Tiling window manager 的应用与设计

宋方睿
emacs.ray@gmail.com

2012 年 1 月

摘要

Window manager 是窗口系统中管理窗口布局和装饰的组件，本文研究了其中 tiling window manager 的应用。尽管历史久远，但一直以来应用不多，直到近来随着硬件发展，显示器越来越大，它才重新开始受到关注。本文从其概念开始，探讨 tiling window manager 中布局的选取、设计、实现，提出了一些方便使用的功能。

关键词： window manager, tiling window manager, layout algorithm

目录

1 X Window System 和 window manager

Window manager 是窗口系统中用于管理窗口的位置、大小、外观等的软件，是桌面环境的一个重要组件。提到 window manager，就不得不提 X Window System，尽管 window manager 不是 X Window System 特有的概念，但 X Window System 具有的 window manager 类别、数量最多，而且它在整个 X Windows System 中的分工也非常明确，从它入手能使我们更专注于对 window manager 特性的研究。

X 窗口系统主要分成 X server 和 X client 两个组件。简单来说，X server 管理硬件，接受键盘、鼠标等设备的输入信息，并负责把图形绘制到显示器上；X client 就是窗口应用程序，处理来自 X server 的输入信息，并告知 X server 它所对应窗口的图像。

X window manager 是一类特殊的 X client，它负责管理所有的 X client(包括一些特殊程序如 taskbar [1] ，system tray [2])，其主要用途是设置 X client 的位置。它也提供某些特殊功能，如为程序分配焦点，管理 virtual desktop [3] ，提供窗口控制的接口 (允许用户移动窗口，使窗口最大、最小化等)，管理壁纸。

Window manager 主要有以下三类：

- Stacking window manager

  又称 floating window manager。在屏幕上按一定顺序绘制各个窗口，允许不同窗口重叠。若发生重叠，则先绘制的窗口的这一部分将不会显示。这就好比在桌子上摆了一些纸，它们可能会互相重叠，放在下面的纸会被上面的纸遮住。

  典型的 stacking window manager 有 Sawfish，FVWM，Blackbox，Window Maker，Microsoft Windows XP 的默认 window manager 等。

- Compositing window manager

  类似于 stacking window manager。区别在于。每个窗口不是直接被描绘到屏幕上，而是先描绘到每个窗口特有的缓冲区中，然后把所有窗口对应的缓冲区的图像合成描绘到屏幕上。这样附加带来的一个好处是可以应用很多特效，如阴影、半透明等。

  典型的 Compositing window manager 有 Compiz，KWin(4.0 之后版本)，Mutter，Microsoft Windows Vista 的默认 window manager 等。

[1] 显示当前运行程序的图标、标题，可以通过点击来使该程序的窗口获得焦点
[2] 一般显示当前运行程序的小图标，类似于 Windows 中的 notification area
[3] 虚拟桌面，一个时刻只有一个虚拟桌面显示在显示器上

