Creating Web Content for TV

Please note that parts of this article refers to older Presto-based versions of the Vewd products (Vewd Core3.x). These versions are still live in a lot of devices that are active in all markets, but most devices shipped since 2015 are based on the newer Blink-based Vewd product.

Update history:

- 15 September 2011: change from Opera CDK to Opera TV Emulator
- 20 December 2011: clarify `outline` handling for `:focus` styles
- 11 April 2012: clarify lack of support for `media="tv"` type
- 25 September 2012: include note about `outline:none` support in Vewd Core 3.4
- 22 January 2013: clarification on lack of support for `:active` style

Introduction

This guide is aimed at web developers wishing to optimize their web sites and applications for better compatibility with web-enabled/connected televisions — with a particular emphasis on the Vewd Core and its functionality.

The Vewd Core and Vewd TV Emulator

The Vewd for Devices Software Development Kit (SDK) is aimed at device manufacturers wishing to provide a customized browser and widget manager for use on their devices. It is compatible with pretty much any device that could conveniently require web access, with existing implementations ranging from web-enabled TV sets and set-top boxes to tablet PCs and connected picture frames.

The SDK provides building blocks, modules and example implementations. Manufacturers who choose the Vewd Core then do further integration work to hook the browser (and other components not covered in this guide, such as the widget manager) into their specific devices. As such, the SDK itself is not publicly available for download.

However, developers wanting to more easily test and debug their sites for delivery on web-enabled devices can use the Vewd TV Emulator. Built on the same core technology as the SDK, the emulator comes as a VirtualBox image which runs on Windows, Linux and Mac OS X. It allows for a more accurate test environment that matches the capabilities of the SDK. In addition, the Vewd TV Emulator allows for remote debugging with Opera Dragonfly — a feature that is normally turned off on retail TVs and devices.

Download the Vewd TV Emulator.

It's worth noting that, even though the emulator can facilitate testing and debugging, developers should — if possible — still test their content on actual devices, as certain features and specific constraints (such as the speed of the device's processor and available memory) can vary from device to device.

Standards support on web-enabled TVs

When building web sites and applications accessed through a web-enabled TVs regular web browser, developers can take full advantage of the web standards supported by the browser on the device. For Opera's suite of products (desktop, mobile and devices), the specific features that are supported depend on the Presto engine version running at the heart of each browser. At the time of writing, the Vewd Core deliveries are based on Presto 2.6 — the same engine found in the current desktop and mobile releases. This allows developers to take advantage of some of the latest standards:

- HTML5: `<canvas>`, `<audio>` and `<video>`, expanded forms support (although certain features, such as the new datetime types, are implemented, but not currently optimised for keyboard/spatial navigation access)
- CSS3: backgrounds and borders, Web Fonts, Media Queries, transitions and 2D transforms

For more details, see the list of web specifications supported in Opera. Note that certain features (such as the Geolocation API and Offline Web Applications), though present in the specific Presto version, may still be disabled on certain platforms running the Vewd Core. In addition, web-enabled TVs and devices already in use will have older versions of the browser shipped as part of their firmware. Since end users can't easily update their browser, unless it's part of a firmware update provided by the device manufacturers, it is safe to assume that most current of these devices will be running an older browser version, and many of the latest features will not be present. Developers should employ techniques such as JavaScript feature/object detection to determine the capabilities of the user's browser.

Developers can find which Presto version is present in a particular Opera browser by checking the user agent identification string in the internal `opera:about` information page. In the case of the Vewd Core, the string is currently:

```
Opera/9.80 (Windows NT 6.0; U; en-GB) **Presto/2.6**.33 Version/10.61
```

Designing for context and form factor

Although actual usage may vary, there are some general considerations with regards to context when talking about web content on TV.

Consumption of content (including web content) on TV, because of its traditional setting in the living room, is often a shared/social experience.

Primary activity often revolves around quick information look-up (for instance, cast and crew details for a particular movie, weather reports, TV listings) and quick access to services. Web content for TV should therefore be optimised — in terms of overall presentation, navigation and functionality — and task-focused, giving quick and clear access to all relevant features and information.

