OpenCable Specifications

OCAP Well-Behaved Applications

OC-SP-OCAP-WBA-I01-080722

ISSUED

Notice

This OpenCable specification is the result of a cooperative effort undertaken at the direction of Cable Television Laboratories, Inc. for the benefit of the cable industry and its customers. This document may contain references to other documents not owned or controlled by CableLabs. Use and understanding of this document may require access to such other documents. Designing, manufacturing, distributing, using, selling, or servicing products, or providing services, based on this document may require intellectual property licenses from third parties for technology referenced in this document.

Neither CableLabs nor any member company is responsible to any party for any liability of any nature whatsoever resulting from or arising out of use or reliance upon this document, or any document referenced herein. This document is furnished on an "AS IS" basis and neither CableLabs nor its members provides any representation or warranty, express or implied, regarding the accuracy, completeness, noninfringement, or fitness for a particular purpose of this document, or any document referenced herein.

© Copyright 2008 Cable Television Laboratories, Inc.
All rights reserved.

Document Status Sheet

Document Control Number:	OC-SP-OCAP-WBA-I01-080722			
Document Title:	OCAP Well-Behaved Applications			
Revision History:	I01 - 7/22/08			
Date:	July 22, 2008			
Status:	~~Work in Progress~~	~~Draft~~	Issued	~~Closed~~
Distribution Restrictions:	~~Author Only~~	~~CL/Member~~	~~CL/ Member/ Vendor~~	Public

Key to Document Status Codes

Work in Progress An incomplete document, designed to guide discussion and generate feedback that may include several alternative requirements for consideration.

Draft A document in specification format considered largely complete, but lacking review by Members and vendors. Drafts are susceptible to substantial change during the review process.

Issued A stable document, which has undergone rigorous member and vendor review and is suitable for product design and development, cross-vendor interoperability, and for certification testing.

Closed A static document, reviewed, tested, validated, and closed to further engineering change requests to the specification through CableLabs.

Trademarks:

CableLabs®, DOCSIS®, EuroDOCSIS™, eDOCSIS™, M-CMTS™, PacketCable™, EuroPacketCable™, PCMM™, CableHome®, CableOffice™, OpenCable™, OCAP™, CableCARD™, M-Card™, DCAS™, tru2way™, and CablePC™ are trademarks of Cable Television Laboratories, Inc.

Contents

1 SCOPE......1

1.1 Introduction and Overview......1
1.2 Purpose of document......1

2 REFERENCES......2

2.1 Normative References......2
2.2 Informative References......2
2.3 Reference Acquisition......2

3 TERMS AND DEFINITIONS......3

4 ABBREVIATIONS AND ACRONYMS......4

5 CONVENTIONS......5

5.1 Specification Language......5

6 THE WELL-BEHAVED APPLICATION......6

6.1 Xlet State Transitions......6
6.1.1 Xlet Entry Points......6
6.2 Input Focus Management......9
6.2.1 HScene Z-Ordering......9
6.2.2 Obtaining Focus......9
6.2.3 Focus Switching Behavior......10
6.2.4 Yielding Focus......10
6.2.5 Window Events......10
6.3 OCAP Event Management......11
6.3.1 Key Reservation Management......11
6.3.2 Reserving Access through AWT......12
6.3.3 OCAP Key Processing......12
6.3.4 Supported Key Sharing Models......13
6.4 Overlay Techniques......13
6.4.1 HScene Clipping......13
6.4.2 Repairing Overlays......15
6.4.3 HScene Stack Management......16
6.4.4 Stack Events......17
6.4.5 Blocking Behavior......17
6.5 Drawing in Application Threads......17
6.6 Screen Saving......17

Figures

FIGURE 1 - FULLY BLENDED SCENES 15
FIGURE 2 - FULLY BLENDED SCENES OVER VIDEO 15

Tables

TABLE 1 - KEY EVENTS AVAILABLE THROUGH AWT KEYLISTENER....................................13

1 SCOPE

1.1 Introduction and Overview

This document presents guidelines for developers of OCAP applications, and describes the properties of a "well-behaved" OCAP application.

OCAP supports the ability to have more than one application executing at a time. The platform defines tools and policies to facilitate the execution of multiple applications, but there are a number of things that applications can do to optimize the execution environment and make the user experience more seamless. This document describes dos and don'ts for applications to facilitate their execution in a cooperative environment.

