Nested Compositing Window Managers

Anthony Savidis[1,2] and Andreas Maragudakis[1]

[1] Institute of Computer Science, Foundation for Research and Technology – Hellas (FORTH)
[2] Department of Computer Science, University of Crete, Greece
{as,maragud}@ics.forth.gr

Abstract. Compositing is currently the prevalent rendering paradigm for window managers. It applies off-screen drawing of managed windows with final image composition by the window manager itself. In this context, a compositing system is presented, enabling the concurrent presence of multiple window managers, being arbitrarily nested while facilitating switch managers on-the-fly. Two distinct managers are implemented, 2d desktop and custom 3d book, that can be freely combined into nested hierarchies. To allow such nesting two extensions are introduced. Firstly, the compositing process is turned to a rendering pipeline with window managers directly in-the-loop, with an imaging model combining diverse geometries. Secondly, to facilitate focus control in such geometric spaces, a cascaded pointing translation process is implemented, enabling geometric mapping of pointing events across nested window managers. The entire compositing system is implemented in a custom widget toolkit named sprint (in C++ with OpengGL and shaders) that is publicly available.

Keywords: Development methods, Interaction techniques, platforms and metaphors, Window managers, Toolkits.

1 Introduction

Currently, the imaging model of most window managers is compositing. The latter relies on the drawing of managed (top-level) windows into off-screen buffers (normal rendering phase), with the window manager responsible to eventually compose the final picture from such buffers (compositing rendering phase), shown in Fig. 1. Also, compositing does not radically affect individual window rendering, since it is still performed as before, however, with output redirected in off-screen-buffers.

Effectively, it brought two changes: (i) the final rendering stage; and (ii) initial pointing filtering from window manager space to local (planar) window space. While the amendments are overall simple, they are powerful in terms of the visual scenarios they support, and radical regarding the underlying implementation rework they require (i.e., GPU rendering).

We discuss two novel extensions along the standard compositing features. Firstly, we extend the imaging model to support nested window managers, thus enabling arbitrarily nested interactive spaces. In this context, nesting is possible on different window managers, while enabling users switch managers on-the-fly. Secondly, we extend

C. Stephanidis and M. Antona (Eds.): UAHCI/HCII 2014, Part I, LNCS 8513, pp. 149–160, 2014.

the initial pointing filtering process towards a cascaded translation process in order to support focus control across nested window managers and their custom geometries.

We continue with the novel features of the compositing system. Then, we discuss the primary implementation aspects and patterns to accommodate them. Finally, we compare with related work, draw key conclusions and outline future steps.

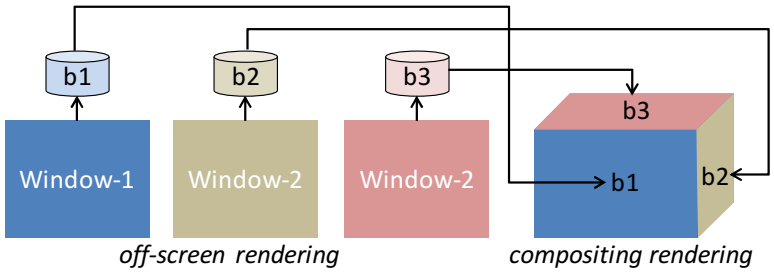

Fig. 1. Compositing rendering process combining buffers from off-screen rendering

2 Related Work

Window managers (Myers, 1988) with compositing implementations appeared originally under X windows more than a decade ago, following early pioneering work with first 3d window managers such as the Task Gallery (Robertson et al., 2000). Event today compositing managers like Compiz (Canonical, 2013), KWin (KDE, 2013) and Quartz (Apple, 2013) do not have interoperating variations and always work in a standalone fashion. Currently, for users to alternate across different spaces the notion of virtual desktops is offered (Ringel, 2011), but always with a similar window management style.

