Display Maintenance

A Pattern Language

August 2, 1998

Dwayne Towell
dwayne@imagebuilder.com
ImageBuilder Software
(503) 684-5151

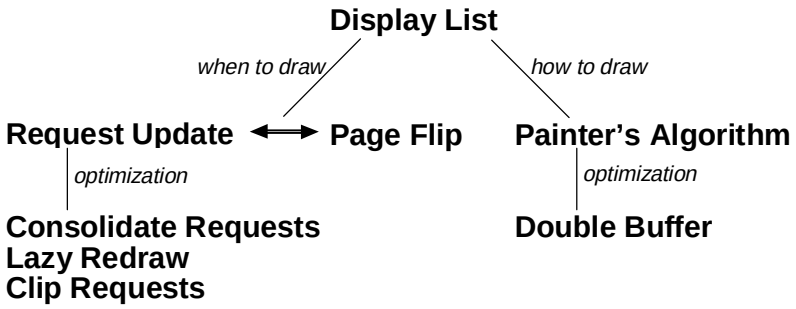
Abstract

Correct and timely rendering of graphics to a physical display is complex. Obvious solutions tend to be monolithic. *Display Maintenance* describes eight common design patterns for designing display architecture. *Display List*, *Request Update* and *Painter's Algorithm*, the kernel of the language, decompose the problem providing a modular architecture with correct behavior. The other patterns address the issue of speed.

Introduction

Display maintenance involves the correct and timely rendering of a virtual environment to a physical display. Accuracy and speed are important because the extent to which the user can experience the virtual environment is limited by the interface actually presented. If the presentation fails, the user is confronted by a discontinuity in the virtual environment and the experience suffers. If the presentation lags behind the user, again the experience suffers. Speed and accuracy are not simple goals, so together they demand complex solutions.

The *Display Maintenance* pattern language describes proven solutions to a variety of display update problems for graphical interfaces from games to desktops. These patterns capture experience from many operating systems and application frameworks. Although engineers developing these types of architectures will receive the most benefit from them, understanding the display strategies that have become so commonly used should be beneficial to anyone.



Display List, *Request Update*, and *Painter's Algorithm* form the kernel of the language as shown to the right. Although they are applicable independently, together they have a synergy that is worth more than the parts. *Page Flip* describes a radical alternative to *Request Update*. *Consolidate Requests*, *Lazy Redraw*, *Clip Updates*, and *Double Buffer* describe optimizations. The first three improve speed or perceived speed; *Double Buffer* improves the visual effects of *Painter's Algorithm*. A summary of each problem-solution pair is listed below.

Pattern	Problem	Solution
Display List	Visible component occlusion is important, yet should not be a burden to the components.	Maintain a list of all visual components; let order dictate occlusion.
Request Update	Visual components may be occluded by others, complicating display updates.	Components request display updates rather than redraw directly.
Painter's Algorithm	Rendering with correct occlusion requires significant knowledge.	Draw all components, "back to front".
Consolidate Requests	Performing individual update requests is unexpected and inefficient.	Consolidate update requests generated by a single user action.
Lazy Redraw	Multiple redraws may hinder interaction and are inefficient.	Postpone redraws until time allows.
Clip Updates	Updating hidden regions is unnecessary and inefficient.	Automatically exclude opaque foreground regions from update requests.
Double Buffer	Updating components directly to the display causes display "flicker".	Render to a hidden display, then use a single operation to update the display.
Page Flip	Tracking update requests and copying Double Buffered changes becomes inefficient.	Update the entire display on any request and swap display buffers.

Display List

Visible component occlusion is important to the graphical interface presented, yet these components should not be burdened with this responsibility.

Many graphical interfaces rely on depth cues to provide feedback to the user. Important objects such as the current document, active tool set, or dialogs appear "above" other currently less important interface components. A common way to generate an illusion of depth requires foreground objects to cover background ones. This occlusion allows work to be sorted and tools to be organized.

Adding or changing visible objects, such as a view object from a Model-View-Controller architecture [Krasner88], may directly affect other objects. For example, when a dialog box covers an active video, the real-time video component must avoid corrupting the dialog's display real estate. This requires the video component to know to avoid the dialog box. This could be solved by assuming dialogs always occlude video; a static relationship between object types could be enforced. Video components always yield to any occluding dialog box. On the other hand, if this assumption cannot be made, dynamic relationships must be established and maintained. This is the case with document windows. As the user "pulls" each window forward, it must occlude any windows "behind" it to give the illusion of depth.

New collaborations between existing types or new object types exacerbate the problem even more--object definitions must be updated. This added overhead in both the object definition, as well as the runtime organization, obfuscates the primary task.

Therefore:
Maintain a list of all visual components; let order dictate occlusion.