1.2 Purpose of document

The purpose of this document is to describe a set of application conventions and application characteristics to aid their integration into a suite of concurrently executing applications.

2 REFERENCES

2.1 Normative References

In order to claim compliance with this specification, it is necessary to conform to the following standards and other works as indicated, in addition to the other requirements of this specification. Notwithstanding, intellectual property rights may be required to use or implement such normative references.

[OCAP] OpenCable Application Platform Specification, OCAP 1.0 Profile, OC-SP-OCAP1.0.2-080314, March 14, 2008, Cable Television Laboratories, Inc.

[MHP] DVB Multimedia Home Platform 1.0.3, DVB-MHP 1.0.3, ETSI TS 101 812 V1.3.1 (2003-06)

[MSM] OCAP Multiscreen Manager (MSM) Extension, OC-SP-OCAP-MSM-I01-071012, October 12, 2007, Cable Television Laboratories, Inc.

2.2 Informative References

None.

2.3 Reference Acquisition

- Cable Television Laboratories, Inc., 858 Coal Creek Circle, Louisville, CO 80027;
  Phone +1-303-661-9100; Fax +1-303-661-9199; http://www.cablelabs.com
- ETSI, www.etsi.org

3 TERMS AND DEFINITIONS

This specification does not define any new terms.

4 ABBREVIATIONS AND ACRONYMS

This specification does not define any new abbreviations.

5 CONVENTIONS

The following conventions are used in this document:

5.1 Specification Language

The following words are used throughout this document to define the significance of particular requirements:

SHALL	This word means that the item is an absolute requirement of this specification. Text that contains the term SHALL is separated into paragraphs and highlighted in grey.
SHALL NOT	This phrase means that the item is an absolute prohibition of this specification.
SHOULD	This word means that valid reasons may exist in particular circumstances to ignore this item, but the full implications should be understood and the case carefully weighed before choosing a different course.
SHOULD NOT	This phrase means that there may exist valid reasons in particular circumstances when the listed behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY	This word means that this item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because it enhances the product, for example; another vendor may omit the same item.

6 THE WELL-BEHAVED APPLICATION

This section defines specific requirements for well-behaved applications, as well as providing informative background text for the benefit of developers. An application must follow the specific requirements defined in this section for it to be considered "well-behaved".

6.1 Xlet State Transitions

The base class for all OCAP applications is javax.tv.xlet.Xlet. This section discusses the Xlet execution states and recommended behaviors for an application during each state transition.

An Xlet has four states: loaded, active, paused, and destroyed. See [MHP] Section 9.2.3.2 for a description of these states, related transitions, and entry points associated with each transition.

6.1.1 Xlet Entry Points

There are five entry points defined by the Xlet interface. These methods are called by the OCAP implementation based on the various events and conditions described within each section.

- A no-argument constructor
- initXlet()
- startXlet()
- pauseXlet()
- destroyXlet()

This section describes required behavior for well-behaved applications within these entry points.

Applications SHALL NOT intentionally block, including invoking Thread.sleep, during any of these methods, as this may delay the OCAP implementation from starting other processes or applications.

6.1.1.1 Constructor

When the OCAP implementation first loads an application to prepare it for execution, it calls the no-argument constructor for the base class of the application. This constructor is not expected to do anything, and need not be implemented by the application. Instead, all application initialization is expected to be handled by the initXlet() method.

The one exception to this convention is the Initial Monitor Application, which should implement the no-argument constructor, during which it should call OcapSystem.monitorConfiguringSignal().

6.1.1.2 initXlet

initXlet() is the initial entry point for an OCAP Application. It will only be called once, so an application need not check for re-entrance.

Upon the return of this method, the application is in the ***paused*** state. Typically an application is expected to release resources and remain quiescent while in this state; see [MHP] Sections 9.2.3.2, 9.3.3.3.2, and A.5.4.2. In this case, a call to startXlet() is usually made immediately after the return of initXlet(). Therefore, during initXlet(), an application MAY initialize any other classes or variables necessary for later execution of the application. Typical tasks may include:

- Initialization of any application variables;
- Requesting access to the Root Container for the application and setting its bounds;
- Adding the Xlet Component to the Root Container (but not making the Root Container visible);
- Creating DVB UserEventRepositories for all key groups that will be managed by the application (but not requesting exclusive access to those events or requesting AWT focus);
- Loading files necessary for the application to run. It is recommended to use background threads for any extended I/O operations at this point, returning from initXlet() as quickly as possible.