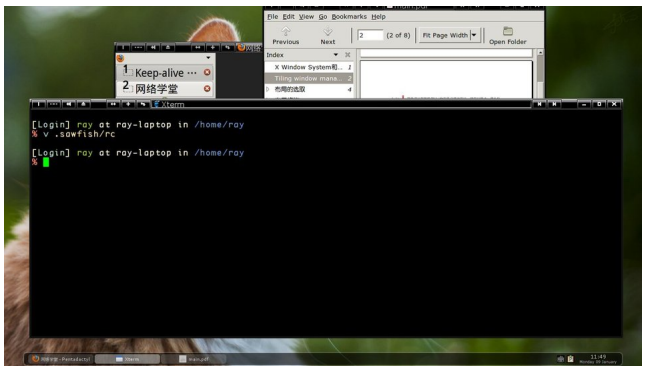

图 1: Sawfish: stacking window manager

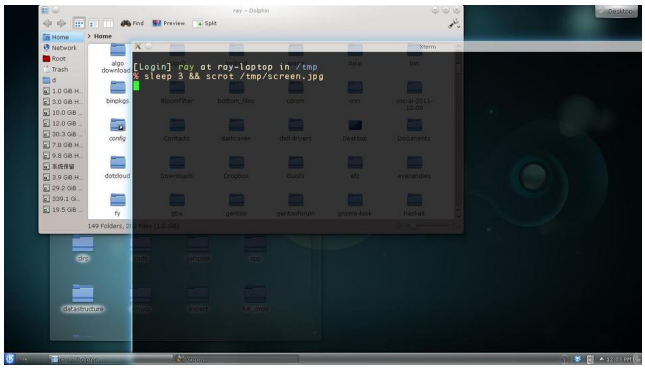

图 2: KWin: compositing window manager

图2是 KDE，其 window manager 为 KWin，可以看到其半透明特效。

- Tiling window manager

  与上述两种 window manager 区别在于，不允许窗口重叠。这类 window manager 通常把桌面划分成不相交的区域来显示各个窗口。通常较少注意窗口装饰 (比如去除常见的窗口标题栏，使用较窄的窗口边框)，铺满整个屏幕，以尽可能大地利用显示器空间。Tiling window manager 一般有丰富的快捷键支持，鼠标的使用是可选的，能提高工作效率。

  Xerox PARC 开发的 CEDAR(1982 年) 是最早的使用这类 window manager 的窗口系统。尽管历史悠久，但 tiling window manager 长期未收到重视，主流 Linux、BSD 发行版的默认桌面环境都不使用 tiling window manager。但近年来，tiling window manager 作为提高

工作效率的强有力武器开始受到越来越多的关注。如 KDE 从 SC 4.4 开始加入 tiling 的基本支持，4.5 加入完整支持。

典型的 tiling window manager 有 XMonad，Awesome，Stumpwm 等。本文着重考虑这类 window manager。

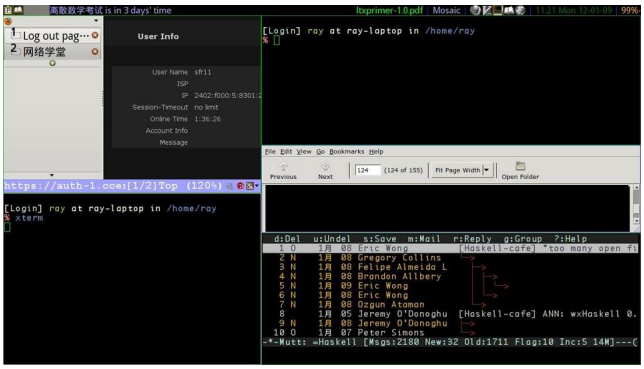

图 3: Xmonad: tiling window manager, 可以看到其窗口标题、边框都被去除了

2 Tiling window manager 的概念

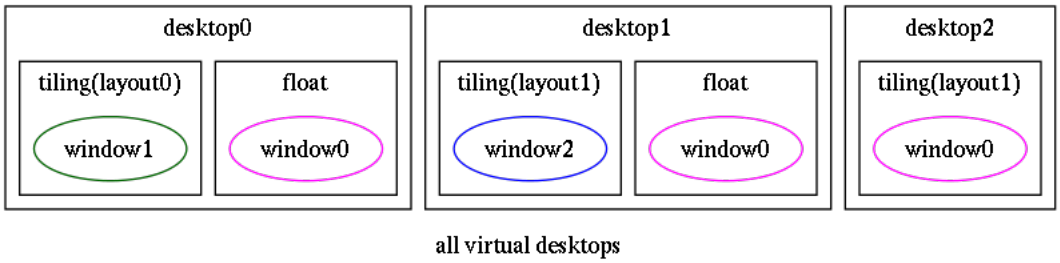

图 4: Relations among some common concepts

如图4所示，tiling window manager 管理了若干 virtual desktop，每个 virtual desktop 都有一个平铺层和一个可选的浮动层。平铺层不允许窗口重叠，设置了一个布局，由用户指定，各个窗口的摆放位置、尺寸由相应的布局算法确定。一般这些布局也会提供一些可调节的参数以增加灵活性。