Adaptive layouts

TVs usually run at the following resolutions:

- Standard-definition television (SDTV): 720×480 (NTSC) and 720×576 (PAL), splits the display into two separate interlaced fields (243 and 288 lines, respectively).
- Enhanced-definition television (EDTV): 720×480 (NTSC) and 720×576 (PAL), delivered as single progressive scan images.
- High-definition television (HDTV): 1280×720 (also referred to as "HD Ready" or 720p, using progressive scan) and 1920×1080 ("Full HD", in either 1080i interlaced or 1080p progressive scan variants).

Most modern web-enabled TVs support 1280×720 as a minimum resolution. On TVs capable of displaying the full HD resolution of 1920×1080, 720p content is usually upscaled (although this depends on the individual device). Set-top boxes and similar web-enabled devices may be connected to standard-definition or enhanced-definition TV sets — in these situations they will usually downscale and letterbox their virtual resolution to fit the physical resolution of the TV screen.

Regardless of what type of TV set they're connected to, some web-enabled devices may run at a variety of virtual resolutions — as an example, the Nintendo Wii Internet Channel (which uses a custom version of the Opera desktop browser) has a virtual width of 800 pixels, with the height varying based on the type of TV (4:3 or 16:9 aspect ratio) and user settings.

Developers targeting only web-enabled TV devices should opt to design for 1280×720 as a minimum, or provide separate versions of their content optimised for 1280×720 and 1920×1080.

If content should also be displayed on standard-definition and advanced-definition TVs, developers could potentially create an additional version optimised for this resolution as well.

However, a more sustainable and maintainable solution would be to ensure that content — and specifically the screen layout — is built with enough flexibility to adapt to a variety of screen resolutions.

There are a few technologies that can be employed for these types of adaptive layouts:

Fluid layouts

Using percentage-based values to define the width and height of layout containers. This can further be combined with `max-width`, `min-width`, `max-height` and `min-height` in pixel values, to provide some upper and lower limits to the fluidity of the containers.

CSS2.1 Media Types

CSS2.1 allows for the definition of styling blocks that only apply to specific classes of device. This technique is most commonly used to provide separate styling for screen and print, negating the need for creating separate printer-friendly pages. CSS2.1 also allows the definition of styles specifically for TVs.

Whether or not a web-enabled device identifies itself as a TV depends on how its browser has been set up and integrated by the manufacturer, so developers should only use this method if they're sure their target platform has this setting enabled. For reasons of compatibility with the majority of web content "in the wild" which assumes a `media="screen"` device, practically all current-generation devices ignore `media="tv"`.

CSS2.1 Media Types can be specified when linking to separate external stylesheets:

```
<link rel="stylesheet" media="screen" href="…">
<link rel="stylesheet" media="tv" href="…">
```

or within a single stylesheet, by defining separate `@media` sections:

```
@media screen { /* styles for normal on-screen presentation */ }
@media tv { /* tv specific styles */ }
```

CSS3 Media Queries

While CSS2.1 Media Types classify devices into very broad categories (`screen`, `print`, `tv`, etc.), and rely on devices actually identifying themselves as such, CSS3 extends this concept by allowing developers to define specific conditions under which styles should be applied, providing far more granular control over how a layout is to be rendered under different conditions. For example, it is possible to create layouts that change depending on various factors, including the device's screen width, screen height, resolution and color depth.

CSS3 Media Queries, just like CSS2.1 Media Types, can be defined as part of the `<link>` elements:

```
<link rel="stylesheet" media="screen and (min-width: 1920px)" href="…">
<link rel="stylesheet" media="screen and (min-width: 1280px) and (min-width: 1920px)" href="…">
<link rel="stylesheet" media="screen and (max-width: 1280px)" href="…">
```

`or using @media sections in the stylesheet:`

```
@media screen and (min-width: 1920px) { /* Full-HD styles */ }
@media screen and (min-width: 1280px) and (max-width: 1920px) { /* HD-Ready styles */ }
@media screen and (max-width: 1280px) { /* smaller than HD-Ready styles * }
```

Safe areas

Traditionally, TV sets were not able to display the full width and height of their nominal resolution, resulting in content positioned in the edges of the screen to be cut off or not displayed at all. Although this problem has been mitigated or – on some high end devices – resolved on modern digital TV sets, developers are still advised to only place important content and controls in the "safe area", leaving a margin of approximately 5% along each edge of the viewport.