While it is possible for applications to receive AWT focus events or user input events while in the paused state, an application SHALL NOT request focus, install AWT or DVBEvent listeners, or make itself visible within initXlet().

6.1.1.3 startXlet

The startXlet() method is called in order to transition the application to the ***active*** state. The application transitions to the new state when startXlet() returns without an Exception.

Applications SHOULD perform the following operations within startXlet():

- Allocate all major blocks of memory;
- Start any background threads for loading data from the file system;
- Install all event listeners;
- Make the initial GUI components visible;
- Request AWT focus, if using AWT events;
- Otherwise prepare the application for execution.

startXlet() may be called more than once in an application's lifecycle. The MHP specification indicates that it will be called immediately after the successful return of initXlet() and also when returning from a ***paused*** state.

There are two reasons that an application may be transitioned to an ***active*** state. The first is that the application's home environment has become selected. The second reason is in response to a programmatic request to become active. An application may request its own transition to an active state from a paused state using XletContext.resumeRequest(). An application may be activated by a second application using appProxy.start() and appProxy.resume().

It is not guaranteed that the resume requests will be honored by the OCAP implementation, but this request typically results in an immediate call to startXlet().

Applications SHOULD take care to only re-initialize objects or data that were released while in the paused state. A simple solution is to keep a boolean flag that indicates whether startXlet() has already been called.

An application SHALL be prepared for multiple calls to startXlet(). An application SHALL reinitialize any and all components disabled by an intervening call to pauseXlet().

6.1.1.4 pauseXlet

The pauseXlet() method is called in order to transition the application to the ***paused*** state. The application transitions to the new state when pauseXlet() returns.

There are two reasons that an application may be transitioned to a ***paused*** state. The first is that the selected cable environment no longer supports the application's home environment. The second reason is in response to a

programmatic request to pause. An application may request its own transition to a paused state from an active state using XletContext.notifyPaused(). An application may be paused by a second application using appProxy.pause().

The implementation may unilaterally remove resources from an application in the paused state. See [OCAP] sections 11.2.2.3.19, 10.2.2.4, and Annex Y.

While applications are expected to minimize their resource use, they should keep themselves ready to become active again. These contradictory imperatives lead to a non-deterministic recommendation to give up any buffers or data elements which can be quickly restored. Those that require more lengthy operations or access to remote files should be maintained, if at all possible. An example of this may be an application that keeps a large "raw" data buffer and periodically uses that data to create a series of objects, based on the application context and state. Keeping the raw data is probably more important than the cache of objects which were contextually generated based on that data.

Applications SHOULD release all reserved DAVIC resources during pauseXlet().

Applications SHALL be prepared to lose reserved resources while in the paused state that were not released in pauseXlet().

Applications SHALL NOT be visible while paused, SHALL hide their UI, and SHALL NOT request AWT focus during pauseXlet().

6.1.1.5 destroyXlet

This entry point is called when the application is to be destroyed. destroyXlet() is called with an *unconditional* parameter that indicates whether this is a suggestion or a demand. If unconditional is set to true, the implementation will terminate the application. If unconditional is set to false, the application can delay its termination by returning with an XletStateChangeException.

Applications SHALL follow the directions for this state transition as described in [MHP] clause 11.7.1.2:

- Cause any threads that they have created to exit voluntarily;
- Stop, deallocate and close any JMF players that they have created;
- Stop and destroy any JavaTV service selection ServiceContext objects that they created;
- Release any other scarce resources that they created, e.g., NetworkInterfaceControllers if they do any tuning;
- Flush any images using the Image.flush() method;
- Xlets shall not cause any unnecessary delay in their Xlet.destroyXlet method;
- De-register any event listeners.

In addition to these tasks, an application SHALL terminate any active timers. Also, an application SHALL NOT sleep during destroyXlet().

An application MAY save any state information in persistent storage in the event that it is re-started in the future and wishes to resume from where it left off.

An application can terminate itself by calling XletContext.notifyDestroyed(). In this case, the implementation will NOT call the destroyXlet() entry point, so the application SHOULD perform all of these functions before calling notifyDestroyed().

6.2 Input Focus Management

Focus is the ability to receive Java AWT KeyEvents, specifically key presses on a remote control or extended keyboard (if one is present). An application may receive KeyEvents via the KeyListener interface when it has focus.