浮动层则类似于 stacking window manager，允许窗口重叠，位置、尺寸不受平铺层布局算法的影响，比如工具栏、对话框等就是常见的浮动层窗口。

所有 virtual desktop 中有一个正被显示 (有焦点)。所有 virtual desktop 中有一个窗口称为当前窗口，对于具有焦点的 virtual desktop，它的当前窗口就是焦点窗口，会获得用户的键盘、指针等输入信息；对于其他 virtual desktop，当你把它设为焦点 virtual desktop 后，它的当前窗口会成为焦点窗口。[4]

一些 tiling window manager 还有主窗口的概念，每个 virtual desktop 都有一个主窗口，它不一定是当前窗口，但一般具有较大的面积，它所在 virtual desktop 里的其他程序可能都是它的附属程序。

不少 tiling window manager 具有自动分类程序的功能，用户可以根据应用程序名、窗口类名、窗口标题等指定规则，告诉 window manager 应该把相应窗口放在什么 virtual desktop，放在平铺层还是浮动层，指定默认大小、位置等。

3 布局

布局是 tiling window manager 的灵魂。这一部分旨在探讨 tiling window manager 中布局的选取和设计。

3.1 全屏

最简单的布局就是全屏，即只允许显示一个全屏的窗口。这个布局适用于一些办公软件 (如 LibreOffice)、浏览器 (如 Firefox)、制图软件等。由于 virtual desktoop 只显示这一个应用程序，最大化、最小化、窗口移动就不再需要了，相应的窗口标题、边框都可以去除以增加应用程序窗口的可视面积。

3.2 网格状

需要显示较多的窗口 (比如同时显示大量图片) 时选用，把桌面划分成网格状，根据总窗口数来决定显示的列数和行数。边框可以考虑采用单像素细线用于区分不同的应用程序。用户可以调整每一行的高度，每一列的宽度。

[4]这一段只是简要说明 tiling window manager 的作用，为了叙述方便，假设只有一个显示器，所以称当前 virtual desktop 有焦点。

3.3 一个主窗口，多个副窗口

主窗口放在桌面中央，显示面积较大，其余附属窗口显示在主窗口周围。

3.4 简单二叉树结构

类似图论中的 full binary tree [5]，根代表屏幕对应的矩形，叶子表示一个矩形窗口，内部节点则有一根竖线或水平线把该节点对应矩形分割成两部分，这两部分都可用二叉树来表示，成为这个节点的两个孩子。

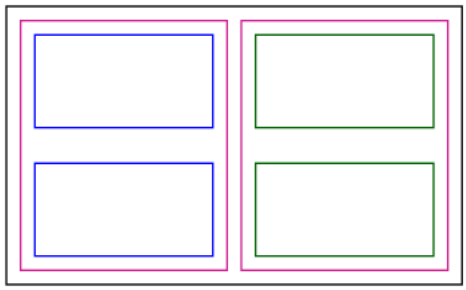

图 5: 四个窗口的二叉树结构

图5中黑色矩形代表整个屏幕区域，被分成了两个区域 (紫红色的矩形)，左边那个紫红色矩形被分成了两个蓝色矩形，右边那个分成了两个绿色矩形。实际上矩形间没有间隙，图中为了清晰加入了间隙。