Require visual components to be inserted into the display list in order to be added to the interface. Items covered by any particular object have positions later in the list, while items covering it occur earlier in the list. Each individual component is no longer responsible for its inter-component property of occlusion; that property has been moved to the list.

As new object types are created, no existing type relationships must be updated. As new components are added at runtime, no private inter-component relationship needs to be updated, only the public list. If depth cues change the list can be reordered.

Consequences

Display List extracts from an architecture "visual depth" and encodes it in a data structure. This allows operations such as "am-I-in-front" and "what-is-the-mouse-over?" to be answered. It also allows the concept of depth to be well represented within an architecture. Inter-object coupling is reduced which increases object independence. *Display List* requires all visible objects to be globally known; this increases display cost both during development and at runtime.

Known Uses

Most desktop environments utilize *Display Lists*. For example, *Microsoft Windows'* windows (`HWND`s) belong to system-managed *Display Lists*. Many drawing programs expose *Display Lists* through their user-interface via features like bring-forward and push-behind. *Director* worksheets represent a type of two-dimensional *Display List*; vertical is depth and horizontal is time.

Related Patterns

Painter's Algorithm is typically used to render a *Display List*. *Request Update* can be used to decide when rendering should occur.

Display List is a specialization of *Collection Object* [Noble97].

Very expressive, hierarchical displays can be developed if each visible object is a component of a *Composite (163)* [Gamma94] object.

Request Update

Visual components may be occluded by others, complicating display updates.

If visual components overlap in predictable ways, updating the display can be relatively simple. For example, an object moving across a static background must simply redraw the background as it moves. Of course this requires the foreground component to know about the background. Or, a component moving across a background composed of several images might simply take a "snapshot" of the display before drawing itself. When the visual component moves, it restores the saved image to the display, takes a snapshot of the new location, and redraws itself. Even in this case, however, the background is not allowed to change.

These interactions between otherwise independent components complicate display updates. They force components to know details of their environment they would not normally need to know.

Even worse, if components overlap in complex and unpredictable ways, updating the display can be very difficult. For example, when a component "behind" another changes, only the visible parts should be updated until the foreground component no longer occludes it.

Requiring this level of inter-component negotiation complicates not only display updates, but design, implementation and maintenance as well.

Therefore:
Components request display updates rather than redraw directly.

As time or events force a component to change, it modifies its internal state, but does not actually perform a display update. Instead it posts a request for the display to be updated, usually including information about the area suspected to be out of date.

For example, a moving ball might periodically update its location and then request a display update of its previous and current locations. When the system redraws the display, it can guarantee the area once occupied by the ball will be updated with any items behind it. Likewise, the ball will be drawn in its new location.

Consequences

Request Update requires visible objects be able to redraw upon request. This means extra state information could be required if requests can be occur during state changes.

Request Update decouples the invoking objects from the resolving objects. This improves design, but can severely impact performance over the monolithic design requiring all objects to know all related objects.

Known Uses

Microsoft Windows' uses `InvalidateRect()`, and related functions, to invoke a *Request Update*. Windows receive a `WM_PAINT` message when requested to redraw. HotDraw `Figures'` use *Request Update* to invalidate the view when their state changes.

Related Patterns

Request Update assumes a *Display List* is available in order to honor the request. Usually, *Consolidate Requests*, *Lazy Redraw*, and *Clip Requests* are used to improve efficiency.

Painter's Algorithm

Rendering a single visible component with correct occlusion requires significant inter-object knowledge of boundaries.

Using depth as an interface element requires each visible object be subject to arbitrarily complex visible boundaries. For example, in desktop environments the desktop itself, simply another visible object, must always appear "behind" other objects. Updating an object's appearance requires determining its display boundaries. This, in turn, requires all objects to report their exact boundaries in a globally defined format. Although possible, this is generally not practical in light of text and other complex shapes. Assuming, boundaries can be determined, all objects are required to support complex display restrictions in order to prevent incorrectly occluding other objects.

If objects can be translucent boundary information is not enough. Image data for partially occluded objects must be available for blending with the "foreground" image. This allows the background objects to appear visible "through" the foreground object.

Therefore:
Draw all components, without regard to occlusion, "back to front".

Similar to the way an oil painter might fill a canvas, *Painter's Algorithm* applies each object to the display from "back to front". This allows each visible object to replace or change any part of the image "behind" it. Because all objects are guaranteed to be drawn in order, each object need only be responsible for its update mechanism. Update boundary management is eliminated because all occluded areas will have been correctly overwritten.

Consequences

Painter's Algorithm requires the display to be re-writable, and in the case of translucent drawing, that it be readable. *Painter's Algorithm* also imposes a heavy penalty: display update cost. Completely updating of all visible objects is time consuming and wasteful. Fortunately, several *Display Maintenance* patterns provide optimizations.

