Design and Implementation of a New Run-time Life-cycle for Interactive Public Display Applications

Jorge C. S. Cardoso[1] and Alice Perpétua[1,2]
[1]*CITAR/School of Arts, Portuguese Catholic University, Porto, Portugal*
[2]*FEUP, University of Porto, Porto, Portugal*
jorgecardoso@ieee.org, ei08060@fe.up.pt

Keywords: Interactive Public Displays, Run-time Life-cycle

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 a more flexible and powerful application management. In this paper, we propose a run-time life-cycle 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 transition and 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. We have implemented our model as a Google Chrome extension that allows any computer with the Google Chrome browser to become a public display driver without further software. In this paper we present our model, implementation, and evaluation of the system.

1 INTRODUCTION

In this paper, we propose a run-time 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 playlist where each content item is given a pre-determined amount of display time. In this approach, display systems instantiate and kill content according to their scheduled time. 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 (Davies, Langheinrich, Jose, & Schmidt, 2012) 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 efficiently manage the resource of an increasing number of applications in a more flexible and unanticipated way.

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 run-time life-cycle. This will allow a better display resource management just like we have in other platforms. For example, the Android platform defines a rich run-time application life-cycle 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 the design, implementation and evaluation of a new life-cycle model for public display applications.

The contributions of this paper are twofold:

1. A new run-time life-cycle model for public display applications that allows a fine-grained management of the display and application resources.
2. An implementation of the proposed model in a public display application player as a Google Chrome extension, available as an open-source project (Cardoso, 2014a).

2 RELATED WORK

Many public display content players / content schedulers have been implemented by researchers and industry. For example, (Lindén, Heikkinen, Ojala, Kukka, & Jurmu, 2010) 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. The UBI-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 (Clinch, Davies, Friday, & Clinch, 2013) 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 (Storz, Friday, & Davies, 2006). 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.

(Elhart, Langheinrich, Memarovic, & Heikkinen, 2014) identified several limitations in existing scheduling systems for public displays, particularly when dealing with interactive applications. They proposed a scheduling system that is able to schedule applications with arbitrary start time and durations. They define a notation that allows the description of the display environment, the application environment, and rules for display behaviours. The work presented in this paper addresses similar issues but at a different level. While (Elhart et al., 2014) were concerned with the high-level scheduling issues, at the display network system level, we are concerned with lower-level issues such as managing the resources of the individual display and application.

3 EXISTING PROBLEMS AND DESIGN GOALS

Previous work on interactive public display applications (Cardoso, 2014b; Elhart et al., 2014) has identified a number of shortcomings in existing public display systems. In this section, we present and extend the observed problems, and the associated design goals for the run-time life-cycle we propose in this paper.

3.1 Application Loading

Many interactive applications have noticeable loading times that designers usually address by

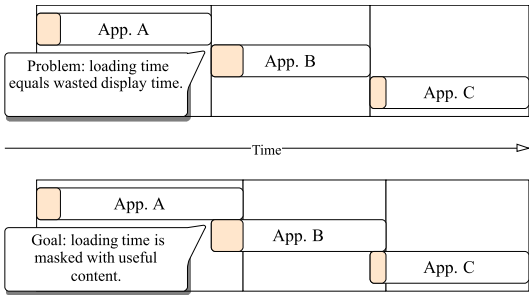

Figure 1: The problem with application loading times.

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 display resources and hinder the user experience: the time an application takes to load could have been used to display the previous content for a bit more time. This problem is illustrated in Figure 1.

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

3.2 Graceful Transitions

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 execution. 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. This problem is illustrated in Figure 2.

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

3.3 Abrupt Termination

Another issue we notice in interactive applications is the difficulty of running proper finishing/cleaning 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.

Our goal is to allow applications to terminate properly, giving them time to contact servers and save their state remotely or locally.

3.4 Pausing and Resuming

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 a notification 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. However, the most common approach is to unload the current application and then reload it again after the notification has ended. This problem is illustrated in Figure 3. Our goal is to support application pausing, and resuming. Applications should be able to quickly resume operation if they are interrupted by the system, without having to be completely loaded again.