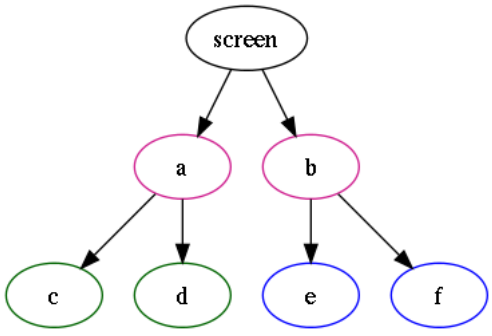

图 6: 图5对应的二叉树

[5]除了叶子外每个节点都有两个孩子的二叉树

图6就是它所对应的二叉树。绿色和蓝色节点表示实实在在的窗口，而内部节点并不实际表示窗口。

下面介绍该布局的算法设计：

一个布局可以抽象为输入屏幕和所有窗口对应矩形 (包含位置和大小信息)，输出作用布局后窗口的对应矩形 (位置、大小) 的算法。假设 virtual desktop 上要显示 n 个窗口 ws ，它们占屏幕面积的比例分别是 $a_0,\ a_1,\ \ldots a_{n-1}$ ，显然有 $\sum_{k=0}^{n-1} a_k = 1$ 。我们设法把这些窗口分成两部分 xs 和 ys ，xs 的面积和与 ys 的面积和大致相等，xs 的面积和与 ys 的面积和之比为 r 。然后把屏幕垂直或水平划分成两部分，两部分面积比为 r 。规定所有 P 中窗口显示在左边 (上边)，Q 中窗口显示在右边 (下边)。然后递归对 P 、Q 所对应矩形做上述操作。下面是该算法的伪代码，调用 layout(rectangle corresponding to the screen, all windows) 即可。

```
function layout(rect, ws)
  if size(ws) == 1
   return { ws[0] => rect } # assign ws[0] the region corresponding to rect
  partition ws into xs and ys
    such that area(xs) and area(ys) are approximately equal
  split rect into (rectl, rectr) by ratio area(xs) / area(ys)
    (horizontally if rect is wide, otherwise vertically)
  return union(layout(rectl, xs)), layout(rectr, ys))
```

其中 partiton 这步相当于一个 knapsack problem，由于窗口不是很多，窗口比例不需要考虑得非常精细，可以直接用伪多项式的动态规划算法求出 xs 与 ys 。窗口比例可以看作常数，即 partition 的时间复杂度为 $O(n)$ 。再假设所有窗口的面积差不多大，那么 xs 与 ys 包含的窗口数会差不多，layout 的时间复杂度可以表示为：

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

由 master theorem 可知：

$$T(n) = O(n \log n)$$

最坏情况下每次 partition 都非常不平均，一边只有一个窗口，那么时间复杂度为 $O(n^2)$ 。

当需要添加窗口时，只要把当前窗口所对应叶子分裂成一个度为 2 的节点和它的两个孩子，分别代表当前窗口和新增加的窗口。需要删除窗口时，把当前窗口所对应的叶子的父节点对应子树收缩成一个节点 (即把当前窗口代表节点和其兄弟合并)。这种布局的优点在于每当添加、删除一个窗口时，只有当前窗口的大小会发生变化，其他窗口则保持不变。

3.5 考虑长宽比的二叉树结构

通常每个窗口除了想要占据的面积，还有长宽比这一参数。上面的简单二叉树结构未考虑窗口的这一需求。这一布局考虑窗口想要的长宽比。对于只有一个窗口的情况，根据其分配到的矩形和它想要的长宽比计算出一个优劣程度。split 时枚举水平分还是竖直分，比较两种方案的优劣程度，选择更好的那个。同样假设每个窗口的面积差不多，那么这个版本的 layout 的时间复杂度可以表示为：