Detailed discussions of input focus management may be found in [OCAP] section 13.3.6.2 and Annex K. Generally speaking, an application is considered to have focus when one of the AWT Components created by that application has focus. This section addresses focus as it moves from one application to another and does not address focus moving from one Component in the application to another.

An OCAP application is not required to request AWT focus. All input events can be read through the DVB EventManager. Using the AWT InputEvent mechanism is a convenience for application portability. For those applications that do use AWT InputEvents, the OCAP platform ties AWT Focus management to the HScene User Interface Management to create a unified user experience in the absence of a full windowing environment.

6.2.1 HScene Z-Ordering

The HScene Manager keeps a Z-ordered stack of HScenes that are active and visible. The HScene that is closest to the user is considered the "front most" HScene, or the "top" of the stack. The front-most HScene obscures or blends with HScenes "behind" it (or below it on the stack). For more information on the rules for blending HScenes, see Section 6.4. In OCAP, the OcapScene interface defines a number of methods for adding the HScene to the front or back of this stack, and for being notified when the HScene is moved to the front or loses its front-most status.

In terms of focus management, only the front most HScene in the Z-ordered stack is eligible to receive AWT Focus. If it does not request focus, then no application has focus. When the HScene at the top of the stack changes, focus will be removed from the HScene previously at the top of the stack. If the new HScene at the top of the stack has previously requested focus, focus will automatically again be granted to the root container for that HScene.

6.2.2 Obtaining Focus

As discussed above, the OCAP implementation maintains a Z-ordered stack of HScenes that are "eligible" to receive focus.

In order to receive focus, four conditions have to be met:

- The HScene is active (see HScene.setActive());
- The HScene is visible (see Component.setVisible());
- The HScene (or a Component which has been added to the HScene) has requested focus at least once. (see Component.requestFocus());
- The HScene is the front most visible HScene in the stack.

HScenes are, by default, active, so there is no need to take any action in this regard, unless the HScene is specifically made inactive.

Visibility is related to the HScene alone, and not to any Components attached to that HScene. Therefore, it is possible for the HScene to be "visible" but to not have any visible UI on the display.

In terms of requesting focus, once an application calls requestFocus() for any Component connected to the HScene, the HScene is considered to have requested focus for all time. This is true, even if that initial Component is removed from the HScene and/or destroyed.

Although an application is not required to be in an ***active*** state to request focus, when an application is put into a ***paused*** state, it will lose focus. The HScene remains eligible to receive focus, however, so a subsequent call to

requestFocus() during pauseXlet() will return without errors, the HScene will be made the focused HScene, and the application will again be eligible to receive KeyEvents. A WBA SHALL NOT request focus while in the ***paused*** state. See Section 6.1.1.4.

6.2.3 Focus Switching Behavior

An application uses the FocusListener interface to be notified of focus transitions. The FocusListener interface defines two methods that are called by the OCAP implementation on focus transitions: focusGained() and focusLost().

A typical example of this follows:

```
public class MyApplication extends Component
      implements Xlet, FocusListener, KeyListener
{
      …
      public void focusGained(FocusEvent ev)
      {
            // expect AWT input events
            // allow drawing outside of the paint() method
      }
      public void focusLost(FocusEvent ev)
      {
            // remove any highlights and don't expect AWT input events
            // disallow drawing outside of the paint() method
      }
}
```

Applications that display a UI without the presence of a mouse or other pointing device generally highlight the focused Component by changing the color of the Component. This gives the user a visual clue as to what will happen when they press "Select". As the user presses the Up/Down/Left/Right directional keys, the Component that is highlighted changes based on the layout of any interactive elements on the display.

Applications that display a UI SHALL implement either the FocusListener or the WindowListener interface to determine when it loses focus or is no longer the top HScene on the stack and SHALL remove the highlight from any display elements when either of these conditions occurs.

If application A has AWT focus and application B subsequently becomes the focused application, the focusLost() method in application A will be called, and the focusGained() method in application B will be called. If application B terminates or makes its root Container invisible or inactive, the focusGained() method in application A will automatically be called.

6.2.4 Yielding Focus

OCAP extends AWT and MHP focus management rules by also giving applications an opportunity to yield focus to other applications that may have lower priority requests for moving into focus. This is especially useful for applications such as the MSO navigator application, which may be on the "top of the stack" but essentially be transparent to video most of the time. Again, the OcapScene interface is used to conditionally yield focus to applications with deferred requests queued up, without otherwise relinquishing it to other applications on the stack.

