A Runtime Lifecycle for Interactive Public Display Applications

Alice Perpétua[1,2]
[1]Faculty of Engineering
University of Porto
Porto, Portugal
ei08060@fe.up.pt

Jorge C. S. Cardoso
[2]CITAR/School of Arts
Portuguese Catholic University
Porto, Portugal
jorgecardoso@ieee.org

Carlos C. Oliveira
Faculty of Engineering
University of Porto
Porto, Portugal
colive@fe.up.pt

***Abstract*—Public display systems are becoming increasingly complex. They are moving from passive closed systems to open interactive systems that are able to accommodate applications from several independent sources. This shift needs to be accompanied by more flexible and powerful application management. In this paper, we propose a runtime lifecycle model for interactive public display applications that addresses several shortcomings of current display systems. Our model allows applications to load their resources before they are displayed, enables the system to quickly pause and resume applications, provides strategies for applications to terminate gracefully by requesting additional time to finish the presentation of content, allows applications to save their state before being destroyed and gives applications the opportunity to request and relinquish display time.**

Keywords-interactive public displays; runtime lifecycle;

I. INTRODUCTION

In this paper, we propose a runtime life-cycle model for interactive public display applications. This model allows both the display application and the display system to better manage their resources.

The most common and simple approach for content scheduling in public displays is to follow a timetable where each content item is given a pre-determined amount of display time. In this approach, display systems usually have only one active application at a time, using all the display's resources. Applications are simply instantiated and killed by the display system. This approach works well with time-based content where the content's duration is known, such as in videos, or with non-time-based content where the display owner can easily decide how much display time the content should have, as in still images or text.

However, the movement towards open display systems [1] creates a more complex environment where the traditional scheduling approach may compromise the user's experience. In an open network, display owners can easily interconnect their displays and take advantage of various kinds of existing content, including rich interactive applications. Application developers can create applications and distribute them globally, to be used in any display. Users can not only watch the content played on the display, but also appropriate it in various ways such as interacting with it, expressing their preferences, submitting and downloading content from the display.

In this environment, while display owners may still have control over what is displayed, display systems must be prepared to manage an increasing number of applications in a more flexible and unanticipated way. For example, imagine an interactive video application for public displays where users can somehow select videos to play next. Before displaying another application, the display system should make sure the video is allowed to finish, in order not to disturb the viewing experience. Other applications, such as "background" applications, may require display time in response to asynchronous events such as user interactions or other external events. For example, an application may wish to briefly display a calendar notification only when a specific user or group of users, who subscribed to those calendar notifications, are present. In these situations, the currently displayed application that is about to be interrupted should be able to quickly resume operation after the notification. A more detailed analysis of the challenges of content scheduling in open display networks can be found in [2].

This type of environment requires display systems to function more as operating systems, and it also requires a specific application framework that defines a more fine-grained runtime lifecycle. This will allow a better display resource management just like we have in other platforms. For example, the Android platform defines a rich runtime application lifecycle that breaks down all the possible states and transitions between states of an application from the time it is loaded into memory and started, to the time it is shut down and removed from memory. This break down of possible states allows application programmers and system to negotiate the resources that an application needs in each state, guaranteeing an efficient usage of those resources on the one hand, and rapid application switching and loading, on the other hand. For example, an application may be paused if another application comes to the foreground (e.g., because the user requested another application), stopping animations and other CPU consuming operations and save its state to persistent storage (because paused applications may be destroyed by the system if it needs memory). When the application is resumed, it can start the animations again. It is easy to imagine that display systems will need this kind of resource management when the number of applications that each display handles grows.

In this paper, we present our initial effort in this direction. We have looked at existing computing platforms (mobile and desktop) and their typical application runtime lifecycles and synthesized and adapted those models

according to the specific requirements of a public display system. We have also a first implementation of the proposed model as a Google Chrome extension for web-based public display applications.

The rest of this paper is organized as follow. Section II is dedicated to present relevant related work. Section III addresses the observed shortcomings in existing public displays systems and associated design goals for the runtime lifecycle presented in Section V. Section IV summarize all information gathered about runtime lifecycles of existing computing platforms.