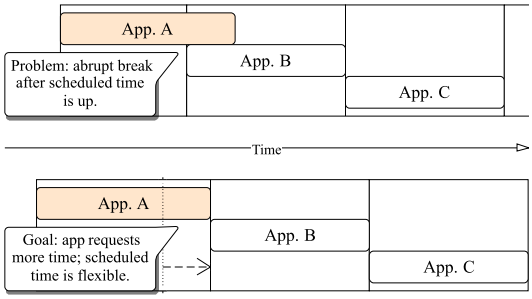

Figure 2: The problem with arbitrary duration.

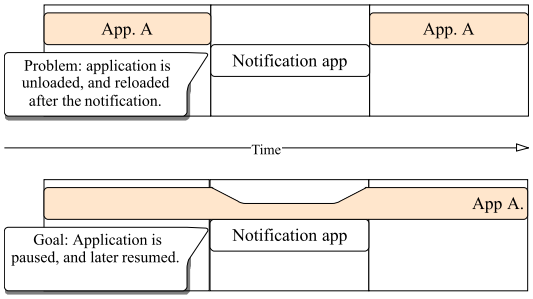

Figure 3: The problem with application interruptions.

3.5 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. This is illustrated in Figure 4. Obviously, this requires additional management policies on the display system to guarantee that applications do not misbehave and take over the display.

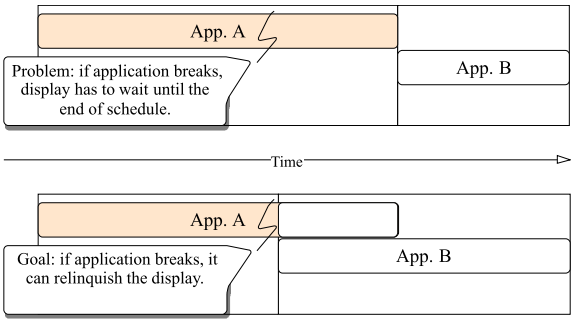

Figure 4: The problem with unforeseen termination.

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.

4 ANALYSIS OF EXISTING PLATFORMS

We have looked at various computing platforms in order to understand the existing approaches to run-time life-cycles. We then synthesized these models and adapted the result to take into account our design goals.

We have analysed Android, iOS, Windows Phone, Windows 8, Applets, and HTML/Javascript platforms. 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 and event callbacks. In all these platforms, primary memory is a central resource. When an application is “loaded” or “initialized”, this means that it is being loaded into primary memory. Conversely, when an application is “unloaded” or “destroyed”, this means that it is being unloaded from primary memory. Most application states are defined for when the application is loaded in memory. These states allow the application, and the system, to better manage their resources (memory, CPU, energy consumption, bandwidth, etc.) in an efficient manner, while still maintaining the responsiveness of the application, and system.

The main event callbacks associated with each platform are presented in Table 1 and described next.

Initializing refers to callback methods that are invoked only once by the system, when the application is initialized. Initialization callbacks are usually called by the system before the application is shown to the user, so that lengthy operations can be executed without disturbing the user experience. Typically, programmers should use these callbacks to instantiate user interface resources and other startup logic that happens only once in the lifetime of the application. On the Android platform, for example, the onCreate() is the only initialization callback and programmers are instructed to declare the user interface, which is usually defined in an XML file and thus must be parsed and converted to actual programming objects.

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. These callbacks may be invoked several times during the lifetime of the application in memory. In general these callbacks allow applications to start graphical animations, sounds, and other quick initializations, but different platforms provide different levels of granularity. For example, the Android platform provides three different start/resume callbacks giving a very fine-grained control to programmers: onStart() signals the transition from an invisible state to a visible state and is always called before onResume(). Together, they allow programmers to separate different initialization procedures to make sure that the application is displayed as fast as possible. Additionally, there is also the onRestart() callback, called before onStart(), that allows programmers to know if the application is starting for the first time, or restarting.