$$T(n) = 4T(\frac{n}{2}) + O(n)$$

由 master theorem 可知：

$$T(n) = O(n^2)$$

最坏情况下每次 partition 都非常不平均，一边只有一个窗口，时间复杂度将达指数级。但如果我们限制只在前层枚举水平、竖直分，之后则直接贪心地选择一种划分方案，那么时间复杂度就与上节的算法相同，不会出现指数时间复杂度情形。

```
function layout(rect, ws)
  if size(ws) == 1
    calculate badness according to ratios of ws[0] and rect
    return ({ ws[0] => rect }, badness)
  partition ws into xs and ys
    such that area(xs) and area(ys) are approximately equal

 split rect horizontally into (rectl, rectr) by ratio area(xs) / area(ys)
  (ml, bl) = layout(rectl, xs)
  (mr, br) = layout(rectr, xs)

 split rect vertically into (rectu, rectd) by ratio area(xs) / area(ys)
  (mu, bu) = layout(rectu, xs)
  (md, bd) = layout(rectd, xs)

  if bl+br < bu+bd
    return (union(ml, mr), bl+br)
  else
    return (union(mu, md), bu+bd)
```

4 布局修饰器

布局修饰器本身并不是布局，但可以作用于已有布局之上来为它们添加新功能，通常和具有应用的布局无关。所以，布局修饰器表示一类通用的功能，可以看作在布局算法应用前对输入 (窗口对应矩形) 进行变形，应用后对其输出结果进行变形。如果把布局看做 monad，那么布局修饰器就是 monad transformer(事实上，这就是 Xmonad 的实现方式)。通过一个包含大量布局修饰器的库，我们可以自由地选取一些来创造适于自己使用的新布局。

4.1 放大焦点窗口

对于具有焦点的窗口，把它对应的矩形显示区域扩大，保持中心不变，同比例增大长和宽。好处是能突出当前正在交互的应用程序。

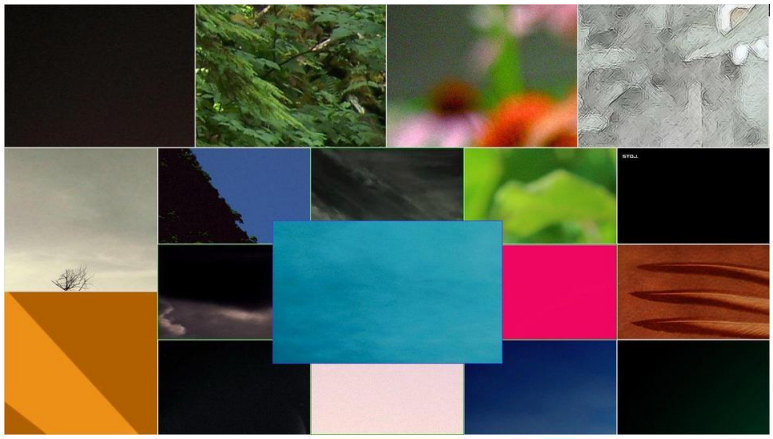

图 7: 放大焦点窗口

4.2 翻转

比如以桌面中心所在水平线为轴，调换线上下方的窗口。

4.3 标签

为窗口添加标签，用于显示该应用程序的标题。特别地，当一些窗口不可见时，可以显示它们的标签，与覆盖它们的窗口的标签显示在一起，这样用户可以方便地点击标签来切换应用程序。

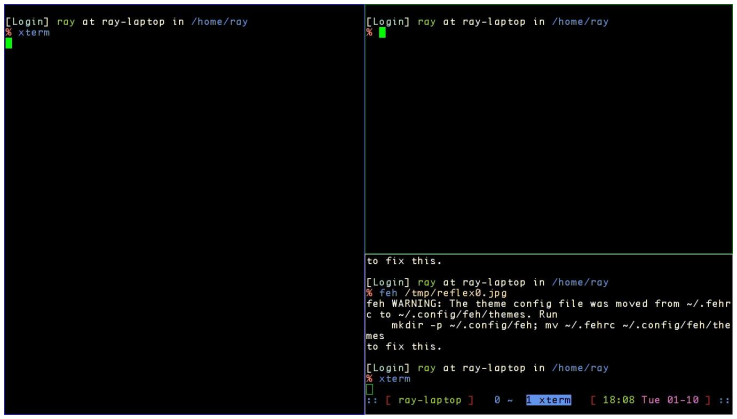

图 8: before reflex

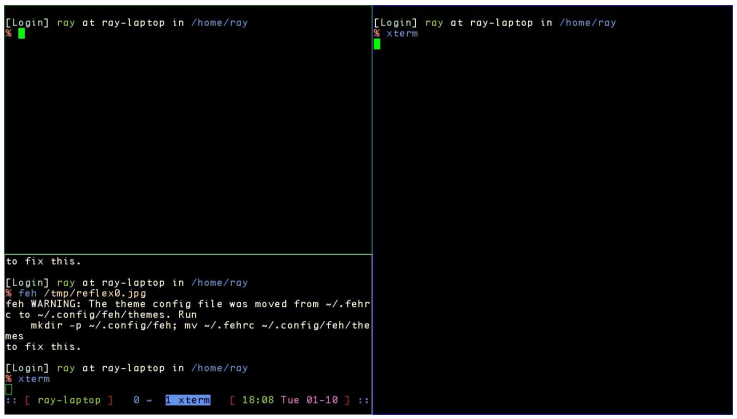

图 9: after reflex

4.4 间隙

在窗口周围添加大小可调整的间隙。

4.5 复合布局

有时候，单一的布局是无法满足需要的，这时可能需要结合多种布局。比如 Gimp [6] ，它会显示很多工具条，而可能又有多幅图片需要处理。一个比较好的方案是把桌面区域分成两部分，一部分采用全屏布局处理图片，工具条则采用网格状布局。

[6]GNU Image Manipulation Program，一款图像处理软件

4.6 交换焦点窗口和主窗口

通常主窗口显示区域较大，当需要切换焦点时，不改变窗口的排列布局，只是把获得新焦点的窗口和主窗口所代表应用程序交换。好处是焦点窗口始终是主窗口，具有较大的显示区域。

4.7 最大化

临时最大化一个窗口，需要时还原原来的桌面布局。

4.8 最小化

临时最小化一个窗口，需要时在原位置重新显示。

5 焦点的切换

5.1 前驱后继

通常 window manager 为每个 virtual desktop 储存了一个线性表储存所有窗口，我们可以设置按键来把焦点设置为焦点窗口的前驱或后继。这相当于传统的 Alt-Tab 切换窗口的方法。

5.2 jump-or-exec

这是 Sawfish 的一个概念，就是说触发某个快捷键，倘若相应的应用程序启动了，就把焦点给这个应用程序，否则打开这个程序并让它获得焦点。一些经常使用的程序可以为它们映射 jump-or-exec 的快捷键。

5.3 基于方向切换焦点

我们在切换布局时，需要考虑到两点，一是直观，二是方便。很明显，当窗口较多时，用基于前驱后继的切换方式会非常麻烦，而且会跳转到哪个窗口也不可预测。而 jump-or-exec 也不是一个通用的解决方案。下面介绍两种基于方向切换焦点的方法。

5.3.1 直观方法

一个比较自然的方法是假想当前焦点窗口往右面延伸 (即高度不变，左上角在桌面上的坐标不变，宽度增大)，第一个碰到的窗口定义为该窗口*右面最近的窗口*，为之设置一个快捷键。同样的，对左面、上面、下面也分别设置快捷键。该方法比较自然，但是无法保证在所有布局都能遍历到所有窗口。

但可以证明，在树形结构的布局里，该方法能遍历所有窗口，即所有窗口联通。简证如下：只有一个窗口时显然联通。假设窗口数 $< n$ 时联通，则当窗口数为 n 时，不失一般性，不妨设有竖线把整个矩形分成了左右两部分，两个子矩形的窗口数都 $< k$，由归纳假设，子矩形内部所有窗口是联通的，而左边子矩形中必有窗口的*右面最近的窗口*在右边矩形里，所以左边可以和右边联通，同理右边也可以和左边联通。

