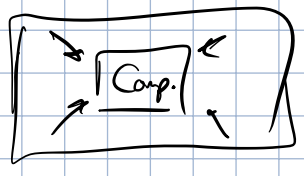
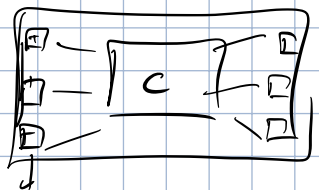
History Of Interaction

1945 — 1965 — 1980 — ~1981

Batch
- Cards → computer ↳ instructions.
- Barely any interaction
- Highly trained



Conversational
- Terminals



Only I/O, no comp.
- CLI ⇒ memorize (Bash)
- Not friendly, explorable
- Highly trained indv.

Interactive Interfaces
- Require user input ⇒ text
- Ex: // C++ calc. on CLI

GUI

Graphical User Interface

History:

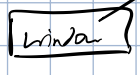
1st: Xerox Star Info. System (1981)
↳ Founder of GUI, expensive

Popularization ⇒ Apple Macintosh (1984)

Defn:
1. High resolution + high refresh graphics
2. Keyboard (inc. touchscreen)
3. Pointing device

Common fts:
- 2D + 3D graphics
- Animation
- Videos
- Point + click
- Text only

Another name ⇒ WIMP system



, icons, menus, pointer

Windows & Windowing System

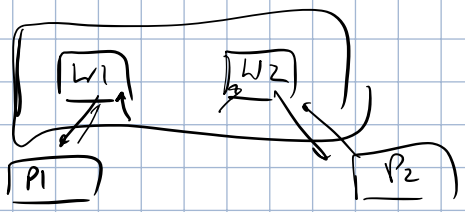
Imp. innovation of GUI

Benefits:
- Independence of apps. (apps don't need to think about each other)
- Multiple apps

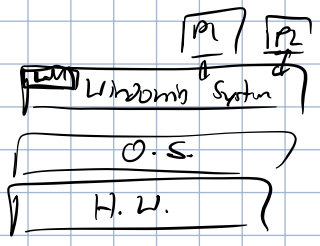
Windowing system: provides window interface.

↳ Responsibilities:

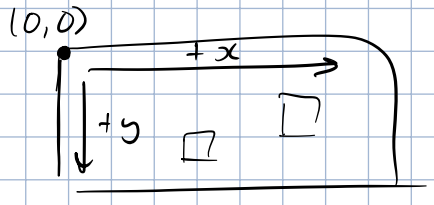
1. Provide I/O routes into windows.



2. CRUD windows.
3. Access to graphical hardware.

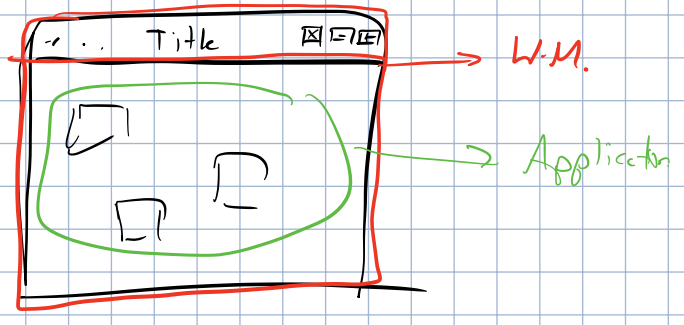


↳ We don't care about create → W.S. will take care



Windowing manager: $\subseteq$ W.S.

- Look + feel of window.



- Our app. can interact w/ W.M. API to setup window
- Why separate?
  ↳ W.S. is very modular ⇒ components can change!

GUI INTERACTIONS

How to use & benefits

GUI interaction — point + click (extensible → drag, rotate, drop)

Benefits:

1. User has control at all times.
   ↳ System, not user, is waiting
2. Recognition > recall

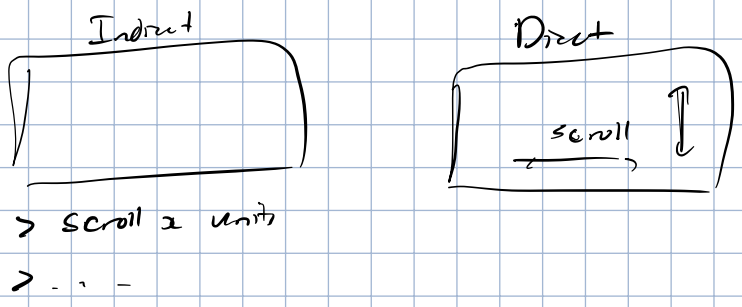
↳ Common interfaces → exploration

3. Real-life metaphors
   ↳ Mimics real-life ⇒ easy to learn
   ↳ Ex:// desktop, folder, drag + drop

Direct Manipulation (DM)

Defn: manip. virtual obj like real-life physical obj.

First: Sutherland's sketchpad

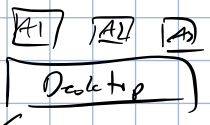


DM deals w/ task obj. (thing to manip.)

Guidelines for DM:

1. Objects should be cts. + persist.
2. Manip. by physical action.
3. Fast, incremental (optionally, reversible) — not DM, but usch
4. Self-learning

Benefits: Users focus on task, not how to do task



} DM applies

Doesn't use DM as much (eg. pulling folder to open)

This is not a requirement for GUI

When to not use:

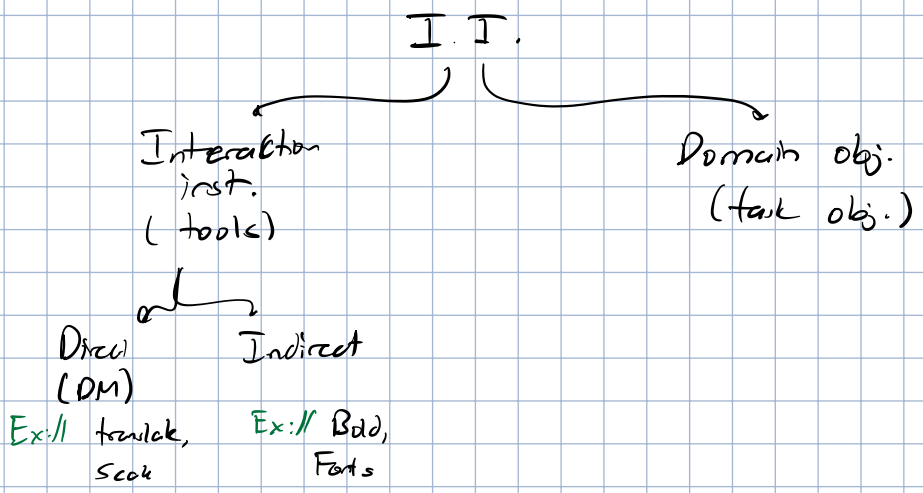
- Accessibility
- Takes up space
- Unclear what metaphors
- Keyboard ↔ mouse switch is v. annoying