II. Related Work

Many public display content players / content schedulers have been implemented by researchers and industry.

For example, [3] proposes a web-based framework for managing the screen real estate of the UBI-hotspot system - a public display system that supports concurrent applications on a single display. The framework was implemented using Mozilla Firefox browser and custom JavaScript code that manages the temporal and spatial allocation of the screen to various applications. These hotspots support two modes: a passive broadcast mode, and an interactive mode. These two modes represent different ways for deciding when and which application/content should be loaded by the display system. The framework does not support any type of fine-grained control over the execution of an application. For example, if an application takes a long time to load, the user will be aware of this (at best the application may use a splash screen). Similarly, when unloading, the system simply unloads the content, giving no possibility for the application to run clean-up operations. Even if an application is often used, it will always have to be completely loaded and unloaded every time it is used; the system does not put applications in a suspended state for rapid resuming.

Yarely [4] is a public display player for open pervasive display networks that was developed to replace the existing software infrastructure of the Lancaster e-Campus system [5]. Yarely uses a subscription management system where each display node receives a content descriptor set that lists the content that the player should play and how it should be scheduled. It also supports caching of content items so that displays still function under network failures and disconnections. Even though Yarely is a very powerful software player even capable of running native content, it is still geared towards passive content that is scheduled consecutively and where the content length can be known *a priori*. Yarely supports dynamic schedule changes that allow it to display unforeseen content such as emergency broadcasts, but it does not provide any specific support for interrupted content to be resumed.

III. Existing Problems and Design Goals

Work on interactive public display applications [6] [7] has identified a number of shortcoming in existing public display systems. In this section we present the observed problems and the associated design goal for the runtime lifecycle we propose in this paper.

A. Application loading

Many interactive applications have noticeable loading times that designers usually address by showing a splash screen or loading indicator. Loading times may be, in some cases, avoidable or reduced by leveraging on caching techniques, but they are not generally solvable. Many applications, particularly web-based applications, have to set up communication channels with their own servers and with external services. These initialization processes may be hard to circumvent to give users the impression of instant loading. On public displays these loading times represent wasted resources and reduce the user experience: the time an application takes to load could have been used to display the previous content for a bit more time.

Our goal is to create a display system that efficiently manages the screen in these situations by assigning display time only when the application is ready to display useful content.

B. Graceful termination

Interactive applications have no intrinsic duration that display owners can use when setting up their display's schedule. The result is that applications may be assigned an arbitrary time slot for running. For some applications, this results in a suboptimal user experience because they are sometimes interrupted in the middle of an important operation. The interactive video player application is a paradigmatic example: an application that lets users search/select videos to play next. The public display player may terminate this application before the video finishes, representing an obvious failure for users.

Our goal is to allow applications to, within system-defined bounds, request additional display time to finish an import operation or process. Obviously, these requests may not be honored by the system if another content with higher priority needs display time.

C. Forced unloading, pausing, and resuming

Another issue we noticed in interactive applications was the difficulty of running proper finishing processes before the application is terminated. Usually, applications are simply unloaded from the browser component without warning. This results in added difficulty for the application to save state and terminate connections in a proper manner. Although standard web events could be used in this case, they would still be very dependent on the concrete implementation of the player (some players assign browser tabs to applications, others reuse a single tab). Additionally, in some situations it is more efficient to pause and resume an application instead of unloading and reloading it again in the future. For example, if an alert must be displayed, the interrupted application probably does not need to be unloaded, but simply taken to a paused state where it stops most activity, until the alert is removed from the display.

Our goal is to support application termination, pausing, and resuming. The system should allow applications to terminate properly if the application is to be killed. Additionally, applications should be able to quickly resume

operation if they are interrupted by the system, without having to be completely loaded again.

D. Application-requested loading and unloading

Another problem faced by interactive applications for public displays is that they usually have no way to request display time by themselves, or to relinquish the display if they have no possibility to continue. Although some public display players do allow unanticipated content to be displayed, this usually requires manual intervention. Ideally, applications should be able to request display time in order to display short-term notifications, for example. Conversely, applications that find themselves in a situation where they can no longer continue to execute (e.g., because a fundamental resource could not be loaded) should be able to inform the display system and relinquish the display. Obviously, this requires additional management policies on the display system to guarantee that applications do not misbehave and take over the display.