Additionally, they still operate outside the toolkit loop: after the toolkit in terms of display composition, and before the toolkit in terms of the initial input translation. While not related to compositing, the work on facades (Stürzlinger et al., 2006) revealed the need for compositional user-interfaces, something that inspired us towards dynamic window manager switching.

Display improvements for various scenarios have been proposed, like optimal space exploitation (Bell & Feiner, 2000), improved desktop rendering for importance-driven compositing (Waldner et al., 2011), optimal display usage for different monitor sizes (Hutchings et al., 2004; Hutchings & Stasko, 2004). In general, display improvements could be modeled in window managers as modular layout add-ons implementing different policies; however, we did not focus on this particular problem in the reported work. A constraint-based approach to model such policies is proposed in (Badros, 2001).

Various research efforts on compositing systems have been carried out, but little progress is made in extending the compositing pipeline with radical refinements such multiple and nested window managers. Metisse is a flexible compositing system (Chapuis et al., 2005) relying on FvwmCompositor which allows 3d transformations

on window rendering, thus offering a framework for rendering windows beyond typical desktop topologies. However, it still cannot combine different window managers, while pick translation is typical single-stage preprocessing involving the window 3d transformation matrix.

3 Features

We implemented a compositing toolkit with two distinct window managers: (i) desktop window manager with a 2d geometric space and rectangular planar geometry; and (ii) custom book window manager with a 3d geometric space with a typical perspective view frustum. A snapshot with nested desktop and book managers, showing texturing and triangulation involved in compositing is provided in Fig. 2, while cross nesting with book and desktop managers is provided under Fig. 3.

Typical live taskbars with window miniatures are implemented, as in most compositing window managers, relying on the off-screen-buffers for the contents of managed windows. In our implementation they are currently included in desktops, while we are working on a geometric model suited to the book window manager.

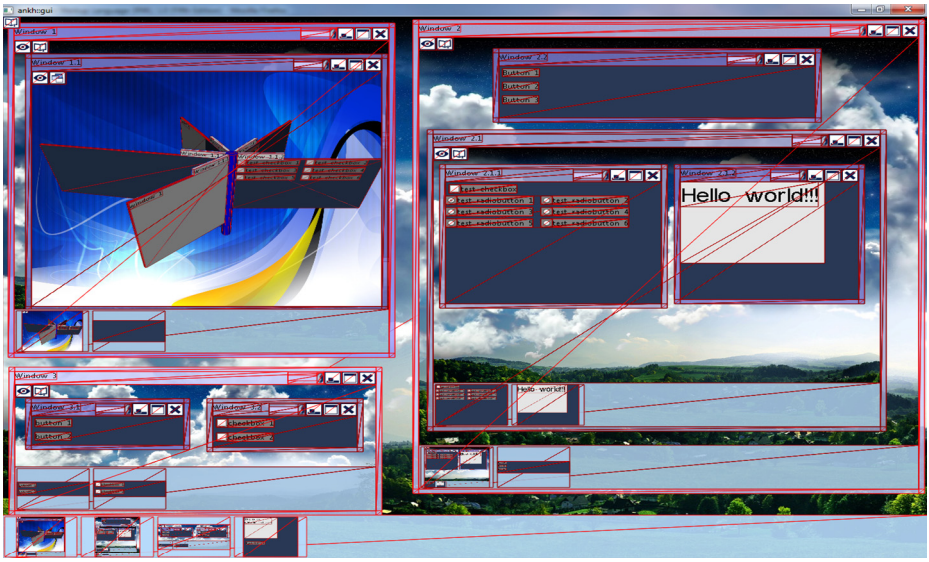

Fig. 2. Mixed window managers with windows as textured (two triangles) rectangles

Switching from one window manager to another is facilitated on-the-fly, and, as we discuss later, in an automated implementation manner. Since switching is also an operation on window managers it is also included in their special local toolbar (see Fig. 3, top-left window areas, right to eye icons, showing a desktop or book icons).