Table 1: Summary of analysed platforms

Callback categories	***Platforms***					
	Android	***iOS***	***Windows Phone***	***Windows 8***	***Applets***	***HTML***
Initializing	onCreate()	WillFinishLaunchingWithOptions() DidFinishLaunchingWithOptions()	Launching()	onLaunched()	Init()	Onload()
Starting/ Resuming	onStart() onResume() onRestart()	DidBecomeActive()	Activated()	Activating() Resuming()	Start()	Onpageshow() onfocus()
Pausing	onPause()	WillResignActive() WillEnterForeground()	Deactivated()	VisibilityChanged() Suspending()		Onpagehide() onblur()
Stopping/ Destroying	onStop() onDestroy()	DidEnterBackground() WillTerminate()	Close()		Stop() Destroy()	Onbeforeunload() onunload()

Pausing refers to callbacks that signal the application that it is being interrupted and is being taken from view, 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 if the system needs the resources.

5 RUN-TIME LIFE-CYCLE

Our model for a run-time life-cycle for public display applications is presented graphically in Figure 5, and described next.

onCreate() – This represents the application's entry point method and is called only once in the lifetime of the application 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 functions are ready before the onCreate() is called.

onLoad() – is called when the display system decides to give display time to the application. Before display time is actually assigned to the application, the system calls onLoad() and expects applications to reply with a **loaded()** call. At the onLoad() stage, applications should perform all necessary loading routines to ensure the application is ready to be displayed. The system will only consider the application ready to be displayed when it receives the loaded() response from the application. The onLoad() callback (in addition to the onResume() described next), address the problem of "Application Loading" described earlier. While the display system is waiting to receive the loaded() response from the application, it can display other useful content. This makes the use of splash screens and loading indicators unnecessary (at least as a way to mask initializations).

onResume() – 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.

onPauseRequest() – this callback signals the

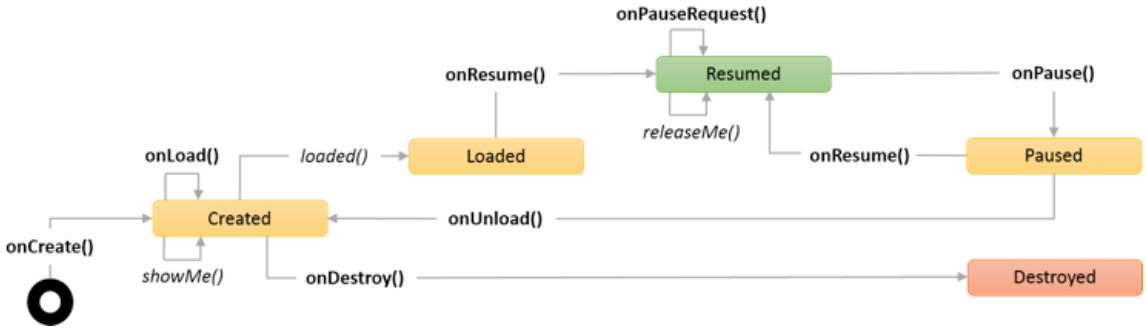

Figure 5: Proposed run-time life-cycle model for public display applications.

application that it will be taken off the display. In this stage applications can still notify the system about how much more time they need to finish gracefully. The system will honour 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. Applications should implement this callback and return a numeric value corresponding to the number of seconds they need to finish gracefully. This callback addresses the "Graceful Transitions" problem identified earlier. With this approach, applications can request extra display time, beyond the time originally defined by the display owner, to finish what they were doing in a way that causes the least disturbance to the user's experience. For example, a video player application can request an extra time that allows it to finish playing the current video.