Single-window browsing

Although some devices will include the ability for browsers to open separate tabs, together with an interface for end users to switch between them, the large majority of web-enabled TVs will only run in single-window mode. Any calls to open a new window (be it through JavaScript, or simpler techniques such as providing a `target` attribute on links) will only affect the current window. Developers need to be particularly aware of this if their sites and applications currently use separate windows to provide contextual help or in more complex situations (such as web-based email clients) where child windows use JavaScript to communicate back to their parent windows in order to keep state or provide the bulk of commonly-used functions. In these cases, web applications will need to be reworked to only use a single window.

Scrolling

Ideally, developers should aim to present all content contained within their TV-optimised sites and applications on the screen, without the need for the user to scroll. If this is not possible, content should at least fit the width of the screen, requiring the user to just scroll vertically.

Using iframes or containers styled with `overflow: auto;` will work as expected in the Vewd Core, and users will be able to scroll these — as well as the entire page, if needed — using the default spatial navigation method of their web-enabled device. However, it may be more user-friendly to create customised controls and interface elements (such as content carousels) instead of relying on the browser's default scrollbar mechanism.

Use of text

Although TVs run at resolutions that are comparable to desktop and notebook devices, users are traditionally positioned further away from the screen (typically a few meters). It's important to keep this in mind when deciding on the size of text and graphical elements.

To increase the readability of text on a TV screen, developers should:

- Choose an appropriately large font size. The exact dimensions will vary depending on the specific typeface used, the physical size of the screen and the virtual resolution the device runs under, but a general minimum size of around 22px is advised
- Keep overall line length to around 10 words or less
- Use generous leading/line-spacing

The number of fonts available on devices is usually limited — much more so than on desktop computers. Typically, only widely available Linux fonts such as Bitstream Vera (serif, sans-serif and monospaced) are present. End users are not able to install any fonts beyond those that came pre-installed with their device. However, the Vewd Core browser offers support for CSS3 web fonts (including fonts served through third-party services like Typekit or the Google Font API).

Graphical elements

For graphical elements, it is best to make the design bold and “chunky”, particularly for interface elements such as buttons. Overly fine detail such as single pixel borders should be avoided, as these can be particularly problematic on devices running on interlaced resolutions (such as SDTV and 1080i). This does not mean that subtlety and texture should be avoided altogether, but distinguishing features need to be clearly visible even when viewed at a distance.

When combining graphical elements with adaptive layout techniques, it may be useful to employ Scalable Vector Graphics (SVG), which can be resized and transformed without distortion or pixelation. However, complex SVG elements (involving a large number of vectors and layered, semi-transparent sections) do require more processing power to calculate and render, so their usefulness needs to be balanced against their performance impact.

Colour

Certain colour combinations can be problematic when displayed on a TV screen. Even on high-end devices pure white, orange and red, as well as extreme contrasts, can lead to distortion and interference of the image and should be avoided.

In addition, the dynamic range of TV screens may not be the same as traditional computer monitors. Subtle gradients and low contrast colours may result in colour bands or indistinguishable combinations.

User interfaces and controls

TVs are “low interaction” devices. When designing interfaces, developers should ensure that they are clear and uncluttered, and that controls are pared down to the minimum necessary.

User interaction with a web-enabled TV is usually performed via a remote control. In addition to the regular number/channel keys, these remote controls feature directional/cursor buttons, OK/confirmation buttons, and a variety of special feature keys.

Devices will also provide users with a way to enter regular text — for instance, to fill in username/password fields in a login dialog or to enter keywords in a search field — by means of an on-screen keyboard or multi-tap interface. This is similar to traditional mobile phones, where a series of characters is mapped to a single key — ABC on the 1 key, DEF on the 2 key, and so on. However, these text entry methods are often slow and cumbersome for users. Applications should avoid or minimise the need for this kind of text entry, opting instead for alternatives such as selectable drop-downs, menus, checkboxes or custom-built scrollable interfaces.

Some devices may offer the option of providing an on-screen pointer, which is moved with the cursor keys or motion sensors in the remote control, as is the case on the Nintendo Wii. On-screen controls should generally be large, with ample padding, to provide a large hit area that facilitates user activation without the need for too much accuracy in moving the pointer.

Spatial navigation

As is the case on Opera desktop, the Vewd Core uses spatial navigation. While on the desktop this mode requires the use of SHIFT+cursor keys, on TV spatial navigation is mapped directly to the directional keys on the remote.

