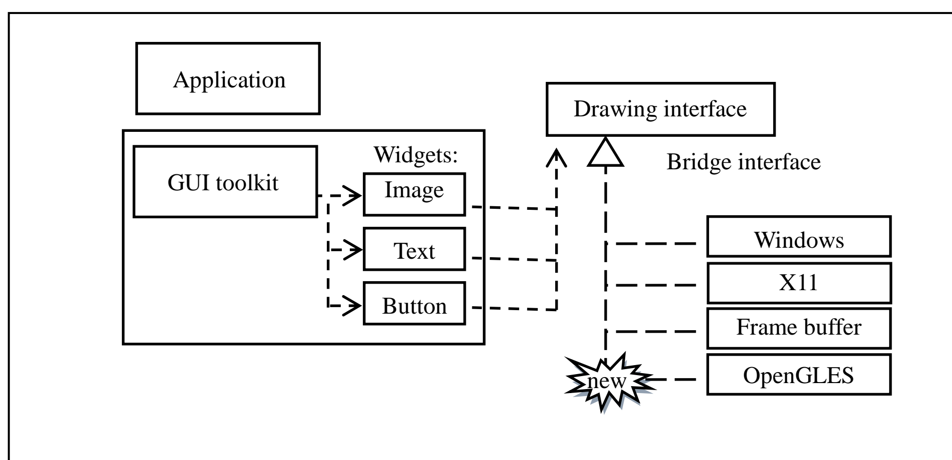

Fig. 1 Structure of a modern GUI framework and support for various platforms

*:Toshiba Corporation, Digital Products & Services Company
Core Technology Center
Embedded System Technology Development Dept.

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

OpenGLES graphics on a canvas but that the very native widgets will be drawn using OpenGLES, this change beeing totally transparent to the other parts of the framework. This drawing mecanism can be easily tested by comparing the graphic result with one of the old implementations.

However, anyone familiar with OpenGLES drawing knows that it requires more thought than the average 2D graphics API. Careful timing and management of the ressources like textures (images) and other memory buffers is necessary for optimised performance. Moreover, features usually not supported by regular graphics API have to be handled, like 3D transformations, transparency, lightning and other effects. This is handled by the simple 3D engine we developped for this purpose.

3 EXTENTION TO 3D

Our goal was to achieve a hardware accelerated GUI framework allowing 3D animation of widgets. In such a framework, one would expect to find [2]:

- ✓ A way to provide 3D transforms, as used in 3D graphics to modify the position, orientation and scale of an object without modifying its own geometry, as well as other setting relevant to 3D special effects.
- ✓ A way to propagate user events like mouse clicks along the 3D engine, in order to determine which object in the 3D space has been picked by a click on the 2D screen.
- ✓ A way to make sure the timing of it all is the one expected by OpenGLES.

As stated earlier, drawing with OpenGLES requires more thought than with the average 2D graphics API. The notion of 3D space has to be taken into account and low level management of resources and timing is necessary to draw anything at all. This supplementary information can't be given through the drawing toolkit API (it was not designed for it) so it was necessary to adapt a simple 3D engine to the original GUI framework. In order to provide the three features above, we made use of a proxy [3] class system that will be explained in details.

GUI frameworks are usually organized in a hierarchical way, with composite widgets including component widgets, and possibly being components themselves in upper level composites. When one modifies the position of a container, the components are expected to move accordingly. 3D engines use the same pattern with tree graphs of ordered objects whose absolute position depends on the absolute position of the objects upper in the hierarchy and their own position relative to their direct parent. The two types of hierarchies can be naturally tied together using proxy objects.

Proxy objects are designed to implement the interface of elements of one hierarchy while transmitting their input to the elements of another hierarchy. They embody a two way communication between widgets and the 3D engine. At drawing time, the 3D engine is in charge of the frame rate. After proper initialization of the OpenGLES state machine, it draws the tree graph - which elements are proxies to widgets - in depth first traversal. Each proxy first sets OpenGLES accordingly, for example - but not limited to - setting its 3D transform. Then, the proxy makes its widget to draw itself using its API. The drawing instructions are converted into OpenGLES operations as stated in section 2. This process of delegating the control of drawing to the 3D engine while disabling it from the original framework gives us the possibility to ensure proper setting of OpenGLES at the proper timing.