Interaction Models

Defn: principles + guidelines about design
↳ Help evaluate designs

1 model ⇒ instrumental interaction

Instrumental Interaction

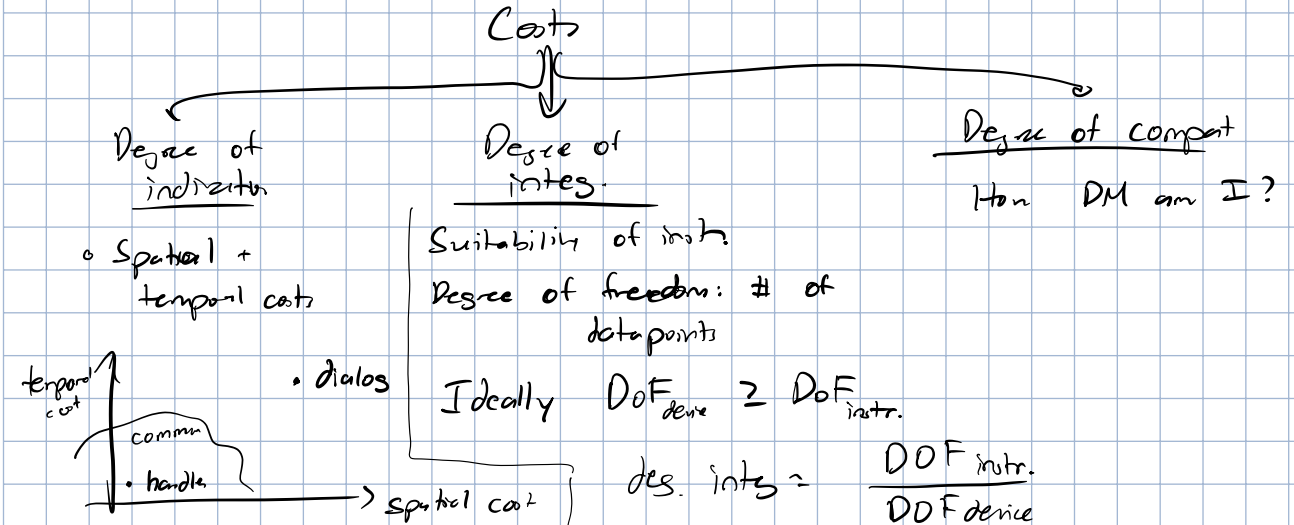


Instruments must be activated
↳ Spatial activation: movement cost (how much to move)
↳ Temporal: temporal cost (time to manip.)

Reification: concept → concrete design
↳ Ex:// Copy Cut

Meta-instruments: instr act on other inst.
↳ Ex:// Notability color Switch

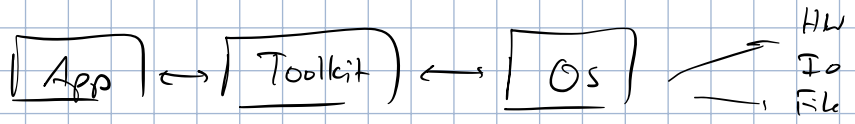
Eval. Instr. on I.I. Model

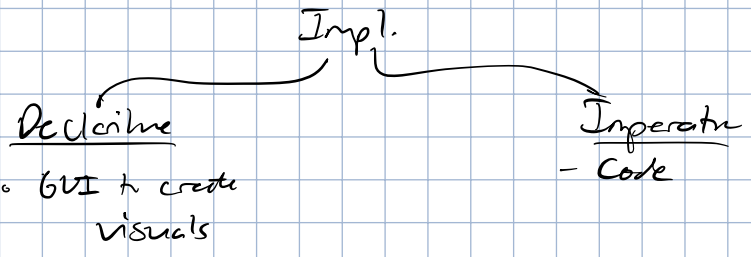


Good DM: low des. of indir, high des. of inter, high des. of compat.

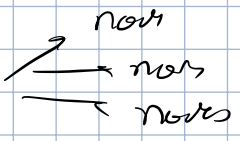
JavaFx

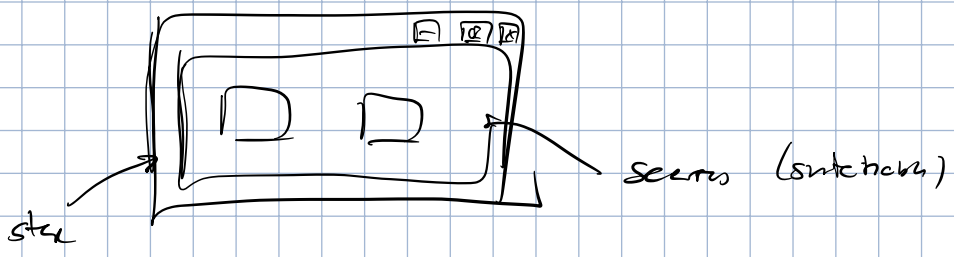
GUI Toolkit



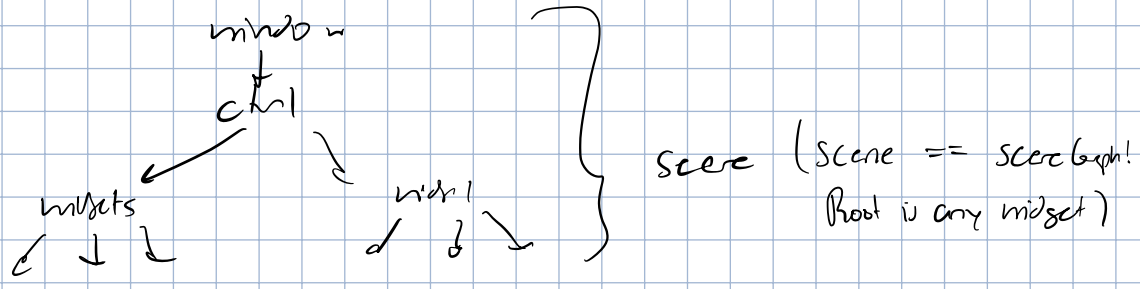


- Every app has a stage → scene





- Scene graph:

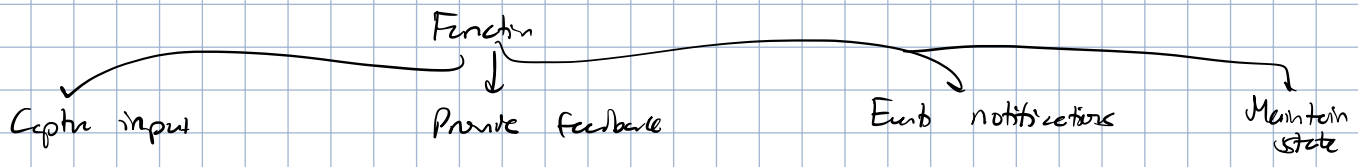


Desirable factors:

1. Completeness
2. Consistent
3. Customizable

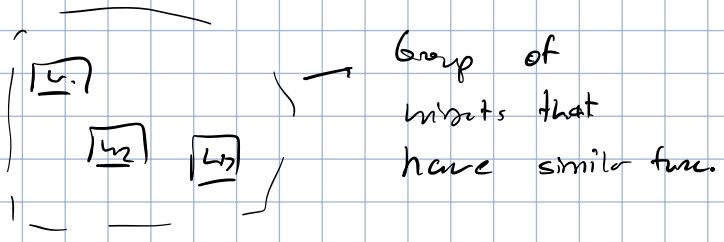
WIDGETS

Defn: reusable comp.



Descendants of Mac Toolkit (& original)

Logical input devices



Compon of

1. State

2. Events

Widgets are instances of log. dev. ⇒ appearance

Properties

Defn: values that deter state + function

Can listen / bind

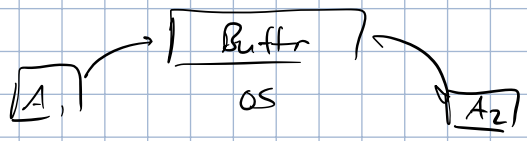
↓
event

or property links to another

↙ ↘
unidirectn. bidirectn.

Moving data

Clipboard



Drag & drop

1. Mouse drag listener

⇓

funct

Issues:

1. Security

2. Translate

Sol:

Ph & S define data formats.

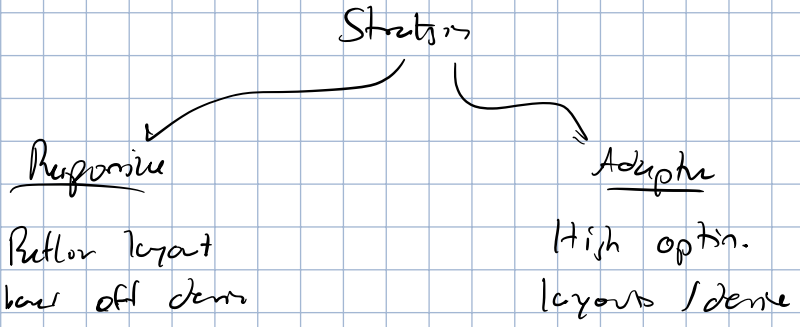
2. Receiving app checks if valid

3. Trust

Layout

Dynamic layout

Can't guarantee window size ⇒ resize & reposition app



Rust also has 3 goals:

1. Max { window space }

2. Consistent

3. Visual quality

D. P.: composite & strategy

↳ Parent will call also on children to resize (strategy, composite)

composite D. P.

Rely on min, max & preferred

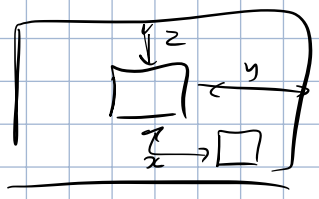
Layout strategies

1. Fixed positioning: exact (x, y) coords.

2. Variable intrinsic: 2 pass

Gets all children's sizes pretend → pick layout that works best for everything.

3. Constraint / relative layout:



4. Custom

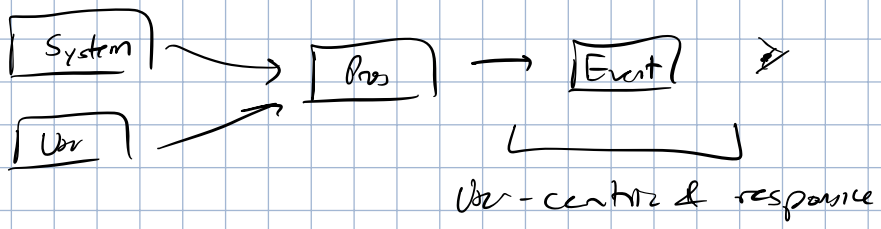
Implementing

Either imperative (code), declarative (CSS / GUI / XML)

JavaFx → Scene Builder

Recursion & nested layouts are good strategies!

EVENTS



Toolkit have abstractions for events

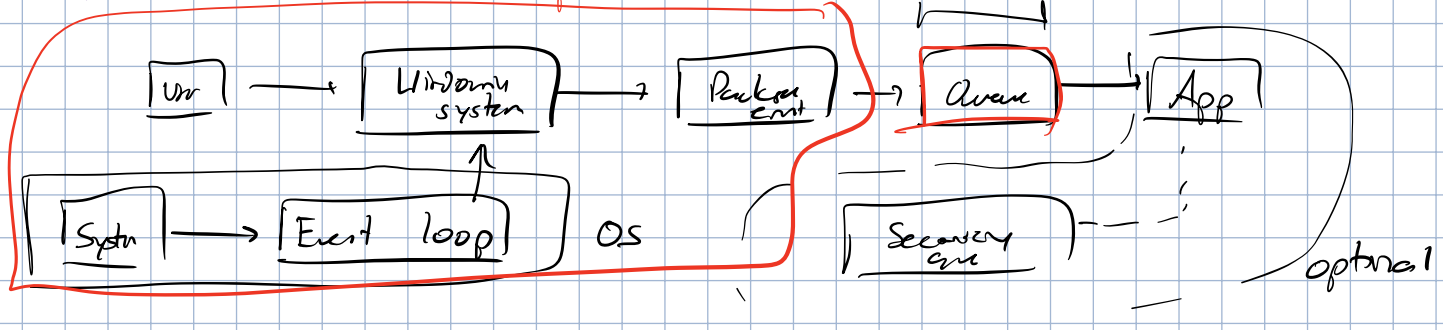
```
Event {
  target
  src
  isConsumed
  ;
}
```

Dispatch → Handle

Implementation.