The most common interaction method is functionally equivalent to keyboard access on desktop/notebook devices. Therefore, designers should ensure that their content is keyboard accessible.

Traditionally, only certain types of element — links, form elements, image map areas, etc. — are keyboard focusable. It is good practice (from an accessibility point of view) to use these elements for controls, rather than simply attaching JavaScript behaviours to other/generic elements, as these are then usually not focusable and usable via the keyboard alone.

In addition, Opera’s spatial navigation mode features built-in heuristics that allow the focus to also be placed on any element that has a `click` or `mouseover` JavaScript handler, either in an `onclick` or `onmouseover` attribute, or hooked into the element via the `addEventListener` method. In addition, spatial navigation also allows elements with a `tabindex` attribute (with a value of 0 or above) to receive focus.

Note that for these non-traditional elements, spatial navigation mimics mouse behaviour, meaning that only `mouseover` and `mouseout` events are automatically fired when moving to/from these elements. `focus` and `blur` events are still only fired on elements that can receive traditional keyboard focus.

Although spatial navigation works “out-of-the-box” in most situations, developers can take further control over the order in which elements receive focus using the CSS3 Basic User Interface specification for directional focus navigation. See our separate article on Tweaking spatial navigation for TV browsing for more details.

Indicating focus

By default, Vewd Core will place an outline around the element that currently has focus. If the design of the specific content does require it, though, this standard outline can be partially suppressed.

The appearance of this outline can vary between devices, as manufacturers can re-skin the default styles applied to the focus indicators and the various elements of a web page. Developers should therefore test their sites on a variety of devices, or ensure that their styles explicitly set all necessary values rather than relying on defaults.

In Vewd Core versions prior to 3.4, suppressing the outline with `outline:none` is ignored, as this particular rule has traditionally been (ab)used on a variety of common `reset.css` stylesheets, without authors providing any alternative indications of focus.

Starting with the Vewd Core 3.4, it's possible to suppress the outline using a simple:

```
:focus { outline: none; }
```

However, for backwards compatibility with older Vewd Core versions, developers should still use the slightly longer syntax to suppress Vewd Core default outline indicator:

```
:focus { outline: 0 solid; }
```

Developers will still need to ensure that there is a sufficiently clear alternative way of indicating to the user where their current focus on screen is, such as setting a contrasting background colour on control elements instead. For example:

```
:focus { **background: #f00;** outline: 0 solid; }
```

Developers using specific styles for `:hover` should — as with generic keyboard accessibility considerations — also consider doubling-up these style rules for the more generic `:focus` state (note that this still requires the elements to be focusable in the first place).

Instead of just using something like

```
a:hover { /* styles applied when hovering over a link */ }
```

the style selector should be modified to also cover `:focus`, like so:

```
a:focus, a:hover { /* styles apply to hovering AND keyboard/spatial navigation focus */ }
```

Indicating `:active` element

As is common across all browsers at the time of writing, keyboard navigation (including Opera's spatial navigation, as used in the Vewd Core) does not trigger any `:active` styles that may be defined. For this reason, additional style rules such as

```
a:active { /* styles applied when a link has been activated */ }
```

will not have any effect. If your application relies on giving the user feedback when a particular element (such as a link or button) has been activated, this will need to be "faked" with something like

```
a.active, a:active { /* styles applied when a link has been activated */ }
```

and the use of additional JavaScript that sets/unsets the `.active` class on the `click` event.

Performance considerations

Processors and hardware acceleration

Typical web-enabled TV sets have modest hardware specifications compared to desktop computers. The main processor power of a TV comes somewhere between high-end smart phones and low end laptops.

By default the Vewd Core just uses the main processor for all operations, including visual compositing through the Vega graphics library (which is also responsible for advanced CSS3 visual effects such as box shadows, text shadows, transforms and transitions), SVG rendering, and JavaScript processing through the Carakan engine.

The main processor is also used for software-based blitting of the rendered page to the screen itself.

TVs often come with specialised chipsets that can provide hardware acceleration for "expensive" operations such as decoding of video streams. Whether or not the browser running on a TV set takes advantage of these chipsets will depend on the specific device, as it's a step usually taken by the

manufacturers during the integration stage of the SDK. On recent TV sets, the most commonly accelerated feature is the blitting stage, through standardised DirectFB and OpenGL libraries.