Our goal is to support this kind of operation, allowing display applications to request display time for short periods, and to give up the display time if they are unable to continue operating.

IV. Analysis of Existing Platforms

The main objective of this paper is to describe our initial model for a runtime lifecycle for public display applications. To arrive at this model, we have looked at existing computing platforms in order to learn about the existing runtime lifecycles. We then synthesized these models and adapted the result to take into account our design goals.

We have analyzed the Android platform, iOS, Windows Phone, Windows 8, and Applets platforms. The main event callbacks associated with each platform are presented in Table 1. Each platform has different ways to manage applications and give applications different levels of granularity for managing their resources. However, we can identify commons categories of application states/event callbacks:

Initializing refers to callback methods that are invoked only once by the system, while the application is in memory. All initial routines related to the user interface or data should be done here.

Starting/Resuming refers to callback methods that are called before the application is put into the foreground, either for the first time, or because the user is resuming the application. Different platforms handle this process differently, but in general these callbacks allow applications to start graphical animations, sounds, and other quick initializations. These callbacks may be invoked several times during the lifetime of the application in memory.

Pausing refers to callbacks that signal the application that it is being interrupted and is being taken out of the display, at least partially. In these cases, applications should stop animations, sound, and other CPU intensive operations.

Stopping/Destroying refers to callbacks that signal the application to stop executing, unload all unnecessary resources, and perform state saving routines. Stopped applications may not be immediately removed from memory, but are good candidates to be destroyed and removed from memory if the system needs the resources.

V. Runtime LifeCycle

The model for a runtime lifecycle for public display applications is presented graphically in Fig. 1, and described next.

onCreate() – This represents the application's entry point method and is called only once while the application is in memory. Depending on the implementation, it is possible that application code may execute before this method is called. In our Javascript implementation for example, we cannot prevent applications from executing before the onCreate() method is invoked. However, only after onCreate() can an application interact with the display system and it should not be assumed that the display system is ready before the onCreate() is called.

onLoad() – The onLoad() method is called when the display system decides to give display time to the application. Before the display time is actually assigned to the application, the system calls onLoad() and expects applications to reply with a loaded() method call. At the onLoad() stage, applications should perform all necessary loading routines to ensure the application is ready to be displayed.

onResume() – this callback is called immediately before the application is put visible on the display. At this phase, applications should make sure they are ready to show content. This callback can be used to perform very fast initialization routines such as starting animations. When this method is called there should be no noticeable delay before content is displayed by the application.

TABLE I. Summary of analysed platforms

Callbacks categories	**Platforms**					
	Android	***Android services***	***iOS***	***Windows Phone***	***Windows 8***	***Applets***
Initializing	onCreate()	onCreate()	WillFinishLaunchingWithOptions() DidFinishLaunchingWithOptions()	Launching()	onLaunched()	Init()
Starting/Resuming	onStart() onResume() onRestart()	onStartCommand() onBind()	DidBecomeActive()	Activated()	Activating() Resuming()	Start()
Pausing	onPause()		WillResignActive() WillEnterForeground()	Deactivated()	VisibilityChanged() Suspending()	
Stopping/Destroying	onStop() onDestroy()	onUnbind() onDestroy()	DidEnterBackground() WillTerminate()	Close()		Stop() Destroy()