Known Uses

Netscape, and many other web browsers, use *Painter's Algorithm* to ensure proper occlusion. When the background image changes, usually because it has (finally) been downloaded, the entire browser display region is redrawn. Microsoft Windows may use *Painter's Algorithm* when updating the display if computing an appropriate "clipping region" is too expensive. *HotDraw* [Johnson92] uses *Painter's Algorithm* within `Drawing`.

Related Patterns

Painter's Algorithm is usually implemented on a *Display List*. If the visible objects are completely known at design time, unlikely but possible in some game environments, a *Display List* is not required.

Painter's Algorithm can be thought of as a *Chain of Responsibility (223)* [Gamma94] for updating the display. As with all *Chain of Responsibilities*, coupling is reduced at the expense of a loss of guarantee of receipt. In this case, receipt can be assumed because presumably the request originates from within the chain.

Double Buffer can be used to reduce display "flicker" normally generated by *Painter's Algorithm*. This is especially true when objects are deeply layered, objects have complex boundaries, and/or objects include translucency, such as in a game or slide show.

Consolidate Requests

Performing individual update requests is unexpected and inefficient.

User events make changes to the virtual environment represented by the display. Even though these operations are perceived as single actions, several objects may be effected. For example, changing a desktop property might effect all the desktop icons. Internally each item must be updated in turn, but they should update nearly simultaneously to allow the user to perceive them as a single operation.

When updating many items it is desirable to pay overhead costs only once. Several things contribute to high redraw overhead. A large number of objects increases iteration and caching costs. Determining each object's relevancy can expensive, as in the case of a CAD application. Also, re-rendering any object certainly decreases performance. If redraw overhead is high, performing multiple redraws can severely impact the application as a whole.

Therefore:
Consolidate update requests generated by a single user action.

As update requests are made, instead of performing each update immediately, merged the request with any previous requests. The merger produces a running "invalid" region to be updated later. After all updates have been made, redraw the display satisfying all requests at the same time. This not only results in all updates happening rapidly, but reduces the total cost by eliminating much overhead.

Consequences

Consolidate Update Requests postpones fulfillment of requests while the system is in transition. This can improve the user's perception of virtual environment being modeled. On the other hand, if transition times are lengthy, it can give the appearance the system has failed to respond.

Known Uses

Microsoft Windows, the *Macintosh Operating System* and *HotDraw* use *Consolidate Update Requests* to organize and optimize redrawing.

Related Patterns

Consolidate Update Requests is an optimization of *Request Update*.

Consolidate Update Requests is also known as *Collect Damage* [Beck94].

Lazy Update

Multiple redraws may hinder interaction and are inefficient.

As long as redraws can be performed as fast as they are requested, the virtual environment interacts well. When events require display redraws more frequently than they can be performed, the virtual environment becomes hindered by unresponsiveness. Frequent requests can occur due to certain types of user actions, such as window resizing on a desktop, and to timed events, such as game elements. Redraw performance can suffer for many reasons including: complexity, number of elements, and bandwidth available. Since the virtual environment is designed for interaction a compromise must be reached.

Therefore:
Postpone redraws until time allows.

Although the display provides a view of the virtual environment, the user actually interacts, not with the display, but with the virtual environment. Therefore, continue to update the virtual environment, but only update the display if time is available. In most environments, this means that as long as input events are actually available, no redraw is performed. Once all user input has been processed the display is brought upto date. This provides a smooth transition from a completely responsive system with resources to spare to an overworked system with few resources available.

Consequences

Lazy Redraw postpones fulfillment of requests until the system has resources available to perform them. This can improve performance for the experienced user, since immediate feedback is not necessary. The novice user, on the other hand, is more likely to wait for confirmation (i.e., redraw) before proceeding.

Since *Lazy Redraw* implements a heuristic for metering performance based on input requests it may not always be correct. Therefore, a method for programmatically forcing a redraw should always be provided. For example, regardless of how poor performance may be, a slide show program is expected to correctly display certain "key frames".

Known Uses

Microsoft Windows and *HotDraw* use *Lazy Redraw* to optimize redrawing.

Related Patterns

Lazy Redraw is an optimization of *Request Update*.

Lazy Redraw is also known as *Update at User-Speed* [Beck94].

Clip Updates

Updating hidden regions is unnecessary and inefficient.

When update requests occur for occluded regions, redrawing the foreground is inefficient and unnecessary since it will not change. Consider a background status window on a desktop updating. If the update region is completely occluded, there is no need to perform the request because when the region becomes visible it will be redrawn in response to revealing request. Or, if the region is partially visible, it would be most efficient to redraw only the visible portion. However, visibility is a dynamic property of the entire environment, not information readily available to the process making changes to a particular window.

Therefore:
Automatically exclude opaque foreground regions from update requests.

The display environment, the desktop in this example, has information about visibility. When each request is made, any completely opaque regions occluding the requested area are automatically excluded from the request. These areas represent wasted effort.