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. Paused applications should make sure they are able to display useful content quickly when they are put back on the display. By allowing applications to be paused, instead of completely unloading them, we address the "Pausing and Resuming" problem described earlier.

onUnload() – the display system may decide to unload a paused application when it expects that the application will not be put back on the display soon. When the onUnload() callback is invoked, applications should unload any memory and CPU demanding resources, and keep only the minimum network connections required.

onDestroy() – is invoked just before the application is completely unloaded from memory. Applications should perform any finalization routines here, perhaps saving state to persistent storage either locally or remotely. The onUnload() and onDestroy() callbacks address the "Abrupt Termination" problem described earlier by explicitly providing a means for applications to perform finalization routines.

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. The showMe() and releaseMe() callbacks address the "Application-requested Loading and Unloading" problem described earlier.

6 PUBLIC DISPLAY APPLICATION SCHEDULER

We have implemented a first version of our model as a Google Chrome Extension where each application is assigned a browser tab. This allows any computer with the Google Chrome browser to become a public display driver, without the need for any further software.

Our implementation manages the life-cycle of each application, invoking the specified callbacks on the application code through message passing, and determining which tab should be displayed at any given time. Our system applies a priorities scheme to determine which applications can interrupt which applications.

In the next sections, we provide a more detailed description of the main concepts.

6.1 Applications

Our scheduler supports full-screen, web-based applications (i.e., only one application is displayed at a time, and it must be able to run on a browser). Our scheduler supports three types of applications:

Foreground applications correspond to traditional public display applications. These are applications that show relevant content whenever they are displayed.

Background applications are applications that are usually in the background, i.e., not showing any content. These applications can only show relevant content when external events occur. A notification application that alerts users for a given calendar event is an example of a background application.

Legacy applications are traditional web applications that do not implement our run-time life-cycle. Legacy applications are always foreground applications but they were not implemented specifically to follow our life-cycle model.

6.2 Scheduling

Our public display application scheduler supports traditional scheduling by allowing display owners to define a playlist of applications. This is done within an options page in the Chrome

Extension. Figure 6 shows the current interface to define the display's playlist.

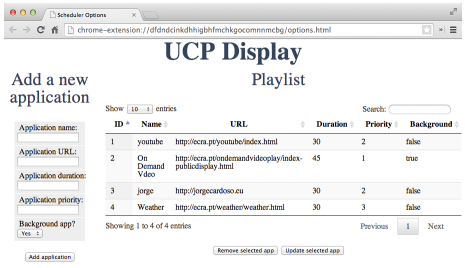

Figure 6: Interface to define the display's playlist.

At the moment, each display's playlist is defined in the display itself, but this could be easily changed so that a central service could manage the scheduling of several displays simultaneously. To define a playlist, display owners must enter the application's URL, how much display time the application should have, what priority it has, and whether the application is a foreground, or background applications (legacy applications are handled transparently from the display owners point of view).

When started, the system loads applications into independent browser tabs. Background applications are all loaded when the scheduler is started so that they can begin executing and possibly ask for display time whenever needed. Foreground applications are loaded only when they are schedule to be displayed for the first time (but are kept loaded until the system is shutdown, or when the system needs the resources to load other applications).

The system will go through all foreground applications in a round-robin fashion, and assign display time to each application according to the value defined by the display owner. However, the initial schedule defined by the display owner is only indicative and can change as result of:

1. The current application requests to be taken out of the display (for example because it lost communication with its own server or with a required third-party service). In this case, the next application will be displayed before its initial scheduled time.
2. Another application requests display time and either
    a. interrupts the current application, or
    b. is scheduled to display after.

6.3 Priorities

Priorities are enforced when an application requests display time. Different policies can be implemented, but currently we use a simple policy to determine if an application that requests display time should interrupt the current application or be schedule to display after the current application: if the requesting application has a higher priority than the current application, the current application will be paused, and the requesting application will be displayed. If the requesting application has a lower or equal priority, it will be schedule to display after the current application. If more than one application requests display time, the scheduler orders them by priority: higher priority applications will be displayed first.