6.2.5 Window Events

The WindowEvents WINDOW_ACTIVATED and WINDOW_DEACTIVATED are available to an application through the WindowListener interface. They are similar to the FOCUS_GAINED and FOCUS_LOST FocusEvents. In OCAP, WindowEvents are tied to the HScene Z-ordering rules, just as with AWT FocusEvents. There are, however, a few differences in these events. WindowEvents go to the top-level scene that contains (or is) the

component that gained or lost focus, whereas FocusEvents go to the component within that HScene that gained or lost focus. Also, WindowEvents are not generated when focus moves within an application (or at least, within the given HScene).

6.3 OCAP Event Management

A second mechanism for receiving input events is through the org.ocap.event.EventManager. This mechanism allows an application to receive key input events regardless of focus. A fundamental difference between the two mechanisms is that with AWT an application will receive ALL input events without specifically requesting notification of individual key presses. With the OCAP EventManager, an application specifically registers interest in a list of keys and will only receive notification when those specific keys are pressed, regardless of its focus status.

The OCAP EventManager allows these requests to be made either exclusively or non-exclusively, which determines whether any other applications will also be provided with the same key event. [OCAP] Annex K describes the decision tree that an OCAP implementation uses to determine notifications.

6.3.1 Key Reservation Management

Within the context of the OCAP EventManager, an input event is called a UserEvent. To receive UserEvents, an application uses the UserEventListener interface.

When registering interest in specific key input events through the OCAP EventManager, an application creates one or more UserEventRepositories and then registers the listener and repository, for example:

EventManager.addUserEventListener(UserEventListener l, UserEventRepository r)

Using this form of the addUserEventListener() method creates a "non-exclusive" reservation. Effectively, this means that more than one application may receive notification of this key press event.

To negotiate exclusive access to a set of input keys, an application uses the org.davic.resource.ResourceClient interface. This interface defines a set of methods that applications use to negotiate access to scarce resources - in this case, remote control keys. To request exclusive access to a list of key press events, a ResourceClient is also passed to the addUserEventListener() call:

EventManager.addUserEventListener(UserEventListener l, ResourceClient rc, UserEventRepository r).

A simplified version of the example in [OCAP] Annex K follows:

```
public class MyApplication extends Component
           implements Xlet, UserEventListener, ResourceClient, FocusListener,
                       KeyListener
{
          …
          public void notifyRelease(ResourceProxy rp) {
            // lost access to reserved keys
          }
          public void release(ResourceProxy rp) {
            // about to lose access to reserved keys
          }
          public boolean requestRelease(ResourceProxy rp, Object obj) {
            // request to release reserved keys
            // either do it and return true, or return false (denied)
          }
```

```
        public void userEventReceived(UserEvent ev) {
          // UserEvent received
        }
}
```

The ResourceProxy argument passed to notifyRelease(), release(), and requestRelease() may be cast as a RepositoryDescriptor, which is the superclass for a UserEventRepository. The RepositoryDescriptor allows the application to query the name of the UserEventRepository in contention. While it is possible that the RepositoryDescriptor passed to ResourceClient may also be an instance of a UserEventRepository, there is no guarantee that it is the same UserEventRepository that was passed to addUserEventListener().

NOTE: The rules for contention management are slightly different with UserEvents than with other DAVIC resources. Contention is not managed by a monitor application; it is purely up to the implementation to resolve contention between two applications that request the same resource. If an application does not give up an exclusive reservation, the resource is granted to the application with the highest priority.

6.3.2 Reserving Access through AWT

The method addExclusiveAccessToAWTEvent() provided by the OCAP EventManager, also allows an application to reserve exclusive access to a key event. However, this method directs the OCAP implementation to deliver that event through the AWT KeyListener interface, instead of the UserEventListener interface.

This method takes a ResourceClient and a UserEventRepository as parameters and behaves in most ways like addUserEventListener(), with the exception of how the event is delivered to the application.

As with other events delivered through the AWT interface, in order to receive input events through this mechanism, the application must be in focus. If the application that was granted exclusive AWT access to a key press is not in focus, no other application will be notified of this key press event.

6.3.3 OCAP Key Processing

OCAP platforms may support input events that are not defined by [OCAP] for use by non-OCAP manufacturers' applications.

Such keys may or may not be passed to the OCAP environment; therefore, OCAP applications SHALL NOT expect to receive keys that are not defined in [OCAP] section 25.