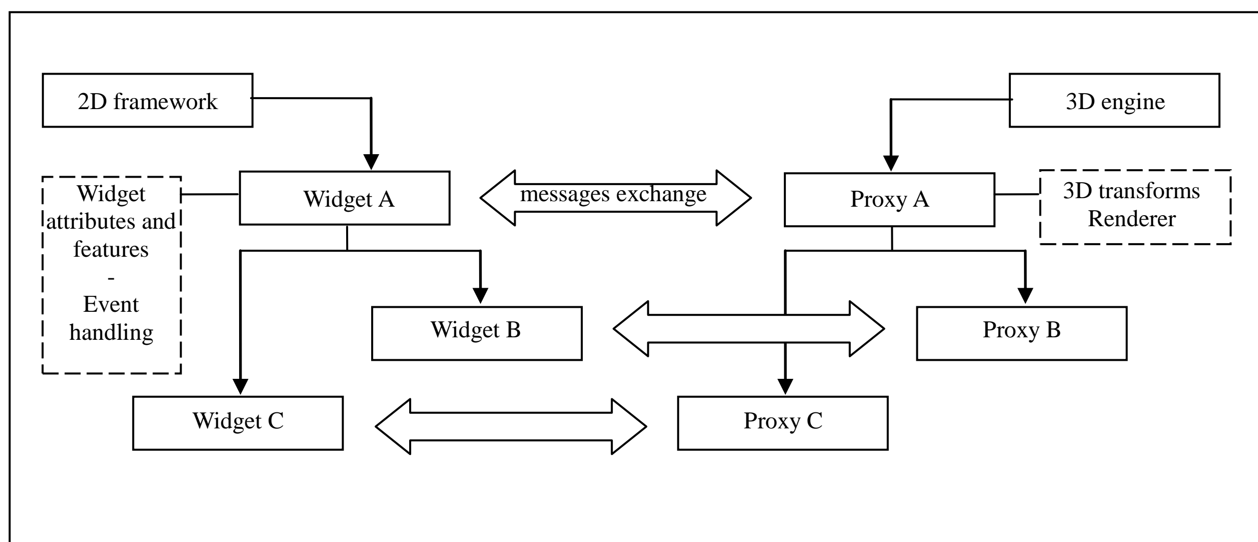

Fig. 2 Extension of the two hierarchies by the mean of proxies

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

4 EVENT HANDLING

In most of the cases propagation of events can be done in the usual way of the original framework because they are not depending on the 3D space (keyboard events, timer events). On the other hand the case of mouse events or touch panel contact, any event that is located on the 2D screen, requires knowledge of the 3D space.

This is done by intercepting them at the root of the widget hierarchy and propagating them in the 3D engine hierarchy instead. The engine having knowledge of the transform of each widget can locate the one that has been picked by the user if there is any. Ray casting is a fairly common way to do this. The corresponding proxy is then used to transmit the event to its widget.

Interception of the original GUI toolkit’s events could be done without any modifications of its code, by registering proper event handlers on the original widget that will notify the proxy. Such registration can be done transparently on linking the widget to the proxy.