6.4 Destroying

Our scheduler destroys applications when the system runs out of memory resources. If a new application is being loaded and not enough resources exist, we destroy the oldest application (the application that was displayed the longest ago). We currently define the resource limits based on the number of open browser tabs: our scheduler limits the total number of tabs and destroys applications when the limit is exceeded. (The Google Chrome API for inspecting the memory resources was not yet available at the time of the development.)

7 EVALUATION

We evaluated the implemented system from two perspectives. The first was an informal evaluation of the behaviour of the system that allowed us to get a better perception of the scheduling policies. The second was an evaluation of the system from an application programmer's perspective in order to understand the usability of the API of the system.

7.1 System's behaviour

To evaluate the system's behaviour, we developed five applications that were then configured to run on a public display using our scheduler:

- Random YouTube Video: a foreground application that plays a random YouTube video from a selected set of videos. Default duration: 40 seconds; Priority: 3 (higher value means lower priority).
- RSS News: a legacy application that displays RSS feed entries from a selected set of feeds. Default duration: 30 seconds; Priority: 3.
- Weather: a legacy application that displays the local weather. Default duration: 20 seconds; Priority: 3.

- Calendar Alerts: a background application that displays calendar alerts 15 minutes before the event takes place. Default duration: 30 seconds; Priority: 1.
- Video on Demand: a background application that plays a YouTube video. This application has a desktop backoffice that allows users to select a YouTube video and create a QR Code that will launch the video on the public display. Users can create the QR Code and distribute it physically so that anyone can launch that specific video on the display at any time. Default duration: 30; Priority: 2.

The "Random YouTube Video" was programmed to relinquish the display when the current video ended, and to ask for additional time if the current video took longer than the default 40 seconds assigned to the application. The "Calendar Alerts" application was configured with the highest priority to make sure it interrupted any application and was able to display the alert timely. The "Video on Demand" application was also configured with a high priority to make sure users did not have to wait much to see the video after they scanned the QR Code.

Although these applications were only setup in a public display in our laboratory, this already allowed us to perceive a few issues with the system:

Applications interrupted very near the end give the impression of error. We noticed that sometimes the Random YouTube Video application or the RSS News application would be interrupted very near to the end of the video, or the default application time. When these applications were resumed, they would appear only very briefly before giving place to the next scheduled application. In these cases, users had not enough time to even read the news post, and this could be perceived as a system error, or at least as a strange system behaviour by users.

Users loose context with applications interrupted for a very long time. When the two background applications requested display time, they could interrupt the current application for a considerable amount of time (30 seconds for the Calendar alert plus the duration of the video to be played by the Video on Demand application). When the interrupted application was resumed, a few minutes could have passed since it was interrupted and users would have lost the context of that application. This was most noticeable when the interrupted application was the Random YouTube Video. Users that were not present when the application was interrupted would begin to see the video from the middle.

Legacy applications continue to consume resources. Another issue we noticed with our system was that, although it supports legacy applications and is able to display them correctly, these applications continue to consume CPU and memory even when not being displayed. We noticed this issue particularly with the Weather application that uses a Flash animation to represent the current weather. Because the application does not implement our run-time life-cycle, the Flash objects used are constantly loaded. Even though the browser's plugin manages these resources so that tabs that are not visible do not use as much resources as visible tabs, this is still not an optimal management, particularly for an application that is only displayed for less than 15% of the overall default time for all applications.

7.2 Programmers evaluation

We conducted programming sessions with a total of 5 programmers with, at least, basic knowledge on Javascript. The participants were asked to develop a simple application or adapt an existing one using our model and framework. We sent documentation to participants the day before the session with the following information: brief introduction (context and motivation of the work), an image of the proposed run-time with an explanation of each state and callback, a template of an application that displays on the screen the name of the callback when called and finally the description of the task we proposed them to do.

On the session day, we started by clarifying the doubts of the participants, followed by a demonstration of our extension working with some of the developed applications. Before the participants started to develop their applications, a small period of time was used to hear and discuss application's ideas, making sure the applications to develop would take advantage of our system. We noted the most relevant doubts during the session and at the end we asked the programmer's opinions.