Consequences

Clip Updates eliminates wasted effort at the expense of additional information. Update requests must include a source, as well as a region, in order to allow the system to determine which areas should be ignored. This requires slightly tighter binding between update processes and the display system.

Clip Updates works best when requests and visible components are restricted to rectangular regions and all objects are opaque. This is typically the case with desktop systems and it is used extensively in these environments to speed redraws and promote the illusion of solidity.

If objects are translucent they should not be considered to occlude other objects, since the final image depends on more than one object. Also, as object boundaries become complex, managing the update region, and clipping redraws to it, can easily generate more work than is being saved.

Known Uses

Microsoft Windows, the *Macintosh Operating System* and other desktop environments use *Clip Updates* to optimize redrawing.

Related Patterns

Clip Updates is an optimization of *Request Update*.

The *Display List* provides a ready-made *Chain of Responsibility (223)* [Gamma94] for trimming requests. Each visible component is responsible for removing any part of the request it guarantees to occlude.

Double Buffer

Updating components directly to the display causes display "flicker".

When using *Painter's Algorithm* to update the display, undesirable side effects result. Changing components momentarily disappear before reappearing in a new state or at a new location. This effect breaks the illusion of permanence and solidity. Although this may be acceptable for desktop environments, it is generally not acceptable for games and other multimedia applications such as slide shows.

Another unwanted effect results when objects change while in "front" of several background components, each of those components must be redrawn before the changed component is updated. This excessive redrawing produces display "flicker" as components are updated. Not only is this distracting to the user, it interferes with the illusion of depth and draws unwanted attention to otherwise simple changes.

Even worse, if a background component changes, foreground components must be redrawn even if they do not change. In cases where the background components are completely hidden, it may only be distracting for desktops, but it is completely unacceptable for games where hidden objects should never be visible.

Therefore:
Render visible components to a hidden display, then use a single operation to update the visible display.

Rendering to a hidden display avoids redraw effects. "Flicker" and hidden component visibility are avoided since the user cannot observe the redraw process. Once the hidden display is complete a single operation is used to copy the new image to the visible display. This allows components to instantly change from one location to another, or from one image to another, even when partially occluded, completely hidden, translucent, or in front of complex backgrounds.

Consequences

Double Buffer trades resources for presentation. Display buffer memory, potentially a premium resource, and time are the most significant resources required. Considerable design and implementation changes may be required especially if objects rely on "extra" display features such as cursors, carets or "rubber-banding" services.

Known Uses

Power Point, *Doom*, *QuickTime* and many other products use *Double Buffer* to improve presentation quality.

Related Patterns

Page Flip is a specialization of *Double Buffer* that may be appropriate if large areas of the display will be updated.

Page Flip

When *Double Buffering* and a large portion of the display is continually updated, tracking update requests and copying *Double Buffered* changes becomes inefficient.

For many games and presentation products, *Consolidate Updates* forces most of the display to be updated each time. In these cases maintaining "invalid" regions, clipping requests against opaque regions, clipping redraws against the "invalid" regions, and especially copying between display buffers can dramatically drain resources needed for the primary task--presenting a rich visual experience.

Therefore:
Update the entire display on any request and swap display buffers.

Instead of tracking update requests individually, force each update to re-render the entire display. Not only does this eliminate many potentially slow maintenance algorithms from the application, usually rendering the entire display is more easily optimized. When rendering is complete, swap the hidden display for the visible one.

Consequences

Page Flip trades finesse for brute force. In doing so it radically changes performance by redefining the bottlenecks. Hardware bandwidth replaces maintenance overhead, which may be appropriate if hardware is cheaper than engineering costs. It also allows a single path through the application to be fully optimitized.

Page Flip can best be utilized if some form of hardware support is available. Usually this means the display subsystem supports physical display buffer swapping. However, if hardware "blitter" bandwidth is available it can be used to emulate the swapping effect.

Known Uses

DirectDraw, *Sega*, *Nintendo* and many other game products use *Page Flip* to improve performance.

Related Patterns

Page Flip is an optimization of *Double Buffer*.

Acknowledgements

This paper was prepared for presentation at PLoP'98. Thanks are due to James Noble for his great shepherding and Roger Bonzer for comments on several drafts.

Bibliography

[Beck94] Kent Beck, Ralph E. Johnson. "Patterns Generate Architectures", *Proceedings of the 8th European Conference on Object Oriented Programming*. Lecture Notes in Computer Science, 821:139-149, Bologna Italy, July 1994.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1994.

[Johnson92] Ralph E. Johnson. "Documenting Frameworks with Patterns", *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10):63-76, Vancouver BC, October 1992.

[Krasner88] Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, 1(3):26-49, 1988.

[Nobel97] James Noble. "Basic Relationship Patterns", EuroPLoP'97, 1997.