5.3.2 L1 norm distance

只考虑每个窗口所对应矩形的中心。问题可以抽象为在平面上有若干点，定义四个方向上的“最近”关系。使得得到的有向图中，所有点在一个强联通分支内。下面定义满足这个条件的“最近”关系：

令 $O(x, y)$ 为当前窗口对应矩形的中心，对于另一个窗口对应矩形的中心 $A(x', y')$，定义它们的距离 $L1(O, A)$ 为 $\max(|x - x'|, |y - y'|)$。S 为除当前窗口外所有窗口的中心的集合。这里采用的横轴正方向为右，纵轴正方向为下。

$O(x, y)$ *左上最近的点* 定义为：

$$A(x_A, y_A) \qquad A \in S',\ for\ \forall B(x_A, y_B) \in S',\ L1(O, A) \leq L1(O, B)$$

其中

$$S' = \{B(x_B, y_B) \mid B \in S,\ x_B \leq x,\ y_B < y\}$$

类似地，可以定义其他三个方向上最近的点。

右上最近的点 定义为：

$$A(x_A, y_A) \qquad A \in S',\ for\ \forall B(x_A, y_B) \in S',\ L1(O, A) \leq L1(O, B)$$

其中

$$S' = \{B(x_B, y_B) \mid B \in S,\ x_B > x,\ y_B \leq y\}$$

右下最近的点 定义为：

$$A(x_A, y_A) \qquad A \in S',\ for\ \forall B(x_A, y_B) \in S',\ L1(O, A) \leq L1(O, B)$$

其中

$$S' = \{B(x_B, y_B) \mid B \in S,\ x_B \geq x,\ y_B > y\}$$

左下最近的点 定义为：

$$A(x_A, y_A) \qquad A \in S',\ for\ \forall B(x_A, y_B) \in S',\ L1(O, A) \leq L1(O, B)$$

其中

$$S' = \{B(x_B, y_B) \mid B \in S,\ x_B < x,\ y_B \geq y\}$$

假设我们要从 O 移动到其左上点 B (即 $x_B \leq x,\ y_B < y$)，选取的最近点为 A 。由 L1 norm distance 性质易得：

$$L1(A, B) \leq L1(O, B)$$

如果 $L1(A, B) = L1(O, B)$ ，那么先移动到 A ，再从 A 选取最近点 C ，就有 $L1(C, B) < L1(A, B) = L1(O, B)$ 。

即每两次移动，当前点到目标点的 L1 norm distance 严格单调递减。而所有可能的 L1 norm distance 即所有点对的 L1 norm distance 是有限的，所以最终 L1 norm distance 必会减少为 0 即到达目标点。所以该方法保证了从任一窗口出发，都能移动到任一其他窗口。由于根据左上、右上、右下、左下最近点的移动方式不够直观，可以考虑把坐标轴旋转 $\pi/8$ ，即变成根据左、上、右、下最近点的移动方式。

6 总结

Tiling window manager 作为一种古老而又新颖的窗口管理方式，给人们的工作带来了很多的方便。希望本文能激发读者兴趣，关注、使用这类 window manager。Tiling window manager 还有很多特别的应用，比如用来代替集成开发环境 (integrated development environment)，代替多文档界面 (multiple document interface) 的程序等。布局的选取是使用 tiling window manager 应用的灵魂，如何进一步完善上面介绍的布局算法，使其更加智能化，加入对窗口类型的判断、对用户习惯的考虑，还是有待继续研究的方向。

参考文献

[1] *X Window System User Documentation*, ***http://www.x.org/wiki/UserDocumentation***

[2] David Rosenthal, Stuart W. Marks, *Inter-Client Communication Conventions Manual*, `www.x.org/docs/ICCCM/icccm.pdf`

[3] Norbert Zeh, 2011, *Navigation2D: A Directional Navigation Module for XMonad*

[4] *Xmonad Documentation*, `http://xmonad.org/documentation.html`