Capture

ordered by time.



Dispatch

Based on scene graph

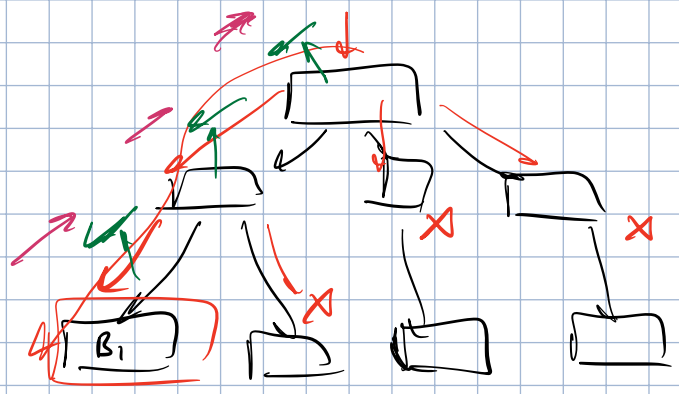
Steps:

① Figure out target (which node should receive) ← area of click

④ ① Route event to target

③ ③ Capture event by traversing down path

② ② Bubble event by traversing up path



Note:

Traverse scene graph

Lightweight?

Y → Toolkit

N → Window System.

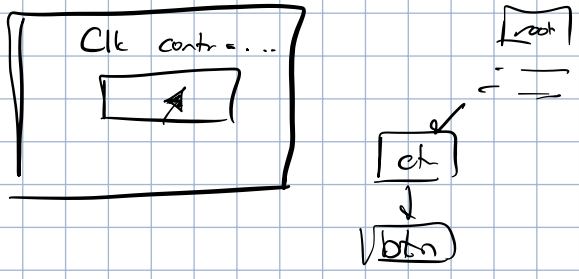
Step ① & ②:

Often based on area of click / focus / center point ⇒ target node.

Step ③ & ④:

Capture vs. Bubble

Q: why allow respond throughout path



Q: why both?

Time: before goes to node vs. after

Capture: event filter

Bubble: event handler

Dispatch

Positional

cursor pos.

Focus

privileged input (keyboard)

(item in focus)

Handling

XII: switch handling:

```
switch (event):
  case ___:
  ___ => behavior
  case ___:
  ___
```

} HUGE — hard to maintain.

Early Java: inheritance bindings

Lingo: is inheriting an abstraction ⇒ custom code to handle.

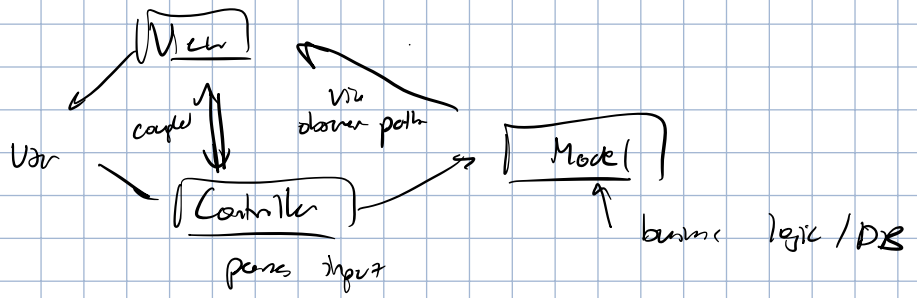
Ls Tech

Kotlin: listeners.

Register handlers into app ⇒ call as necessary

MODEL VIEW CONTROLLER

Design pattern:



Developed in 1979 at Xerox PARC

Benefits:

① Separation of logic

② Multiple views / model

③ Code separation

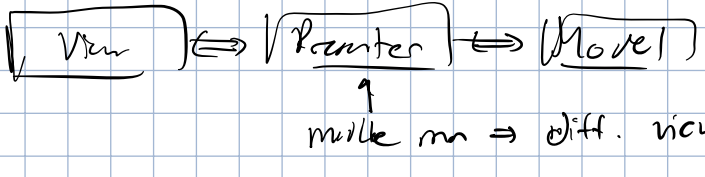
Cons:

① View handles both input & output

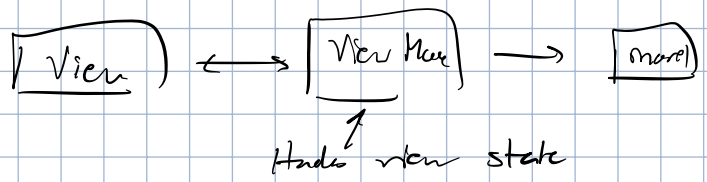
② Diff. models

Variants:

a) Model view presenter

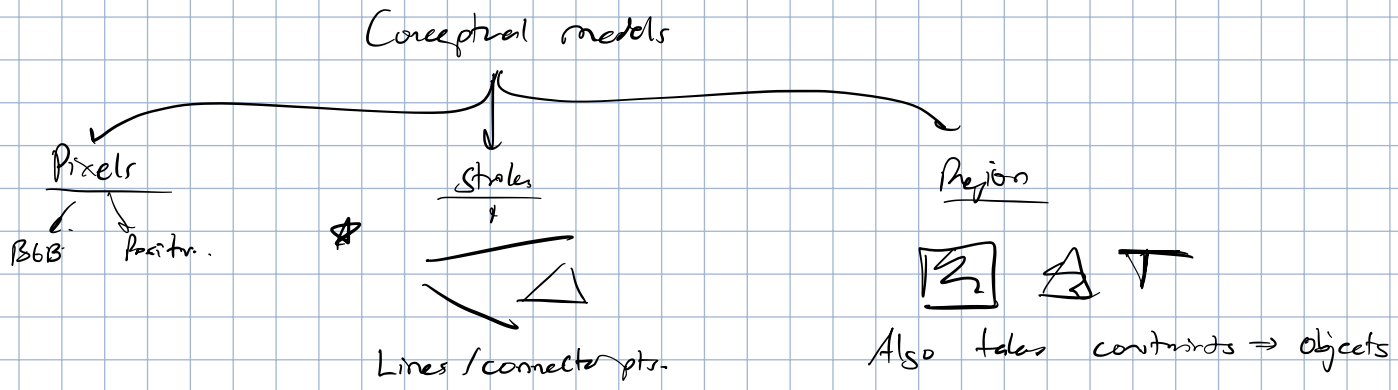


b) Mode-view - View Model



DRAWING

Drawing Models



Graphics context:

{
- - - } Properties
- - - } Styles
}

① Grab the GC
② Change pcb of GC using API
③ Render

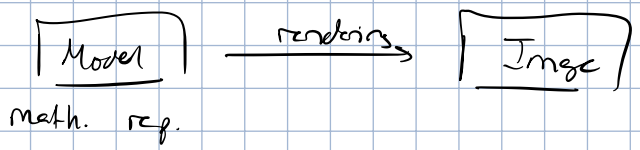
- In JavaFx: Canvas object → graphics context (gc = canvas.getGraphicsContext())

Painter's Algorithm

→ render from top→bottom
back → front

① Find primitives
② Encapsulate in classes
③ Ordered display of classes
④ Draw T → B, B → T

GRAPHICS



Shape Model

Primitive shape model
- $\{p_1, \ldots, p_n\}$
- Properties: isFilled, isClosed, ...

Apply transformations:

1. Translation:

$$x = x + t_x$$
$$y = y + t_y$$

2. Scaling:

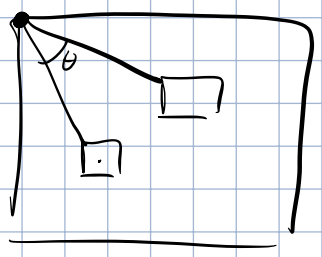
Uniform:	Non-uniform
$x = sx$	$x = s_x x$
$y = sy$	$y = s_y y$

Combine → order matters.

3. Rotate:

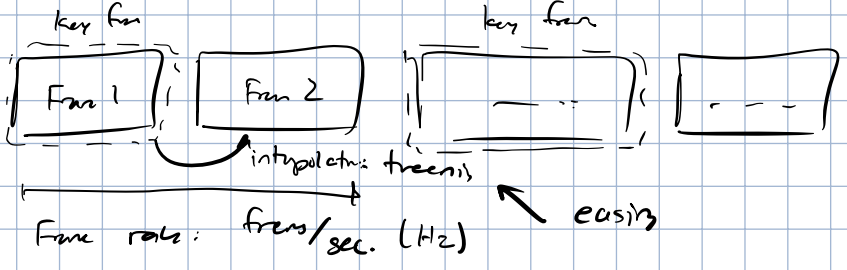
$$x = x\cos(\theta) - y\sin(\theta)$$
$$y = x\sin(\theta) + y\cos(\theta)$$

JavaFX has built-in transform., cascade to children
- Origin:



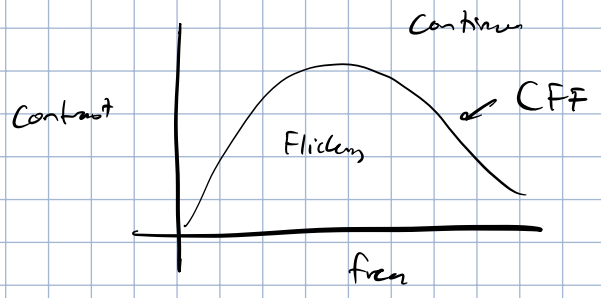
ANIMATION

Terminology



Impl: choose drawing parameters / time click

Critical Flicker Freq:



Timer

Basic:

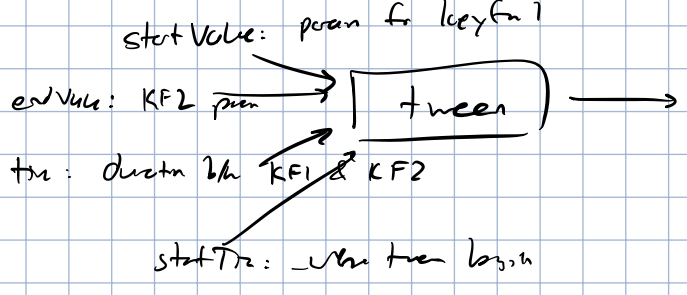
```
timer = Timer (duration) {
        ① Animation parameters.
        ② Redraw                    ⇒ Will occur "duration" ms.
        ③ Timer restart
}
timer.start().
```

UI frameworks are single-threaded: simpler

↳ All timer must be on the thread of UI framework (cannot use Java timer)

↳ Must be from JavaFX lib

Tweening



Calculation:

① Proportion of the that has passed:

$$t = \frac{currentTime - startTime}{time}$$

② Interpolate value:

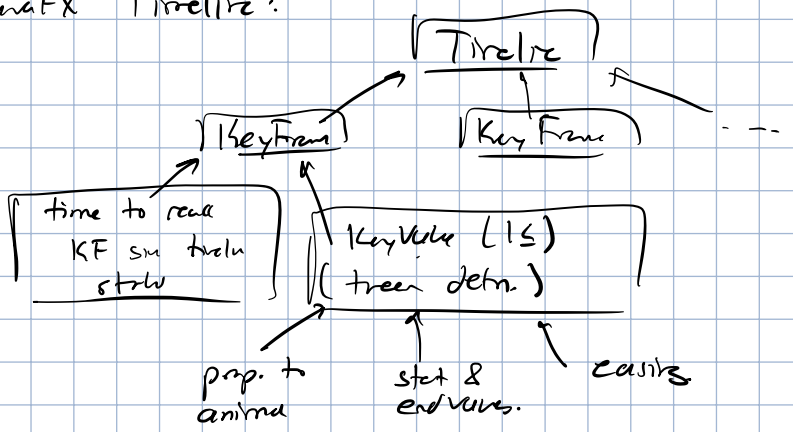
$$value = startValue + (endValue - startValue) * t$$

} lerp(start, end, t)

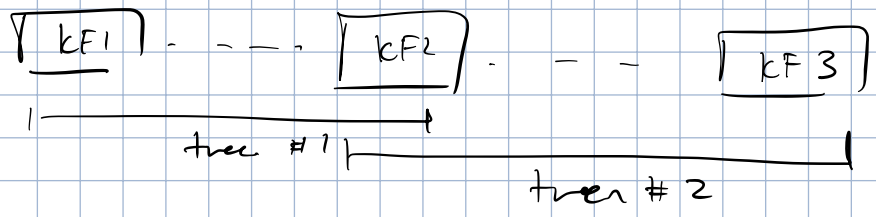
Easing: change t param over time.