It is possible to accelerate most processor intensive features of the Vewd Core — including Vega and Carakan itself — but in general developers should not rely on any particular feature being hardware-accelerated, and instead be mindful of the processing power on web-enabled TVs being comparatively modest.

JavaScript

Because of the comparatively low spec processors found on most common TVs, developers should avoid overly heavy and complex JavaScript. For general visual/GUI effects such as carousels, slideshows and expanding/contracting boxes, custom-written and optimised scripts are usually preferable to large multi-purpose frameworks. Also, in light of cache restrictions and content optimisation, it is recommended to use “lazy loading” techniques to only call scripts when needed.

Layering and opacity

The act of displaying images on a webpage (blitting images onto the viewport) is generally hardware-accelerated on most devices. However, complex layouts — for example with overlapping, semi-transparent graphics and layered blocks with reduced opacity that blend together — require large amounts of additional processing power to calculate the actual value of each pixel that needs to be displayed on screen. These can adversely impact overall performance of pages — particularly scrolling and animations — and should be used sparingly.

Animations

One of the strengths of many JavaScript frameworks is the ability to easily script complex visual effects and animations.

However, if these are simply “eye candy” effects — often given in response to a user interaction — that are simply triggered and don’t require frame-accurate control, it is worth considering the use of new CSS3 capabilities such as transitions and transforms. As these are executed by the Vega graphics library, they should generally perform better than generic solutions using JavaScript, which typically use intervals and timers to explicitly change certain values — like colour, position, dimension — in discrete steps.

Generally however, such effects should be kept to a minimum, as any form of animation can be expensive in terms of performance, potentially rendering sites and applications slow and unresponsive.

Caching and content optimisation

Most web-enabled TVs don’t have a hard drive or solid state drive to cache web content (although some DVRs will, these are generally used to record videos, and are not generally available for use by the browser). In these circumstances the browser will run in diskless mode, using only the available RAM to cache any downloaded web content.

The amount of memory available for caching can vary from device to device, but is generally in the region of 5-10MB.

Because of this fairly low limit on the available cache, it is not safe to assume that any assets are cached when moving from one page of site to the next, or on return visits. This also applies to cookies, which may not be stored after a user has left a particular website — although they should be safe to use to store information during a single visit. Web-enabled TV and device download speeds, however, are usually close to regular broadband speeds, so not having a cache that persists for return visits may not be a huge issue. Nonetheless, developers should aim to optimise all their content — graphics, multimedia files, and even HTML, CSS and JavaScript files — as much as possible, to ensure that they don’t hit the cache limit.

In addition to this regular cache, browsers on devices will also have a fairly limited amount of memory available to store the decoded pages — the browser’s internal representation of a web page, which includes assets such as the DOM, JavaScript objects, and all uncompressed images. Once the limit for the decoded pages’ memory has been reached, the browser will generally stop rendering and loading the current page until the cache is freed again. How this happens is up to individual manufacturers — for instance, users may be alerted through a pop-up that loading has stopped and that the browser is unable to continue rendering the page. After confirmation, the page and its associated cache can then be purged. For this reason, developers should be careful when using large numbers of images and JavaScript resources, as this can potentially cause out-of-memory issues on lower-end devices.

Multimedia

Starting with Presto 2.5, Vewd Core comes with built-in support for the HTML5 `<audio>` and `<video>` elements, allowing developers to include multimedia content in their web sites without plugins. However, the specifics of which codecs are supported on devices can vary considerably, as this depends on the underlying platform and integration work carried out by device manufacturers. Current devices are often optimised for hardware-accelerated playback of MP4/H.264 content in the `<video>` element. Other codecs (such as MPEG2 for video or MP3 for audio) may be available, but this cannot be guaranteed on all web-enabled TVs.

Certain TVs allow for the use of Adobe Flash or Adobe Flash Lite content — for instance the latter is available to the Opera browser on the Nintendo Wii. Others may come with integrated support for displaying audio/video files delivered through an `<object>` element. Once again though, this cannot be relied on as it’s an integration feature. Consumers cannot install plugin technologies and codecs themselves, and such plugins are updated regularly, meaning potential compatibility issues with future content that relies on new features.

This guide only gives a generic overview of content on TV. If you are developing for environments like the Vewd App Store, please consult the specific articles relating to those products, as there will be further restrictions and requirements with regards to visual design, layout and multimedia codec availability.