User key events are processed by the OCAP implementation as follows:

- If the key has been reserved exclusively using addExclusiveAccessToAWTEvent() and the reserving application is in focus, the key event is passed to that application and to no other;
- If the key has been reserved exclusively using addExclusiveAccessToAWTEvent() and the reserving application is not in focus, the key event is ignored and not passed to any application. This behavior is prohibited. See Section 6.3.2;
- If the key has been reserved exclusively by an application using the DVB Event Manager, the key event is passed to the reserving application and to no other;
- If the key has been reserved non-exclusively using the DVB Event Manager by one or more applications, the key event is passed to each of the requesting applications;
- If the key has not been reserved exclusively by any application and an application is currently in focus, the key event is passed to the application in focus.

6.3.4 Supported Key Sharing Models

MSO-deployed applications are expected to reserve a number of key press events that will not be available to third-party applications at any time. A smaller subset of keys will be available to all applications, and may be received using either the AWT or DVB Event Manager interface.

- Section 6.3.4.1 defines the ***Focus Keys***. These key press events will be available to any application with AWT Focus. While it is possible that some additional key press events will be received by an application in some cable networks or on some particular set-tops, an application cannot rely on the receipt of these events on all set-tops or in other networks.
- All other key press events will be available only through the DVB Event Manager. Applications may request access to those key press events, but there is no guarantee that the MSO applications will release them. An application should be designed to function properly with only the Focus Keys. An application SHALL reserve other keys only while in focus and SHALL release all such keys when focus is lost or a release request is received.

6.3.4.1 Focus Keys

Applications SHALL only reserve the Focus Keys events while they are in focus. When an application detects that it no longer has focus, it SHALL release any reservations on these events.

This is the complete list of events that will be available through the AWT KeyListener interface.

Table 1 - Key Events available through AWT KeyListener

Key Event
VK_UP
VK_DOWN
VK_LEFT
VK_RIGHT
VK_ENTER
VK_INFO
VK_EXIT
VK_COLORED_KEY_0
VK_COLORED_KEY_1
VK_COLORED_KEY_2
VK_COLORED_KEY_3
VK_PAGE_UP
VK_PAGE_DOWN

6.4 Overlay Techniques

6.4.1 HScene Clipping

OCAP defines clear mechanisms for an application to control alpha-blending between the graphics that it draws into its own HScene, as well as between the graphics plane (OSD) and the video plane. Furthermore, OCAP gives

applications the ability to control blending between each other, when more than one application is visible on the screen at a time.

Inter-HScene blending rules are consistent with intra-HScene blending rules. The display is created from a composite of all visible HScenes. When each application renders its HScene, it can define each pixel to be:

- unmodified
- modified
- replaced

The pixel is unmodified when the HScene background mode is set to BACKGROUND_NO_FILL *and* the application does not explicitly render that pixel.

The pixel is modified when the HScene background mode is set to BACKGROUND_NO_FILL *and* the application renders that pixel in SRC_OVER mode.

The pixel is replaced when the HScene background mode is set to BACKGROUND_FILL *or* a background image is defined that covers that pixel *or* when the application renders that pixel in SRC mode.

The following figures show two applications' HScenes blended together with SRC_OVER (first over a white background, then over video).

Note that:

- Not only is the background scene visible through the non-rendered areas of the foreground scene, but it is also visible through the translucent blue pixels of the foreground scene.
- Where the translucent pixels of the foreground application mix with the translucent pixels of the background, video can be seen.
- Where the translucent pixels of the foreground application mix with the opaque pixels of the background, video cannot be seen.

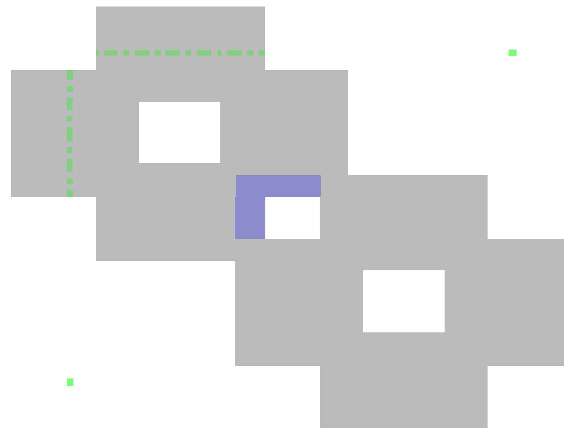

Figure 1 - Fully Blended scenes

Figure 2 - Fully Blended scenes over video

6.4.2 Repairing Overlays

When an application makes its HScene invisible or inactive, the OCAP implementation will recomposite the display by calling the paint() method of the root component of each visible HScene. The Graphics object passed to paint()

may optionally be clipped to the portion of the display that was obscured, but would otherwise be clipped to the size of the HScene.