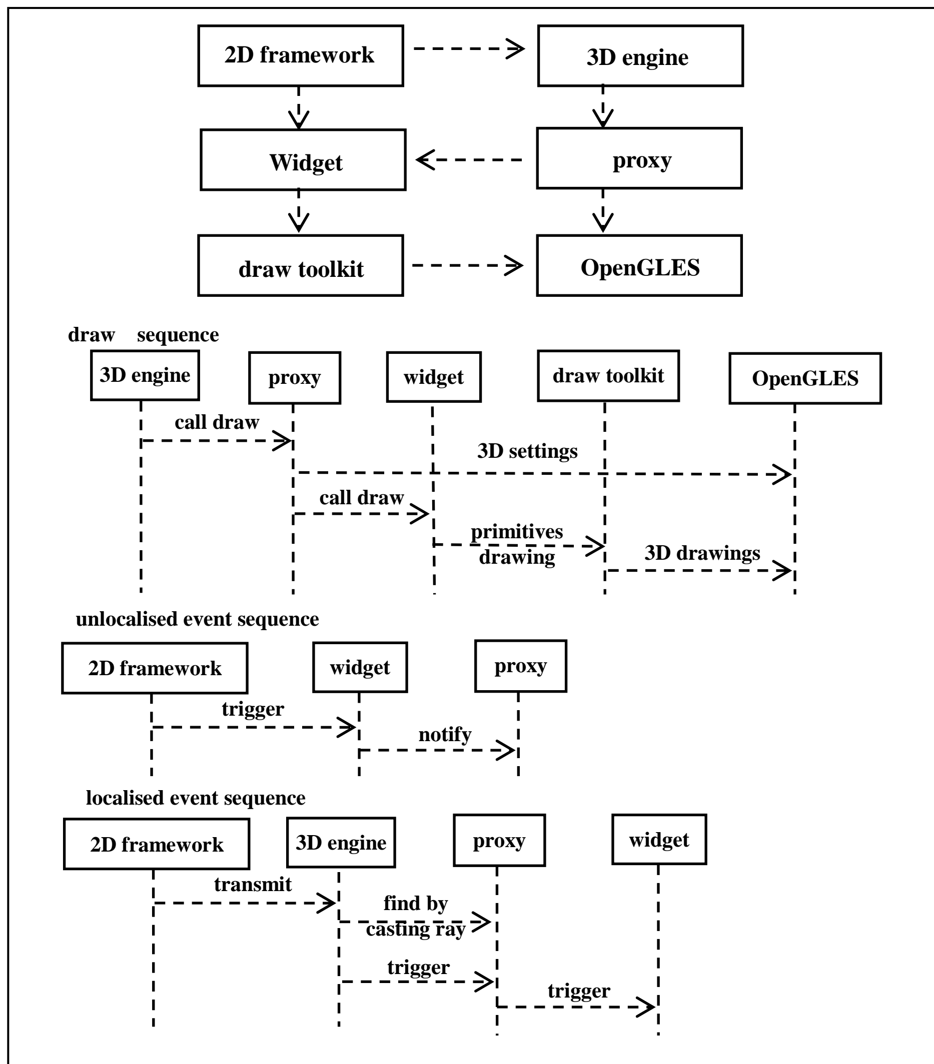

Fig. 3 Circulation of messages between modules and sequences

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

5 SAMPLE CODE

Such framework gives us the possibility to create and user interface and to animate its widgets with simple code.

```
// First create a window, a frame and a widget

Window* win = new Window( … );

Frame* fram= new Frame( win, … );
Proxy* fram_proxy = new Proxy(fram);

Button* btn = new Button( fram, … );
Proxy* btn_proxy = new Proxy( btn );

// Links the button's proxy in the 3D engine hierarchy

fram_proxy->addChild(btn_proxy);

// […]

btn_proxy->setPosition( X, Y, Z);
btn_proxy->rotate( 45 );

// […]

// In the button event handler: rotate when activated

Animation* anim =
btn_proxy->animateToTransform( new_transform );
ScheduleAnimation(anim, start_time, stop_time)
```

6 EMBEDDED GRAPHICS CONSIDERATIONS

The graphic module was developed based on OpenGLES2.0 in order to benefit from its programmable pipeline [4]. It introduced the difficulty that no default fixed pipeline is provided. Re producing a full blown OpenGLES1.1 pipeline is indeed well documented in the literature [4] but raises performance problems depending on the implementation of OpenGLES2.0 it is run on. Shaders with many branch statements tend to run with poor performance because it prevents parallel execution.

In order to counter such inconvenience, we developed a set of specialized shaders with one pipeline for lines, one for plain colors, one for textures, and others designed for special effects like blurring. The shortness of such specialized shader (often less than 5 LOC) allows fast execution even on low power GPUs. In order to use the correct shader at runtime, proxies are affected a "renderer" strategy [3] depending on the type of Widget they have to handle. While traversing the 3D scene graph the engine sets the relevant renderer before calling the drawing method of the widgets.

Performance measurements on various embedded GPUs at our laboratory showed that the overhead of switching between specialized shaders was negligible in regards of the performance lost induced by multifunction, monolithic shaders.

7 CONCLUSION

We have explained the way we extended a standard 2D GUI framework freely available for modification in order to obtain an animated 3D framework, benefitting of graphics hardware acceleration, yet with all the features of the original. This was done by replacing its drawing mechanism in order to draw the very widgets using OpenGLES and adding to it a simple 3D engine for piloting the graphic library.

We also demonstrated that, on embedded GPUs that have low performances when executing long shaders with conditional branches, for example one that would emulate a general pipe line fit for any purpose, good performances could be obtained by switching between specialized, comparatively short shaders depending on the drawing performed at the moment. It also allowed using special effects thanks to the programable pipeline even on embedded level harware.

All of the extensions are based on rather common features of modern GUI toolkits, namely the existence of a graphic portability layer (section 2) and customizable event handler for widgets (section 4). We are confident that such an extention could easily be applied to any modern 2D GUI toolkit.

8 REFERENCES

[1] OpenGLES2.0 http://www.khronos.org/opengles/

[2] Kari Pulli, Tomi Aarnio, Ville Miettinen, Kimmo Roimela, Jani Vaarala, "mobile 3D graphics", Morgan Kaufmann, ISBN 978-012-373727-4 (2008)

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software",
Addison-Wesley. ISBN 0-201-63361-2 (1995)

[4] Aaftab Munshi, Dan Ginsburg, Dave Shreiner, "OpenGLES2.0 programming guide", Addison Wesley, ISBN 0-321-50279-5 (2009)

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.