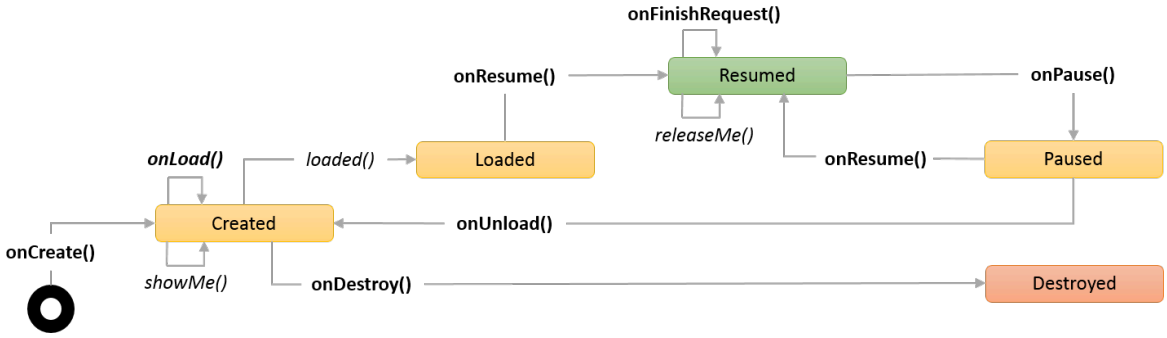

Figure 1. Application lifecyle for public displays.

onFinishRequest() – this callback signals the application that it should finish. In this stage applications should notify the system about how much more time they need to finish gracefully. The system will honor the application's time request, within pre-defined limits, and call onPause() when the time required by the application expires. This callback may not be invoked if the system has another urgent content to display, in which case the onPause() callback will be used immediately.

onPause() – called to signal that the application should pause animations, sounds and other unnecessary operations. In this stage the application is either not visible or only partially visible. Paused applications may be resumed quickly by the system by invoking the onResume() callback.

onUnload() – when an application is closed, it should release all processing resources and clean navigation data as well as state information;

onDestroy() – signals the application that it is being removed from memory. Applications should perform any finalization routines here, perhaps saving state to persistent storage either locally or remotely.

showMe() – Applications can signal the system that they want display time by calling the showMe() method. The system will then apply its internal policy to determine if and when the application should be given display time.

releaseMe() – Conversely, applications can signal the system that they cannot display any more content (perhaps due to a server error or other condition). The system will then take the necessary steps to bring another application to the display.

VI. Conclusions

We have presented a runtime lifecycle model for public display applications that allows a better resource management for display systems that have to handle a high number of independent applications. The model allows applications to load their resources before they are displayed, system to, allows applications to terminate gracefully, allows rapid pausing and resuming, and allows applications to request and relinquish display time.

We have started to implement this model as a Google Chrome Extension where each application is assigned a browser tab. Our implementation manages the lifecycle of each application determining which tab should be displayed at any time. We support two types of applications: foreground and background applications. The display owner schedules foreground applications, to be shown for pre-defined periods of time. Background applications are loaded at startup by the system, but are only assigned display time when they request it. Our system will apply a priorities scheme to determine which applications can interrupt which applications. It will also manage the system memory resource by dynamically destroying and creating applications based on their memory footprints and usage pattern.

Acknowledgements

This paper was financially supported by the Foundation for Science and Technology — FCT — in the scope of project PEst-OE/EAT/UI0622/2014.

References

[1] N. Davies, M. Langheinrich, R. Jose, and A. Schmidt, "Open Display Networks: A Communications Medium for the 21st Century," Computer (Long. Beach. Calif)., vol. 45, no. 5, pp. 58–64, May 2012.

[2] I. Elhart, M. Langheinrich, N. Davies, and R. José, "Key Challenges in Application and Content Scheduling for Open Pervasive Display Networks," in Work in Progress Session PerCom 13, 2013, pp. 393-396.

[3] T. Linden, T. Heikkinen, T. Ojala, H. Kukka, and M. Jurmu, "Web-based framework for spatiotemporal screen real estate management of interactive public displays," in Proceedings of the 19th international conference on World wide web - WWW '10, 2010, p. 1277-1280.

[4] S. Clinch, N. Davies, A. Friday, and G. Clinch, "Yarely: a software player for open pervasive display networks," pp. 25–30, Jun. 2013.

[5] O. Storz, A. Friday, and N. Davies, "Supporting content scheduling on situated public displays," Comput. Graph., vol. 30, no. 5, pp. 681–691, 2006.

[6] J. C. S. Cardoso and R. José, "Evaluation of a programming toolkit for interactive public display applications," in Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia - MUM '13, 2013, pp. 1–10.

[7] J. C. S. Cardoso and R. José, "PuReWidgets: a programming toolkit for interactive public display applications," in Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems - EICS '12, 2012, p. 51-60.