Alternatively, it is possible that the OCAP implementation will keep frame buffers for each visible HScene, and reconstruct the relevant portion of the display on its own. In this case, paint() would not be called, and the application would not otherwise know that its graphics were restored or even needed to be restored.

As above, the OCAP implementation will recomposite the display from all visible HScenes, from bottom to top, clipping on the bounds of the HScene.

6.4.3 HScene Stack Management

OCAP manages all visible HScenes in a stack, which defines their z-ordering when compositing the final output of the graphics plane. There are 2 methods for making an HScene visible:

- HScene.setVisible(true)
- HScene.show()

Each of these calls set the HScene's "visible" flag to true and cause the HScene to be promoted to the top of the stack.

OCAP extends HScene with org.ocap.ui.OcapScene. The OcapScene interface provides methods for:

- Requesting that a scene specifically move to the front or the back of the z-order stack for the containing graphics plane;
- Conditionally yielding focus;
- Conditionally requesting focus;
- Managing the "top of the stack" as a scarce resource.

The methods OcapScene.showToFront() and OcapScene.showToBack() allow applications a greater degree of control over placement of their HScenes within the HScene stack.

The front-most position in the z-order stack for the containing graphics plane is considered a scarce resource. The OcapScene serves as the proxy for this scarce resource. The scarce resource may be explicitly requested using the showToFront() and showToBack() methods. Existing methods that imply a change of position in the z-order (e.g., HScene.show() or HScene.setVisible(true)) also imply an ownership request of the scarce resource. Existing methods that hide or dispose of a scene imply a release of the scarce resource.

The front most position in the z-order stack is treated as a scarce resource for the following reasons:

- It implies exclusivity for user interaction:
  - Display exclusivity is given to the front-most scene,
  - Only the front-most scene may have focus.

NOTE: As there may be multiple graphics planes, each graphics plane will have its own logical resource for the front-most position. Each owner of these resources has equal access to AWT focus and the display. The monitor application is responsible for managing inter-plane interactions via APIs such as [MSM] or the HSceneChangeRequestHandler.

6.4.4 Stack Events

Applications can obtain a ResourceServer from an OcapScene, through which they can register a ResourceStatusEventListener. This listener will provide notification when the HScene is "in front" or has lost its front most position in the z-ordered stack of HScenes.

Alternatively, the rules that tie AWT focus to the top HScene on the stack allow an application to more deterministically use the FOCUS_GAINED and FOCUS_LOST events to determine both when the application loses focus and when its HScene is no longer on the top of the stack.

6.4.5 Blocking Behavior

Applications with MonitorAppPermission "handler.resource" may also be notified whenever the z-order is "about to change" allowing some degree of veto power over an upcoming transition. This allows a monitor application to temporarily block other applications while performing specific activities.

6.5 Drawing in Application Threads

Although most graphics drawing is done when the OCAP implementation calls an application's Component.paint() method, applications also have the choice of drawing on a separate animation thread. This technique is referred to as active rendering.

While this technique is essential for animations, several rules apply to those application doing active rendering, to prevent those actions from disrupting AWT's control of the other applications being rendered:

- An application SHALL NOT perform any active rendering unless that application has the front-most HScene on the stack of visible HScenes.
- Even when doing active rendering, an application SHALL paint its entire display (subject to clipping) when paint() is invoked.
- When doing active rendering, the application SHALL be responsible for all display updates, including clearing or erasing of previous contents, if necessary.
- Active rendering is not expected to play well with inter-scene blending. An application SHOULD disable inter-scene blending via HScene.setBackgroundMode() when doing active rendering.

6.6 Screen Saving

Pixel burn-in is a condition caused by displaying a static image on a television over an extended period of time. Normally, this is not a problem, as video typically changes virtually every pixel quite regularly. With static graphics or menus, however, this is not the case. Computers solve this by using a "screen saver", which draws random graphics or displays random pictures across the display periodically.

In order to avoid pixel burn-in, applications that present a UI SHALL NOT present a static graphic for more than 10 minutes at a time.