All programmers successfully created or adapted an application using our framework. We observed that all the programmers had some initial difficulties to understand the proposed life-cycle and all its specifications. We quickly concluded that the given documentation and example was insufficient to completely understand the life-cycle. At the end, all participants have stated that a more complete documentation would have eased the task.

One programmer suggested to divide onResume() on two different callbacks, in a similar way to the Android platform: one callback for the first time the application is put on the screen, and another for when the application is resumed from paused. For this developer this would make the code

easier to read and would give programmers an easy way to determine if the application was being put on the display for the first time.

Another programmer suggested that the releaseMe() callback could also be available from the paused state (currently it is only available from the resumed state). This would allow applications to unload themselves if an error occurs while paused.

Overall, however, all programmers stated that it was easy or very easy to create an application for our public display system.

8 CONCLUSIONS

We have presented a new run-time life-cycle 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, allows applications to transition and terminate gracefully, allows rapid pausing and resuming, and allows applications to request and relinquish display time.

We have implemented this model as a Google Chrome Extension where each application is assigned a browser tab. Our implementation manages the life-cycle of each application determining which tab should be displayed at any time. We support three types of applications: foreground, background, and legacy applications. Our system provides a priority mechanism that allows display owners to control which applications can be interrupted by which applications.

Our tests with this system have revealed some issues regarding the user experience when applications are paused very near their end, and when they are interrupted for a long time. The best approach to deal with these issues is still an open question that we plan to research in the future.

Our system is, to the best of our knowledge, the first to specifically address the problem of resource management within a multi-application public display system. We believe this line of research can result in more efficient public display systems that provide a better user experience. Our system is available as an open-source project at (Cardoso, 2014a).

ACKNOWLEDGEMENTS

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

A "work-in-progress" version of this work has been presented in the ICIW 2014 conference (Perpétua, Cardoso, & Carlos C. Oliveira, 2014).

REFERENCES

Cardoso, J. C. S. (2014a). A Google Chrome based public display application scheduler. Retrieved from https://code.google.com/p/public-display-scheduler/

Cardoso, J. C. S. (2014b). *An Interaction Abstraction Toolkit for Public Display Applications*. University of Minho. Retrieved from http://figshare.com/articles/An_interaction_abstraction_toolkit_for_public_display_applications/920152

Clinch, S., Davies, N., Friday, A., & Clinch, G. (2013). Yarely: a software player for open pervasive display networks. In *Proceedings of the 2nd ACM International Symposium on Pervasive Displays* (pp. 25–30). ACM. doi:10.1145/2491568.2491575

Davies, N., Langheinrich, M., Jose, R., & Schmidt, A. (2012). Open Display Networks: A Communications Medium for the 21st Century. *Computer*, *45*(5), 58–64. doi:10.1109/MC.2012.114

Elhart, I., Langheinrich, M., Memarovic, N., & Heikkinen, T. (2014). Scheduling Interactive and Concurrently Running Applications in Pervasive Display Networks. In *Proceedings of The International Symposium on Pervasive Displays - PerDis '14* (pp. 104–109). New York, New York, USA: ACM Press. doi:10.1145/2611009.2611039

Lindén, T., Heikkinen, T., Ojala, T., Kukka, H., & Jurmu, M. (2010). Web-based Framework for Spatiotemporal Screen Real Estate Management of Interactive Public Displays, 1277–1280.

Perpétua, A., Cardoso, J. C. S., & Carlos C. Oliveira. (2014). A Runtime Lifecycle for Interactive Public Display Applications. In *Proceedings of The Ninth International Conference on Internet and Web Applications and Services - ICIW 2014* (pp. 72–75). Paris, France: IARIA. Retrieved from http://www.thinkmind.org/index.php?view=article&articleid=iciw_2014_4_10_20083

Storz, O., Friday, A., & Davies, N. (2006). Supporting content scheduling on situated public displays. *Computers & Graphics*, *30*(5), 681–691. doi:DOI: 10.1016/j.cag.2006.07.002