- easeIn, easeOut, easeInOut

JavaFX Timeline:



Chain tweens → complex animations.



- Timeline will keep track of tween progress.

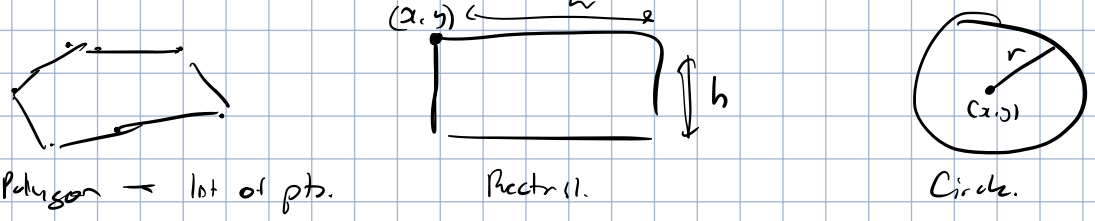
Parallel tweens if not changing same param.

In-built transitions in JavaFX

HIT-TESTING

Shape Models

Diff. shapes → diff models.



Polygon – lot of pts. Rectangle. Circle.

Hit Tests Paradigms.

Click has touched object

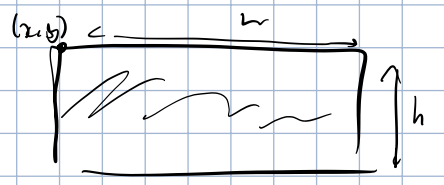
B/C diff models → custom hit tests for models.

↳ Fill/no fill, stroke thickness....

Function defn:

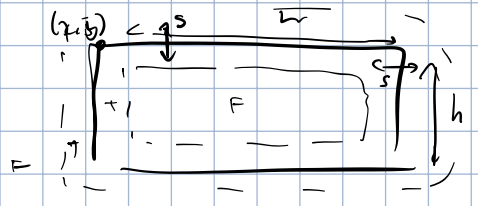
fun hittest (mx, my) : Bool

Ex:// Rectangle:



$\Rightarrow$ hittest $(x,y) \rightarrow (x,y) \in [x+w,\ y+h]$

Ex:// Empty rect.



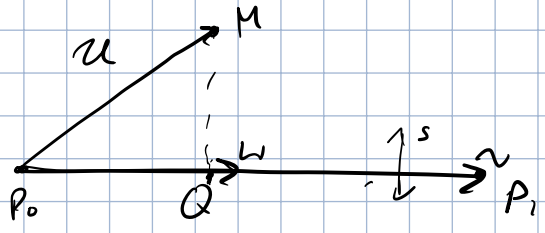
hittest $(x,y) \rightarrow$

$$\left.\begin{array}{l} x \in [x - s/2,\ x + w + s/2] \\ \& \\ y \in [y - s/2,\ y + h + s/2] \end{array}\right\} T$$

$$\left.\begin{array}{l} x \in [x + s/2,\ x + w - s/2] \\ \& \\ y \in [y + s/2,\ y + h - s/2] \end{array}\right\} F$$

Other hit tests:

- Circle fill test: $\|(mx, my) - (c_x, c_y)\| \leq r$
- Circle edge hit: $r - s/2 \leq \|(mx, my) - (c_x, c_y)\| \leq r + s/2$
- Line edge hit



① Project $M \rightarrow Q$

$$w = \frac{u \cdot v}{\underbrace{v \cdot v}_{s}} \sim$$

If $s < 0$ or $s > 1 \Rightarrow M$ out of bounds $\Rightarrow$ fail

O.W. $\underline{Q = P_0 + w}$

② hit test:

$$dist(M, Q) < s/2$$

- Polygon edge test: line edge test for each edge.

Implementation

Tip:

- Avoid $\sqrt{\ }$ in calc.
- Bounding boxes for shapes
- Use cell split to check if mouse is clicking occupied cell

JavaFX: shape.contains $(x, y) \Rightarrow$ hit test

GRAPHICS TRANSFORMATIONS

Matrix Transformations

Efficient

Homogenous coords:

$$(x, y) \xrightarrow{\text{vectors}} \begin{bmatrix} x \\ y \\ \alpha \end{bmatrix} \qquad \alpha = \begin{cases} 0 & \text{if vector} \\ 0.\text{w.} & \text{if point} \end{cases}$$

① 2D translation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

② Scale transformation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x \cdot x \\ s_y \cdot y \\ 1 \end{bmatrix}$$

③ Rotation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ 1 \end{bmatrix}$$

Affine transformation matrix (3×3)

↳ Combo of transform → affine matrix (multiply constit. transforms)

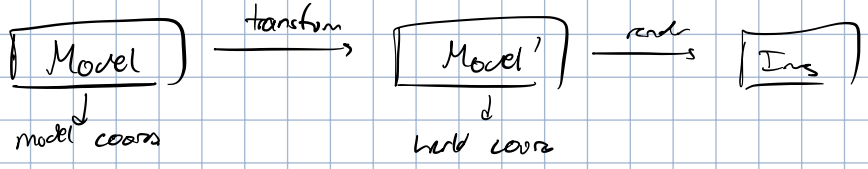
Eg:

$$P' = \underbrace{T(t_x, t_y) \cdot R(\theta) \cdot S(s_x, s_y) \cdot P}_{\text{multiply \& order of transform}}$$

Order matters.