Because of nesting, window maximization can have a dual meaning. The common meaning, as observed in (standalone) desktop managers, is making windows occupy

the full-screen space. However, in managers with non-desktop rendering plugins, like the cubic renderers of Compiz and KWin, the behavior is different with maximization adjusted to the rendering space were windows geometrically 'live' - in the previous cases the actual cube sides. Since the two interpretations are different, we decided to separate them as follows.

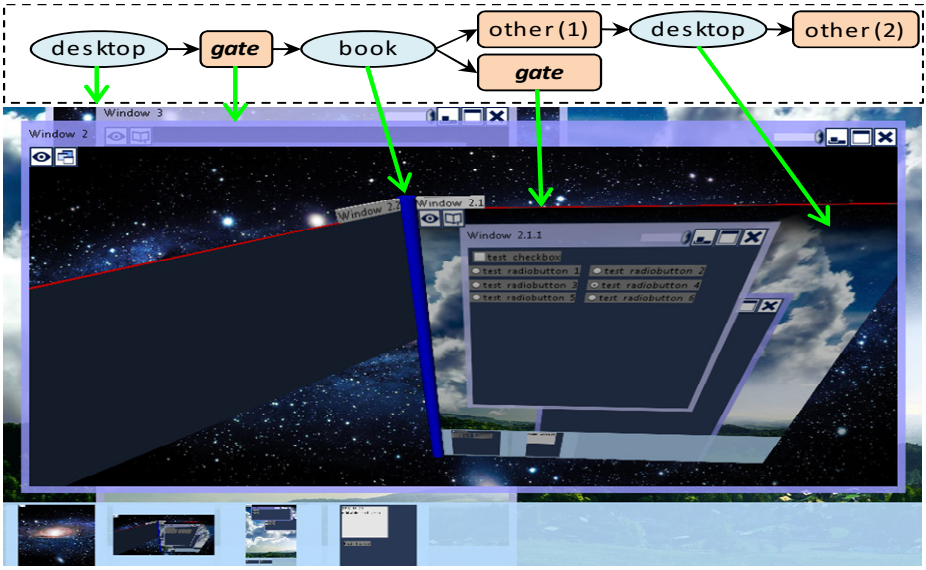

Fig. 3. Desktop encompassing a book encompassing a desktop

Manager maximization / restoration, offered to window managers only, not to windows, allowing them exploit full screen space. It enables give the impression that a window manager is standalone, (see Fig. 4, right part). In our implementation this mode is interactively offered by a local toolbar at the top-left of the window-manager rendering area, and concerns to the 'eye' icon (see Fig. 3 and Fig. 4, toolbars at top-left window areas). Additionally, successive manager maximize requests are allowed on nested managers, with the restore operation always returning to the full-screen state of the hierarchically-closest parent manager.

Window maximization / restoration, an operation on managed windows, is optionally offered by individual window managers depending on their individual rendering models. In our case it is only supplied by desktops and is included in the standard window frame toolbar. Through this operation windows take the full space in area of their parent window manager (see Fig. 4, 'window maximize'). In our context, if a desktop is the root manager or is maximized, this operation maximize behaves exactly as in existing desktops and makes the window occupy full-screen space.

When supporting nested window managers the resulting interactive spaces not only become visually rich, but also sometimes visually complex. In particular, the desktop with its overlapped windows may cause nested window managers to be obscured. To allow users quickly inspect what is behind we introduced an extra item in the desktop

toolbar for interactive transparency control (see Fig. 4). Our implementation concerns appearance and does not allow obscured content to be interactive without the focus, as in (Robertson et al., 2000) were interaction with hidden content is enabled.

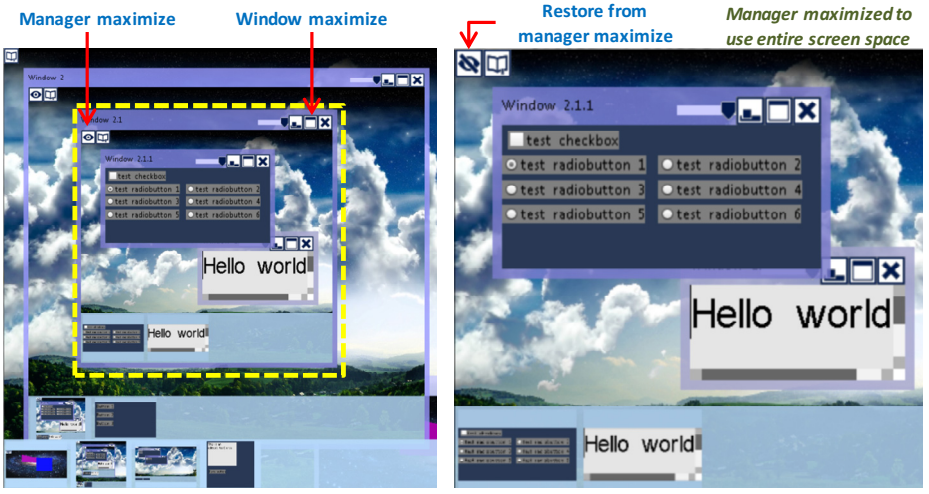

Fig. 4. Dual maximization operation applying on window managers (eye icon) and on managed windows (standard icon)

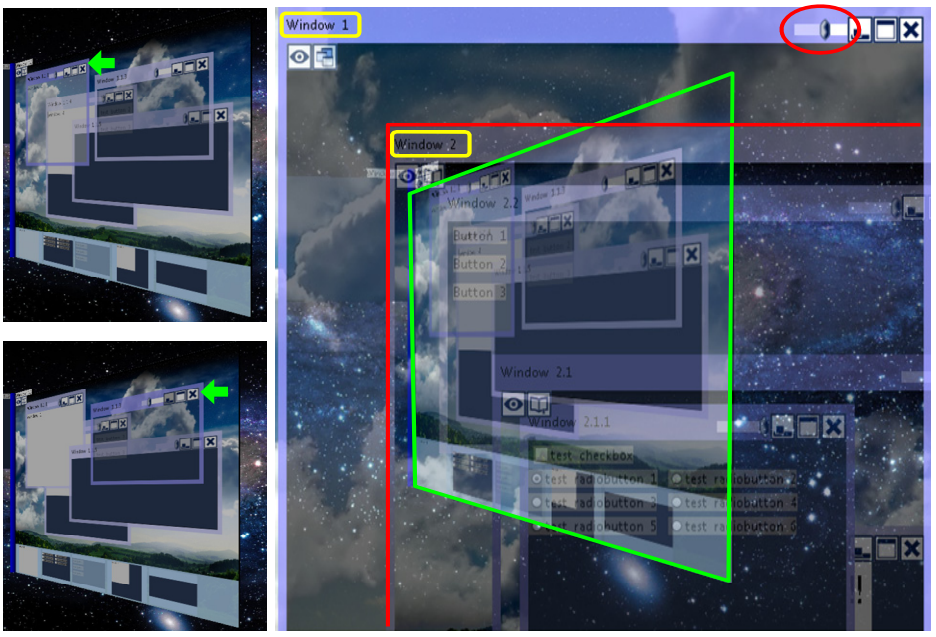

Fig. 5. Transparency control; Left: on windows of a desktop nested in a book; Right: book nested in 'Window 1' of desktop ('Window 2' is transparent and in-front of 'Window 1')

4 Implementation

In compositing toolkits, the classes regarding managed windows have always a local off-screen-buffer. Under OpenGL implementations the latter is typically a frame buffer object with a texture rendering target. In this context, to support dynamically nested window managers we introduced a managed window subclass named *gate* (see Fig. 6), with a dynamically associated window manager instance (optional, can be null). The root of the entire window hierarchy is always a gate, while gates may be freely nested within any container window. However, only instances of managed windows can have a gate instance as a parent.

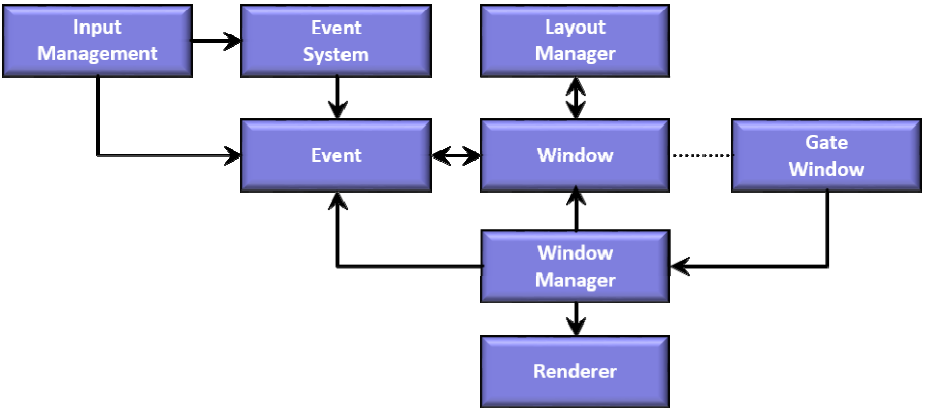

Fig. 6. General architecture of our toolkit, with *Gate* window class playing a key role

The class design to accommodate our requirements is outlined under Fig. 7 (many details omitted for clarity). Overall, window managers rely on decorators, one decorator added per managed window. The latter concerns the WindowManager (WM) abstract class, its Decorate abstract method, and the WindowDecorator (WD) abstract class. This way, managed windows are added / removed to / from window managers as follows:

```
static WM::Add (ManagedWindow win)
        { decorators[win] = Decorate(win); }
static WM::Remove (ManagedWindow win)
        { decorators[win].Destroy(); decorators.erase(win); }
```

As also shown under Fig. 7, subclasses of window manager and decorator are defined in pairs, something reflected in the desktop and book window managers we implemented. One role of the decorator is to optionally introduce extra widgets necessary for interactive window control. The latter is a standard approach on desktops, originally introduced by window managers for X windows, with frames added as extra windows. We continue with the implementation details of the rendering pipeline, dynamic switching and cascaded input translation.

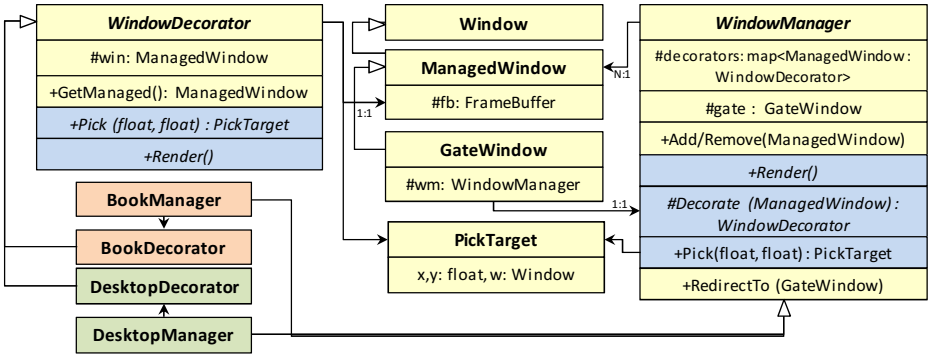

Fig. 7. Class schema to support dynamic and hybrid nesting of compositing window managers - key methods and members are only shown for clarity

The rendering process is recursive as in all toolkits. When a gate window is asked to render itself it simply delegates the rendering work to its window manager instance:

```
GateWindow::Render() {
        Set rendering target to this.GetFrameBuffer( )
        Draw any background image
        GetWindowManager().Render()
}
```

We continue with the *Render*() methods at the level of super-classes. There are also three non-abstract methods, two for rendering managed windows and decorators, and one to render the local toolbar (this is discussed later).

```
WM::RenderWindows() {          ←non-abstract method
        foreach x in decorators do
                x.GetKey().Render()
}
WM::RenderDecorators() {       ←non-abstract methods
        foreach x in decorators do
                x.GetValue().Render()
}
WM::Render() {
        RenderWindows( )
        RenderDecorators( )
        RenderLocalToolbar()
}
```

Then, in subclasses we may refine the abstract Render methods as required. We outline the implementation case regarding for the book window manager:

```
WD::Render(){
    // default implementation of window decorator (WD) super-class is empty
}
BookWM::Render() {
    RenderWindows()
    Prepare all 3d polygons for book
    Refresh all textures for the modified windows
    Set display camera and draw the entire scene
    RenderLocalToolbar()
}
```

As observed, in the book implementation the decorator subclass has no particular role in rendering. This is no general rule, but reflects our choice to gather all triangles for managed windows into one rendering batch (it is much faster than having separate batches per window). As mentioned, the entry points for dynamically attaching window managers are gate window instances. Interactively, the dynamic switching options are included in the local toolbar of window managers. In our implementation, the construction of this toolbar is automatic and can accommodate any future window manager subclasses that may be possibly implemented.

In other words, if an extra window manager is implemented, it will directly appear with a reserved entry in this toolbar. For this to work, a specific design pattern needs to be adopted, as outlined under Fig. 8. In particular, a factory has to be implemented and a factory instance should be initially (at startup) registered to a factory directory with a suitable unique class identifier.

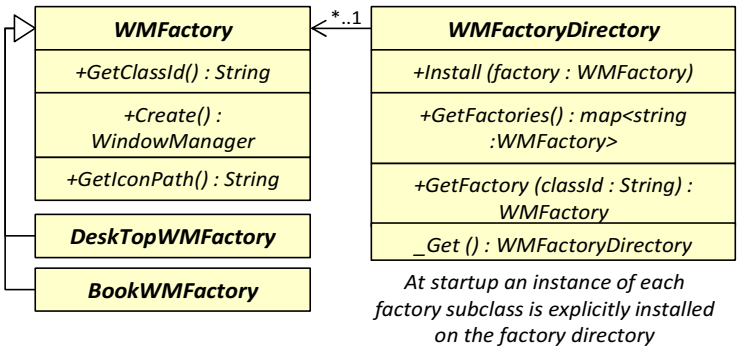

Fig. 8. Abstract superclass for window manager factories and the singleton registry with all factories indexed using their unique class ids

The local window-manager toolbar is placed on top of the rendering output that is drawn in the gate frame buffer, while it is interactively a standard toolbar with clickable behavior. It is prepared on-demand during rendering (i.e., when changed, it is internally cached) as follows:

```
WM::RenderLocalToolbar() {
    Get a copy of all factories from the factory directory
    Remove the entry for the wm class id of the caller (this)
    Draw manager maximization entry and set its click handler
    foreach x in factories do {
      Draw the texture corresponding to x.GetIconPath()
      Set the click handler of the toolbar entry to invoke WM::Switch()
    }
}
```

Once a toolbar item is selected, the associated window manager is instantiated and set as the current in its respective gate instance, destroying the previous manager. This approach enables any number of window managers to be interactively activated and combined. The previous behavior is possible through the Switch method below:

```
static WM::Switch (GateWindow gate, string wmClassId) {
    gate.GetWindowManager().Destroy()
    gate.SetWindowManager(Create(wmClassId))
}
static WM WM::Create (string wmClassId) {
    factory = get factory entry from directory for wmClassId
    return factory.Create()
}
```

```
uniform   mat4      projectionMatrix;
in        vec3      inVertex;
void main()
          { gl_Position = projectionMatrix * vec4(inVertex, 1.0); }
```

```
uniform   vec4      color;
void main()
          { gl_FragColor = color; }
```

```
uniform   sampler2D tex;
uniform   float     transparency;
uniform   bool      useTransparency;
in        vec2      textureCoordinates;
void main() {
          gl_FragColor = texture(tex, textureCoordinates);
          if (useTransparency)
                    gl_FragColor.a = transparency;
}
```

Fig. 9. The simple shaders for applying compositing rendering; the vertex shaders should use orthographic matrices

For the rendering of the windows we used custom GLSL vertex and fragment shaders. Specifically, we had to use different very simple shaders for the rendering of the desktop window manager and the contents of the top-level windows and other shaders for the 3d shapes of the book window manager. Also, the shaders are divided to the shaders with and without texturing (see Fig. 9). The shaders that are not using textures, such as clearing backgrounds, apply a color to the triangle surface that is provided as a uniform variable.

In Fig.9, we see the GLSL shaders that were used for the rendering of the 2d orthographic scenes. In the first program the vertex shader applies the orthographic projection to the input vertices. Usually, the input vertices are the window rectangles, which are converted to two triangles. The second program is the fragment shader that applies the uniform color to every fragment of the window rectangle. Finally, in the last program, we see how the texturing is performed in the fragment shader that renders the rectangles with background images, such as the gate window, the icons and the text. The texture coordinates are provided from the vertex shader, which simply passes through the input variable. The texture function, provided by the GLSL language, is used to get the color from the texture. In some cases, such as when the transparency slider is moved, we want to force a transparency value to the output window. In that case, we assign the transparency value to a uniform.

5 Cascaded Pick Translation

Picking through pointing is of key importance in toolkits and windows managers since it is used others for focus control. In our context, pointing is translated from the top window manager to the nested ones via a cascaded translation process (Fig. 10).

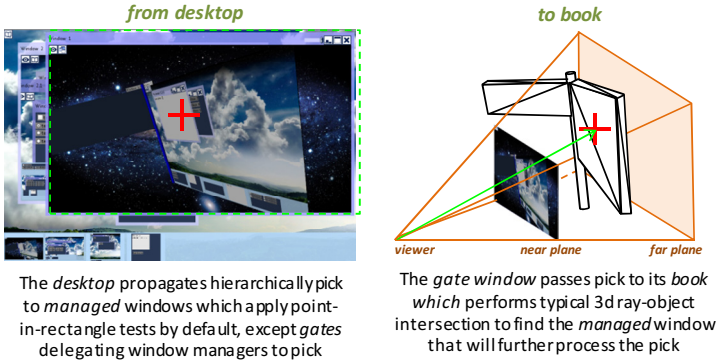

Fig. 10. Cascaded pick processing enabling to support precise pointing interaction and focus window shifting across the diverse geometries of nested window managers

More specifically, every window manager is able to translate from a planar position, whose coordinates are given relative to its gate, to a pair carrying the actual picked managed window and a relative position inside it. Then, the managed window

invokes pick recursively to child windows. As with rendering, gates always delegate pick requests directly to their window manager:

```
PickTarget GateWindow::Pick (float x, float y) {
    return GetWindowManager( ).Pick(x,y)
}
```

The initial pick request is always in screen coordinates and is supplied to the root gate instance. The implementation of the default pick logic at the level of the window manager superclass is to iteratively test managed windows. The latter is directly sufficient for the desktop manager subclass.

```
PickTarget WM::Pick (float x, float y) {
    foreach x in decorators do {
        PickTarget target = x.GetKey( ).Pick( )
        if target.GetWindow( ) ≠ null then
          return target
    }
    return null          ←fallback means no pick target was found
}
```

However, when it comes to the book manager subclass, the pick logic is more comprehensive and is provided below. Initially, the planar pick event is translated to the geometry of the book by producing a ray on the perspective view frustum using the current camera. Then, translation proceeds by recursively invoking the Pick method on the managed window corresponding to hit book page. The latter is done after translating the ray intersection point to the local planar space of the window.

```
PickTarget BookWM::Pick (float x, float y) {
    Translate the pick coordinates x,y to an 3d ray
    Using this ray determine the closest intersecting book page and its hit point
    if a book page element is intersected then {
        Let win be the managed window for this book page
        Translate hit point to page plane coordinates x′,y′
        return win.Pick(x′,y′)   ←this call will recursively pick to nested managers
    }
}
```

6 Conclusions and Future Work

We have implemented a low-fidelity experimental toolkit named sprint, supporting compositing rendering and hosting of arbitrarily nested window managers. We enabled users switch window managers on-the-fly to any registered alternative window manager. To experiment with different geometries we implemented a desktop

and book window managers and tested various dynamic nesting configurations between them. In this context, we mainly focused on the technical amendments and implementation for nested compositing window managers, rather than on the human factors and interaction quality inherent in hybrid and nested workspaces. In summary, we propose the involvement of window managers into the loop with a cascaded rendering and pick-translation pipeline.

Our implementation also indicated that a few extensions on the toolkit side are required, the most prominent being the need for a special window class, called gate in our work, to offer hosting of embedded window managers. The tests also showed that switching is extremely fast, since it only involves the initial creation of the window manager custom geometry. The contained windows and window managers are by default cached due to compositing rendering.

References

1. Myers, B.: A taxonomy of window manager user interfaces. IEEE Computer Graphics and Applications 8(5), 65–84 (1988)
2. Robertson, G., Van Dantzich, M., Robbins, D., Czerwinski, M., Hinckley, K., Risden, K., Thiel, D., Gorokhovsky, V.: The Task Gallery: a 3D window manager. In: Proc. CHI 2000, pp. 494–501. ACM (2000)
3. Bell, B.A., Feiner, S.K.: Dynamic space management for user interfaces. In: Proc. UIST 2000, pp. 239–248. ACM (2000)
4. Badros, G.J., Nichols, J., Borning, A.: Scwm: An Extensible Constraint-Enabled Window Manager. In: USENIX Annual Technical Conference, FREENIX Track 2001, pp. 225–234 (2001)
5. Ishak, E.W., Feiner, S.K.: Interacting with hidden content using content-aware free-space transparency. In: Proc. UIST 2004, pp. 189–192. ACM (2004)
6. Waldner, M., Steinberger, M., Grasset, R., Schmalstieg, D.: Importance-driven compositing window management. In: Proc. CHI 2011, pp. 959–968. ACM (2011)
7. Stürzlinger, W., Chapuis, O., Phillips, D., Roussel, N.: User interface façades: towards fully adaptable user interfaces. In: Proc. UIST 2006, pp. 309–318. ACM (2006)
8. Chapuis, O., Roussel, N.: Metisse is not a 3D desktop! In: Proc. UIST 2005, pp. 13–22. ACM (2005)
9. Hutchings, D.R., Smith, G., Meyers, B., Czerwinski, M., Robertson, G.: Display space usage and window management operation comparisons between single monitor and multiple monitor users. In: Proc. AVI 2004, pp. 32–39. ACM (2004)
10. Hutchings, D.R., Stasko, J.: Shrinking window operations for expanding display space. In: Proc. AVI 2004, pp. 350–353. ACM (2004)
11. Ringel, M.: When one isn't enough: an analysis of virtual desktop usage strategies and their implications for design. In: CHI Extended Abstracts, pp. 762–763. ACM (2003)
12. Apple. Quartz Compositor, `http://apple.wikia.com/wiki/Quartz_Compositor` (accessed April 2013)
13. Canonical LTD. Compiz, `https://launchpad.net/compiz` (accessed April 2013)
14. KDE. KWin, `http://techbase.kde.org/Projects/KWin` (accessed April 2013)