Tip: to do proper rotations
1. Send obj to origin
2. Rotate
3. Translate

Shape Model



In JavaFX: stack transforms:

```
Shape {
    var transform = Transform2D()
    fun stackTransform (t: Transform2D) {
        transform = transform.multiply(t)
    }
}
```

INPUT

Input Devices

Classify:
- Degrees of freedom
- Continuous vs. discrete.
- General vs. specific (tasks)

Text Input

Based off Qwerty:
- History: paper writing → prevents jams
- Cons: left hand dominant & awk. combos.

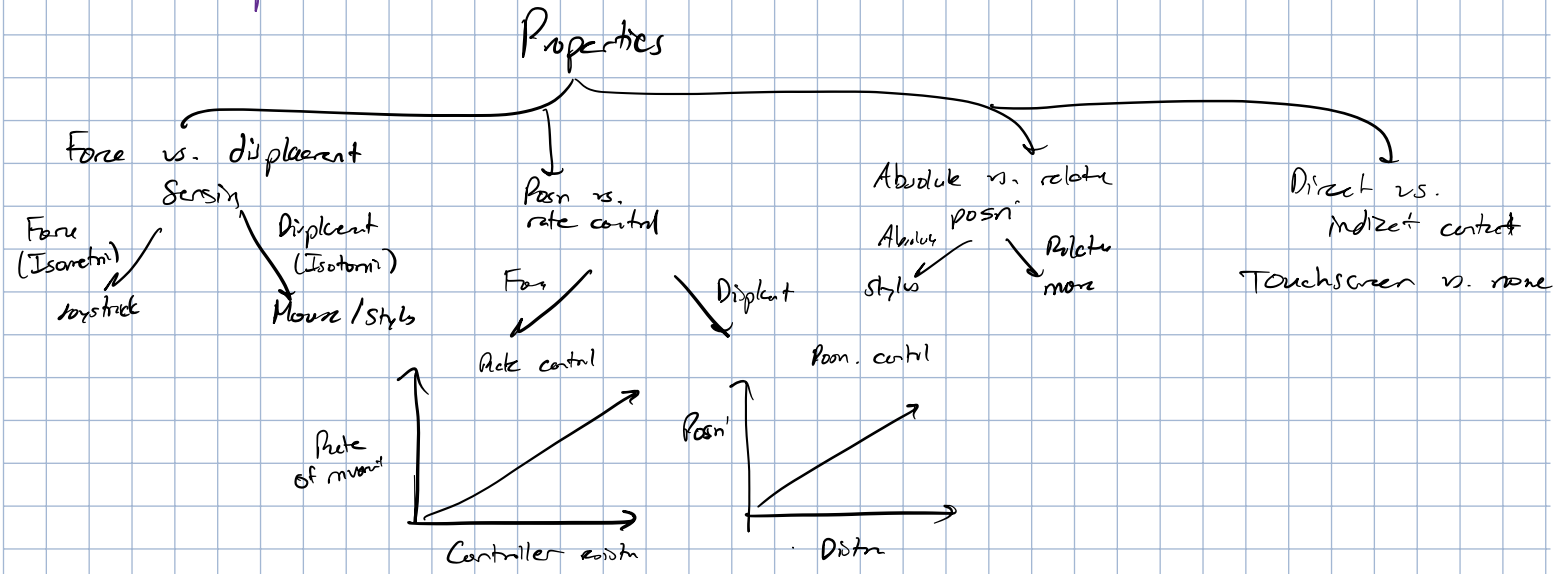
Keyboard improvements:
- Mobile friendly: smaller, travel distance ↓ = quality of typing lower.
  - ↳ Combat: predictive typing (T9), not usable
  - ↳ Touch
  - ↳ Gestural / natural handwriting
  - ↳ Chording: multiple keys → one letter (fast, portable)

Encoding:

- ASCII (1 byte) → 255 char.
- UTF: superset ASCII → more char (more bits)

Validation of text input

Positional Input



Clutching: disconnect input temp. → change posn.
- Lifting mouse.
- Necessary for relative posn.

Control display gain:

$$\frac{V_{cursor}}{V_{device}} \longrightarrow$$ can be changed dynamically to reduce clutching.

MOBILE U.I.

Device Capabilities

Implications:
- Limited resources
- Single app active
- Small screen
- Touch input

Touch input:

Pros:
- Absolute & direct manip
- Gesture & pressure trackers.

Cons:
- Not high precision
- Noisy → occlusion
- No hover state

Multi-Touch

How to fire an event if multiple fingers pressing control?

Option 1: fire event on last contact lift

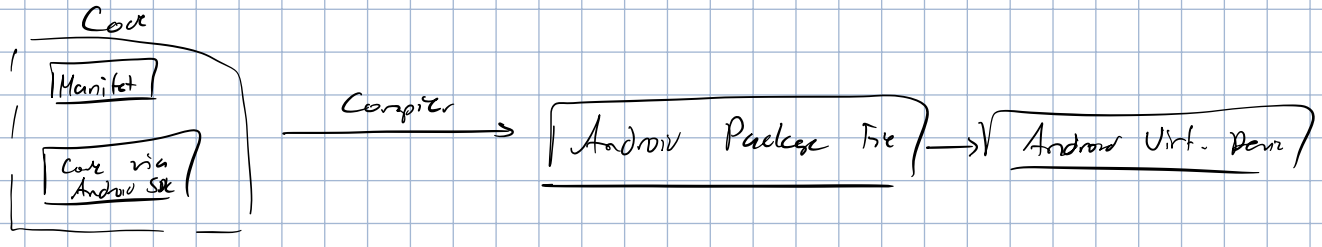
Option 2: fire event entire control touched

Option 3: over-capture (handle multi-touch like 1 input)

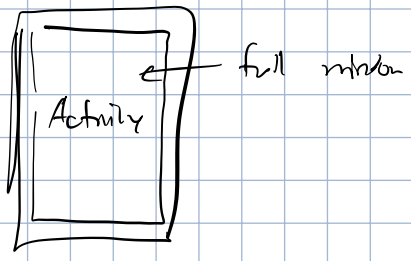
ANDROID

Dev Fundamentals

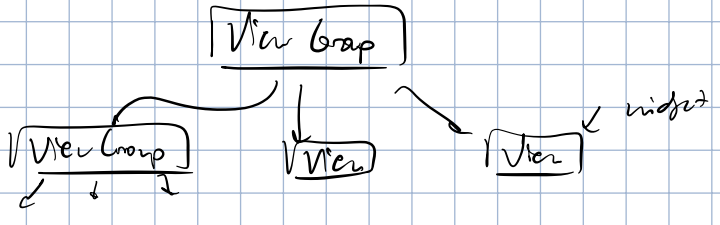
Linux arch w/ each app has distinct profile & process. OS handles entire lifecycle



Main component: Activity.



Scene graph:



Designing layouts declaratively: XML fn:

Renderscript for Android

RESPONSIVENESS

Responsive UI: gives feedback in timely fashion

How?

① Design

② Load data quickly

Most important factor in user satisfaction.

Responsive design ≠ fast app

Examples:
- Confirming action
- Anticipating req.
- Preview times.
- Progress indicators
- Prefetching / progressive loading
- Skeleton screens.

Long task

Blocking model:
- Run entire task on UI thread
- Blocks UI → not good

Subtasks
- Split into subtasks
- Run on UI thread periodically
  - Platform. runLater
- Good for single thread apps & can pause
- Bad for task time prediction & hard to break down to subtasks.

Multi threads
- App has app, UI & worker threads
- Run tasks in worker
- Use @ synchronized for monitor in crit. section
- Pros: multi-core, easy
- Cons: thread safety

UNDO - REDO

Intro

Benefits:
1. Recover from errors
2. Exploration
3. Evaluate modifications

Considerations:
1. What should be undoable?

General principle: Doc changes should be undoable, ask for confirm before undoing hard task.

2. What part of UI should be restored?

3. How much should be restored.

General principle:

A: Direct manip actions are too small

B: Chunk all changes to single event

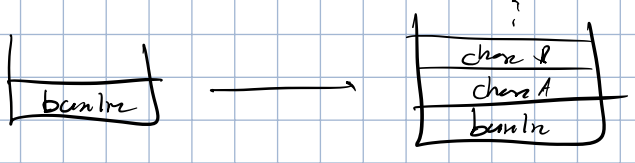
C: Delimit chunks by input breaks

4. Scope: global / local undo?

Implementation

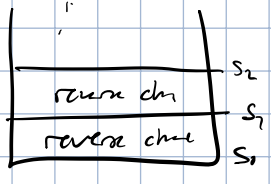
Undo implementation:

(A) Forward undo



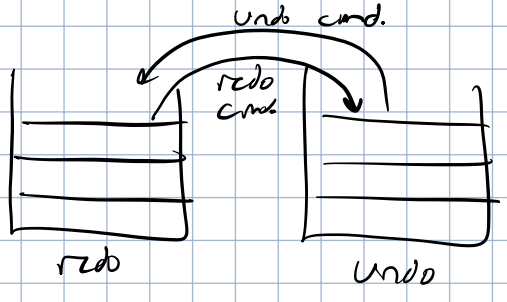
To undo, don't apply last change made.

(B) Backward undo



To undo, apply last reverse change.

Need undo & redo stacks:



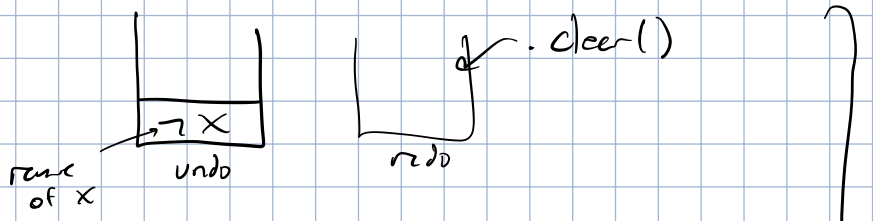
Q: How to encode changes in stack?

(A) Memento pattern: data struct. of state

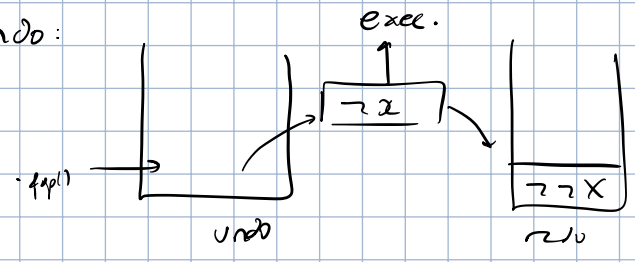
(B) Command pattern: store cmd. to exec. state change. ⇐ Java & Kotlin

Also of command pattern:

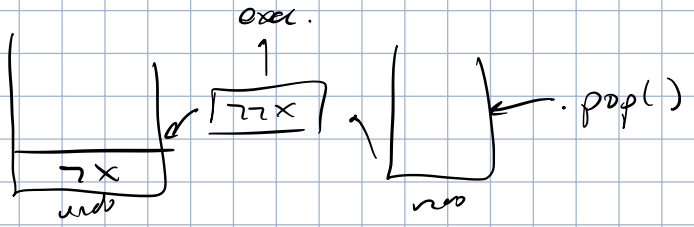
① On some command X:



② On undo:



③ On redo:



In Java & Kotlin, use UndoManager.

Input Performance

Keystroke level Model (KLM)

Every task composed of following:

① Keystroke (K)
② Point (P)
③ Button press (B)
④ Hand movement (H)
⑤ Mental prep. (M)
↳ Finding decisions, ..., usually noise

Each has own time estimate.
⇓
Sum up these for subtasks to find total amt. of time for task

Pros:
+ Easy to model
+ Mockups

Cons:
- Outdated
- Variable timings.
- Simplistic

Fitt's Law

movement time $\propto D/S$ ← D: distance to obj., S: size of obj.

$$MT = a + b \log_2\left(\frac{D}{W} + 1\right)$$

distance → index of difficulty

a, b: constants

input characteristics.

Index of performance

$1/b$

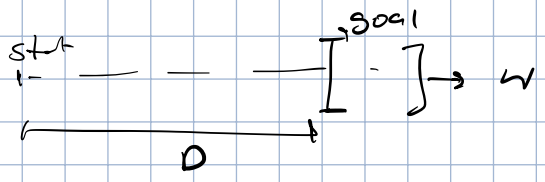
min. size of target

min (H, W, h. . .)

Dynamic C/D gain ⇒ motor space ≠ virtual space → artificially decrease D

Measure MT:

① Crossing selection:



② Steering selection: