

# Output: Systems, Toolkits, Platforms

4

**Georgia  
Tech**

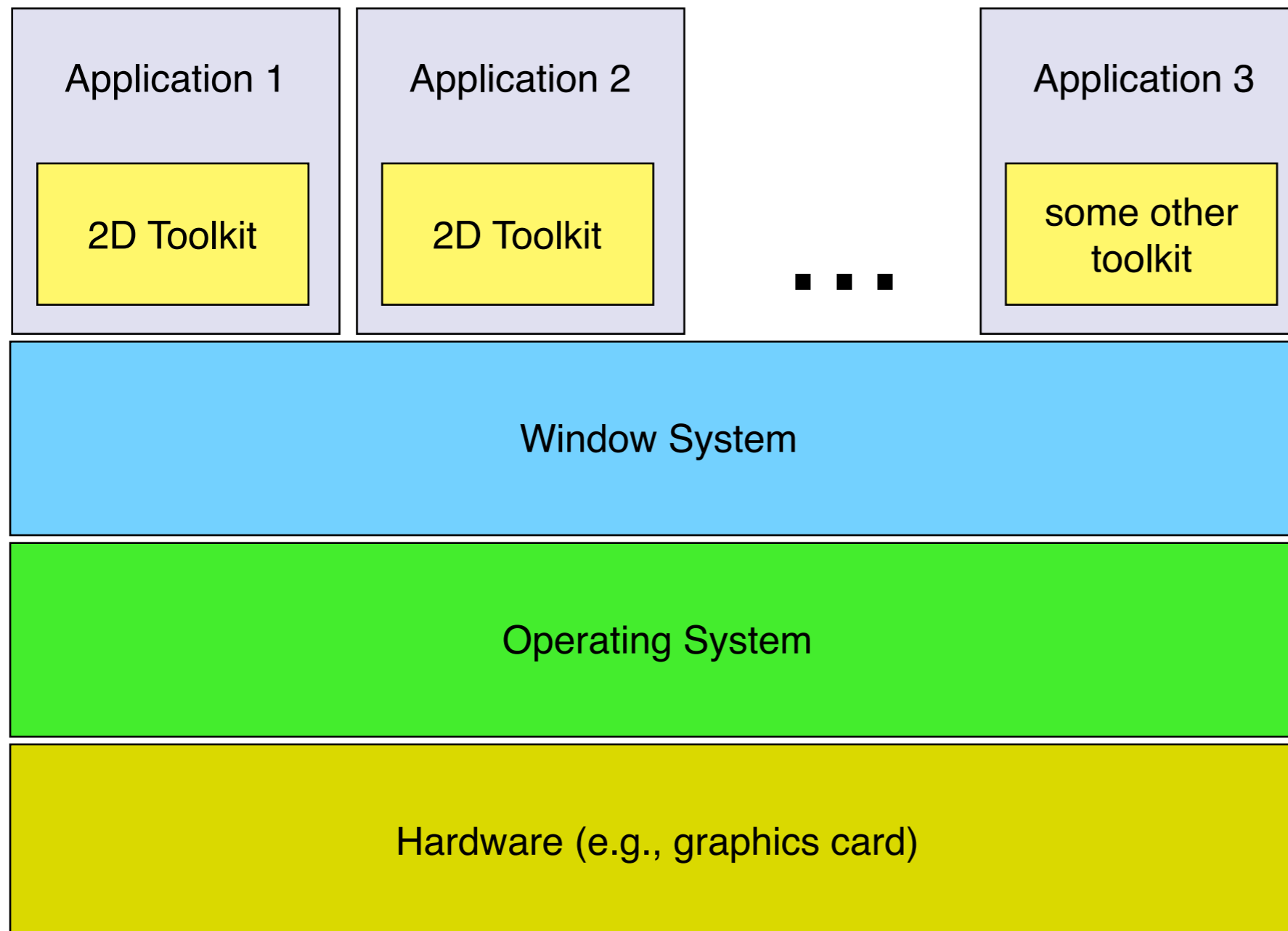


(structure and 2D content based on CS4470/6456 slides by Keith Edwards)

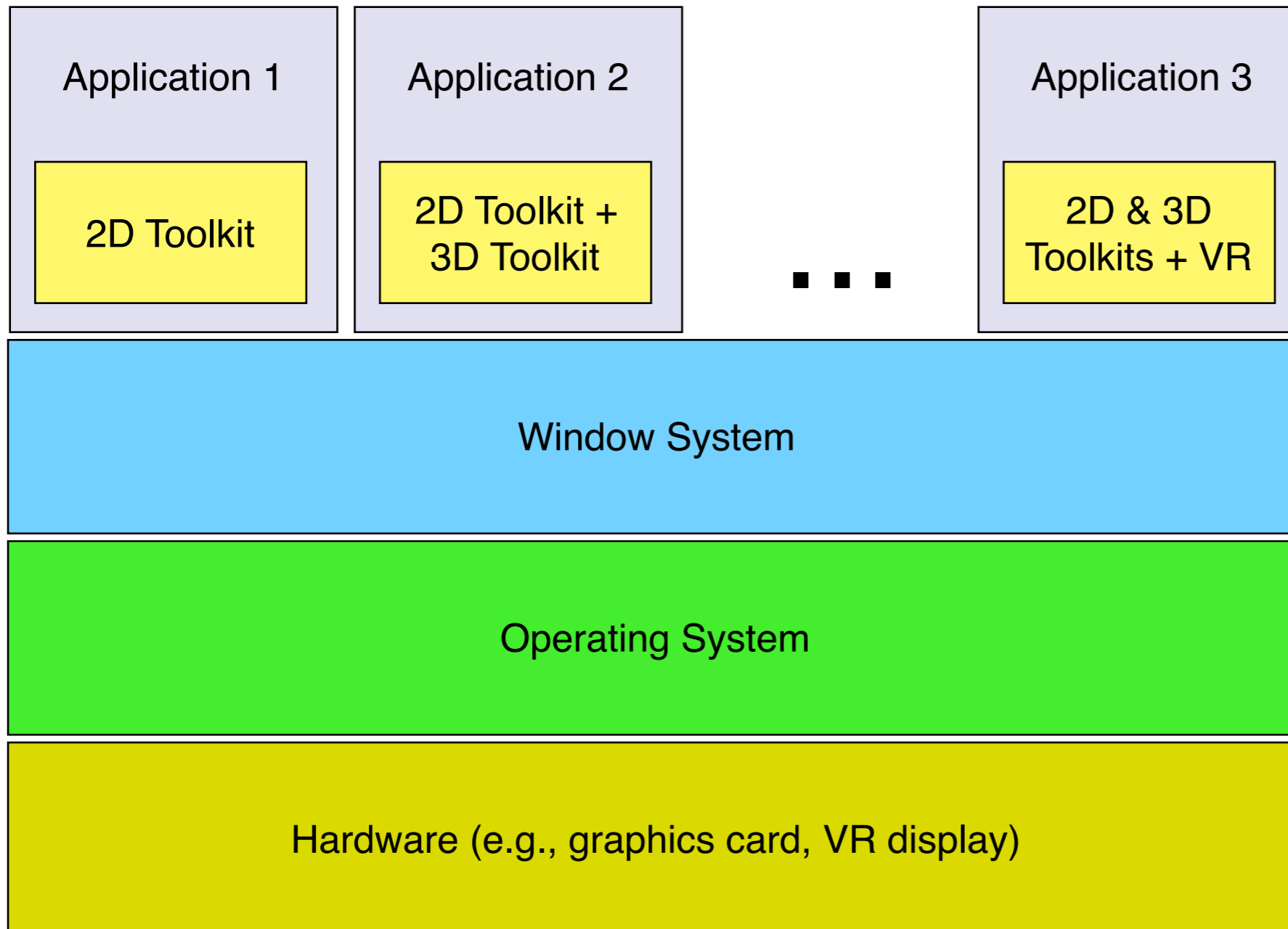
# Output Hardware (3DUI Ch 5)

- Visual Displays (**Ch 5.2**), Audio (last week, Ch 5.3), Haptics (Ch 5.4)
- Visual display characteristics
  - Field of View / Regard
  - Spatial Resolution
  - Screen Geometry
  - Light Transfer Mechanism
  - Refresh Rate
  - Ergonomics
  - Effect of Depth Cues
- Visual display fidelity components
  - Field of View / Regard
  - Spatial Resolution
  - Display Latency
  - Stereoscopy Quality
  - Color reproduction quality

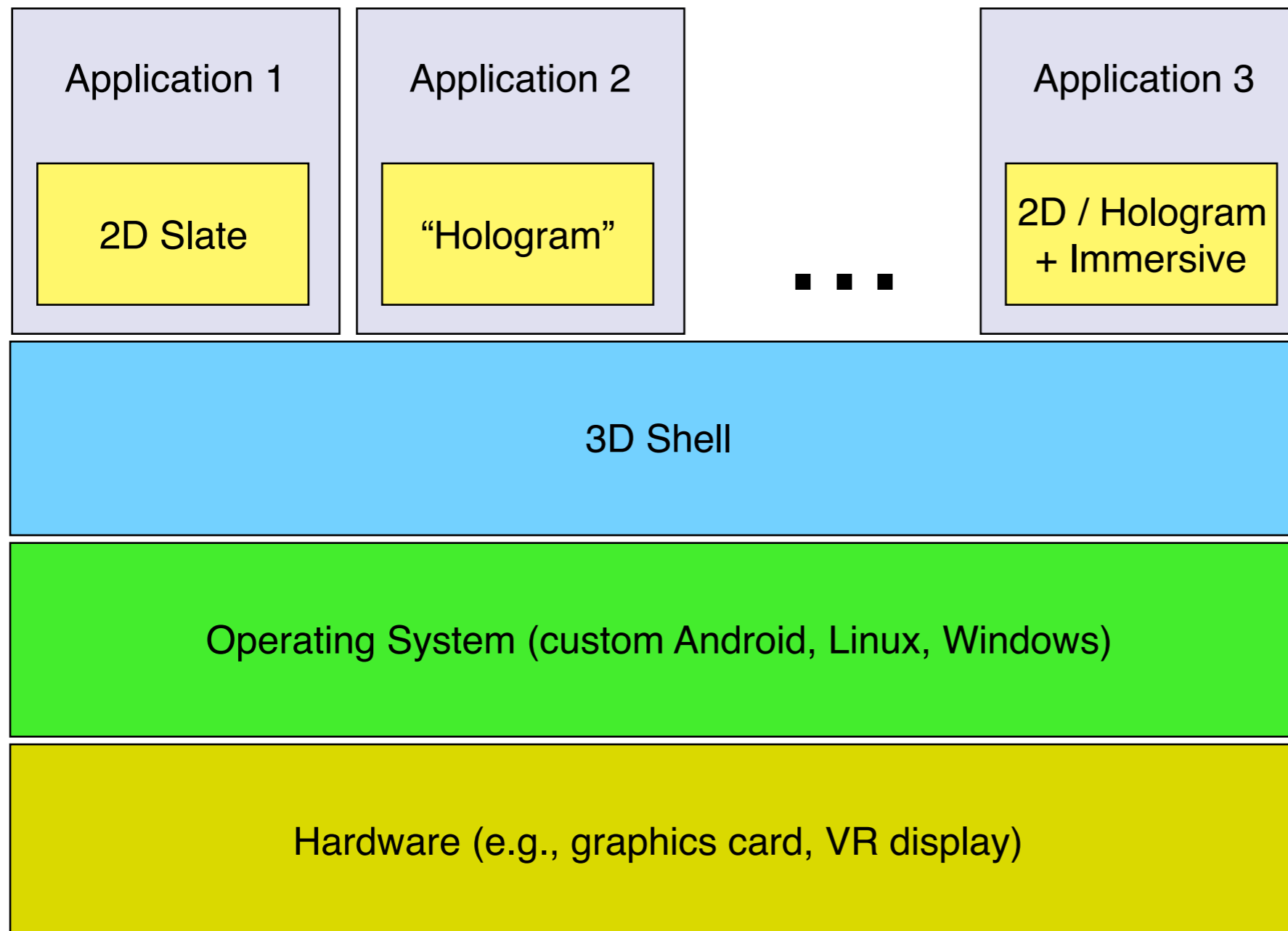
# Output System Layers: Desktops



# Output System Layers: Desktop VR



# Output System Layers: Standalone VR



# The Window System (2D)

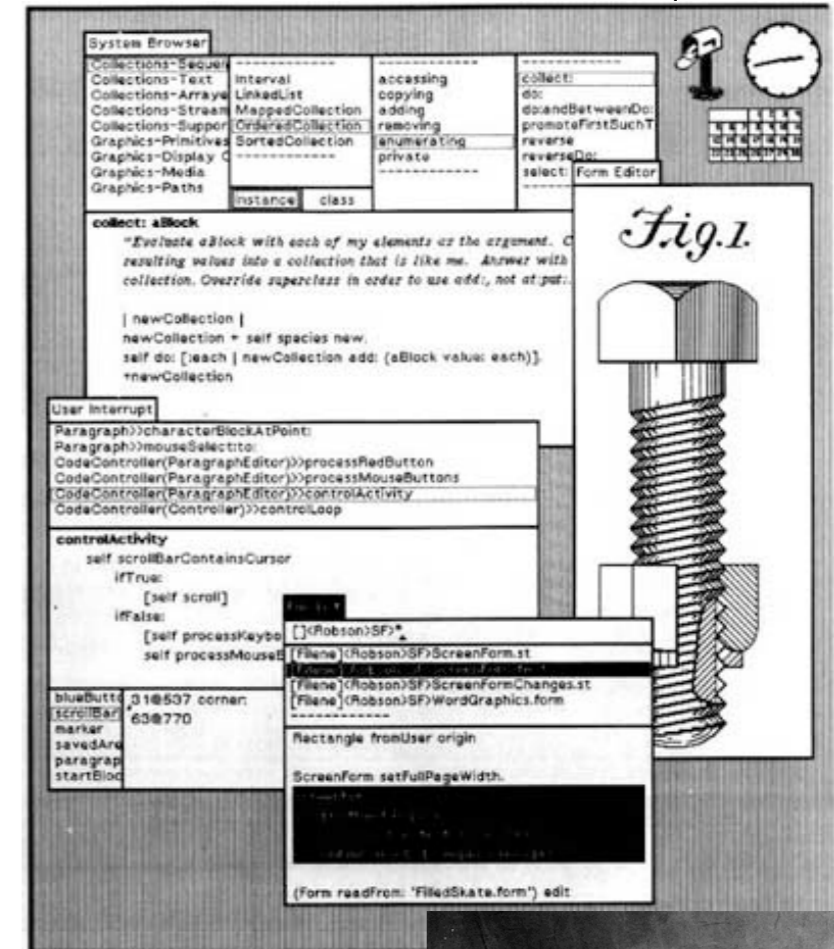
# Window System Basics

- Developed to support metaphor of overlapping pieces of paper on a desk (desktop metaphor)
  - Good use of limited space
    - leverages human memory
  - Good/rich conceptual model



# A little history...

- The BitBlit algorithm
  - Dan Ingalls, “Bit Block Transfer”
  - (Factoid: Same guy also invented pop-up menus)
- Introduced in Smalltalk 80
- Enabled real-time interaction with windows in the UI
- Why important?
  - Allowed fast transfer of blocks of bits between main memory and display memory
  - Fast transfer required for multiple overlapping windows
  - Xerox Alto had a BitBlit machine instruction





# Goals of window systems

- Virtual devices (central goal)
  - virtual display abstraction
    - multiple raster surfaces to draw on
    - implemented on a single raster surface
    - illusion of contiguous non-overlapping surfaces
    - Keep applications' output separated
  - Enforcement of strong separation among applications
    - A single app that crashes brings down its component hierarchy...
    - ... but can't affect other windows or the window system as a whole
- In essence: window system is the part of the OS that manages the display and input device hardware

# Virtual devices

- Also multiplexing of physical input devices
  - May provide simulated or higher level “devices”
  - Manages visual manifestation of cursor
  - Apps can’t generally take over cursor without system/user perms
- Overall better use of very limited resources (e.g. screen space)
  - Strong analogy to operating systems
  - Each application “owns” its own windows, and can’t clobber the windows of other apps
  - Centralized support within the OS (usually)
    - X Windows: client/server running in user space
    - SunTools: window system runs in kernel
    - Windows/Mac: combination of both

# Window system goals: Uniformity

- Uniformity of UI
  - The window system provides some of the “between application” UI
    - E.g., desktop
    - Cut/copy/paste, drag-and-drop
    - Window titlebars, close gadgets, etc.
  - consistent “face” to the user
  - allows / enforces some uniformity across applications
    - but this is mostly done by toolkit

# Uniformity

- Uniformity of API
  - Provides an API that the toolkit uses to actually get bits on the screen
    - provides virtual device abstraction
    - performs low level (e.g., drawing) operations
      - independent of actual devices
    - typically provides ways to integrate applications
      - minimum: cut and paste
      - also: drag and drop, notifications
  - The lower-level window system primitives might actually be used by *multiple* toolkits running in different applications

# GUI Toolkits versus Window Systems

- Early applications were built using *just* the Window System
  - Each on-screen button, scroll bar, etc., was its own “window”
  - Nested hierarchy of windows
  - Events dispatched to individual windows by the Window System, not by the GUI toolkit running inside the application
- Gradually, separation of concerns happened
  - Window system focuses on *mechanisms* and *cross-application separation/coordination*
  - Toolkits focus on *policy* (what a particular interactor looks like) and *within-application development ease*
- Now: GUI Toolkits need to interact with whatever Window System they’re running on (to create top-level windows, implement copy-and-paste), but much more of the work happens in the Toolkit
- **The window system manages pixels and input events on behalf of the application, but knows nothing about the application.** (Analogous to OS support for processes.)

# Window Systems Examples: I

- The X Window System
  - Used by Linux and many other Unix-like OS's today
  - *X Server* - long-lived process that “owns” the display
  - *X Clients* - applications that connect to the X Server (usually via a network connection) and send messages that render output, receive messages representing events
  - Early apps used no toolkits, then an explosion of (mostly incompatible, different looking) toolkits: KDE, GTK, Xt, Motif, OpenView, ...
- Good:
  - Strong, enforced separation between clients and server: network protocol
  - Allows clients running remotely to display locally (think supercomputers)
- Bad:
  - Low-level imaging model: rasters, lines, etc.
  - Many common operations require *round trips* over the network. Example: rubber banding of lines. Each trip requires network, context switch.

# Window Systems Examples: 2

- NeWS, the Network Extensible Window System (originally *SunDew*)
  - Contemporary of X Window System
  - Also network-based
  - Major innovation: stencil-and-paint imaging model
  - Display Postscript-based - executable programs in Postscript executed directly by window system server
- Pros:
  - Rich, powerful imaging model
  - Avoided the round-trip problem that X had: send program snippets to window server where they run locally, report back when done
- Cons:
  - Before it's time? Performance could lag compared to X and other systems...
  - Until toolkits came along (TNT - *The NeWS Toolkit*), required programming in Postscript

# Window Systems Examples: 3

- SunView
  - Created by Sun to address performance problems with NeWS
  - Much more “light weight” model - back to rasters
  - Deeply integrated with the OS - each window was a “device” ( in /dev )
  - Writing to a window happens through system calls. Need to change into kernel-mode, but no context switch or network transmission
  - Similar to how Windows worked up until Vista
- Pros:
  - lightning-fast
  - Some really cool Unixy hacks enabled: `cat /dev/mywindow13 > image.gif` to do a screen capture
- Cons:
  - No ability for connectivity from remote clients
  - Raster-only imaging model



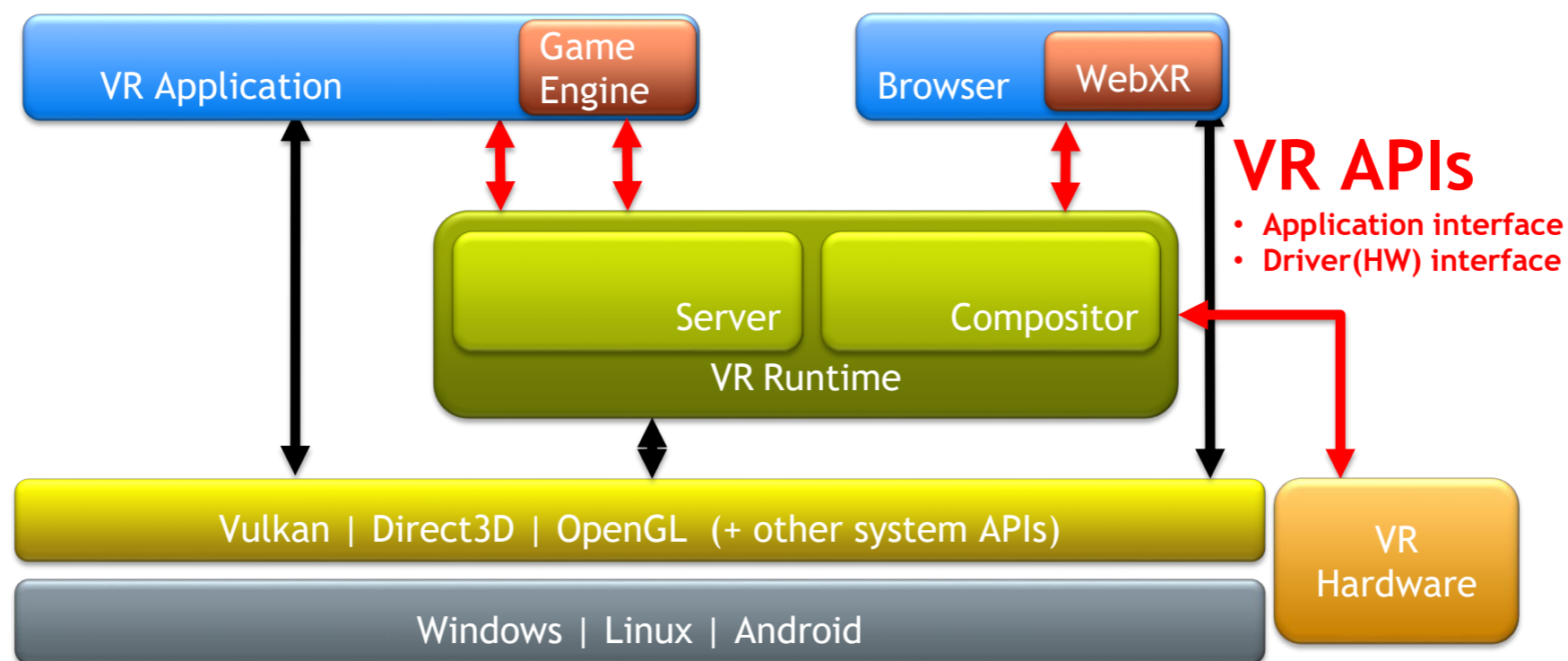
# Resolution Independence and HiDPI

- Recently, window systems taking on role of providing *resolution independence* and support for *HiDPI* displays (high dots-per-inch)
  - E.g., Apple Retina Display, 220 pixels-per-inch, 2880x1800 resolution
  - (Many “normal” displays ~100-120 pixels-per-inch, or roughly 1/4 retina display)
  - Analogous to CSS pixels (~96 dpi) and devicePixelRatio
- Resolution independence: UI elements are rendered at sizes *independent* from the underlying pixel grid. UI elements displayed at a consistent size, regardless of screen resolution.
  - Text rendered smoothly using all pixels, regardless of resolution
- Challenges:
  - Need extremely high bandwidth to display hardware; lots of pixels to update
  - Need stencil-and-paint-based underlying imaging model (vectors so text, strokes adapt to higher resolution)
  - Need high-resolution instances of any artwork used in the UI (icons, images of UI elements)

# 3D Systems

# What's Different? Desktop 3D

- No integration with Window System for 3D in 2D windows
  - Just a region in an app, implement all 3DUI yourself in the app
- VR devices are add-ons, custom drivers per device. Displays, controllers, etc.
  - Windows has OS-level support (WindowsMR)



# What's Different? Standalone VR

- System Shell is more “launcher” than “window system”
  - Shell experience focused on finding content/apps, managing device, launching
  - Immersive apps take over display, devices
  - Typically dedicated button to suspend/exit/summon dashboard
    - No multi-tasking (yet)
    - No system support for input device abstractions (yet)
  
- Future of Shells
  - Currently no “window system information management” capabilities
  - Immersive is all or nothing
    - What would it even mean to have multiple immersive apps at once
      - Argon I allowed multiple overlapping AR apps
    - Probably only makes sense in shell

# Core System Loop for VR/AR

- Once: Setup system to request features
- Each frame:
  - App gets state, renders frame, submits to server. Compositor renders to device
- State synchronized to a single time
  - Display intrinsics, controller poses, controller buttons and other peripheral state
- Render to 3D pre-distorted canvas using inputs
- Submit to VR runtime, compositor takes care of final rendering

# VR Runtime Compositor

- Two jobs: distort image to display geometry, handle latency
- Second job used to be ignored, or dealt with via prediction
  - Post rendering warping is key to modern VR/AR device usability
  - Timewarp, Spacewarp, Reprojection, Motion Smoothing, Asynchronous SpaceWarp
    - <https://uploadvr.com/reprojection-explained/>
    - <https://uploadvr.com/asw-2-rift-released/>
- Ideas have been around for decades, but Carmack / Oculus finally made it work in practice in 2013/2014
  - (<https://web.archive.org/web/20140719085135/http://altdev.co/2013/02/22/latency-mitigation-strategies/>)

# Timewarp

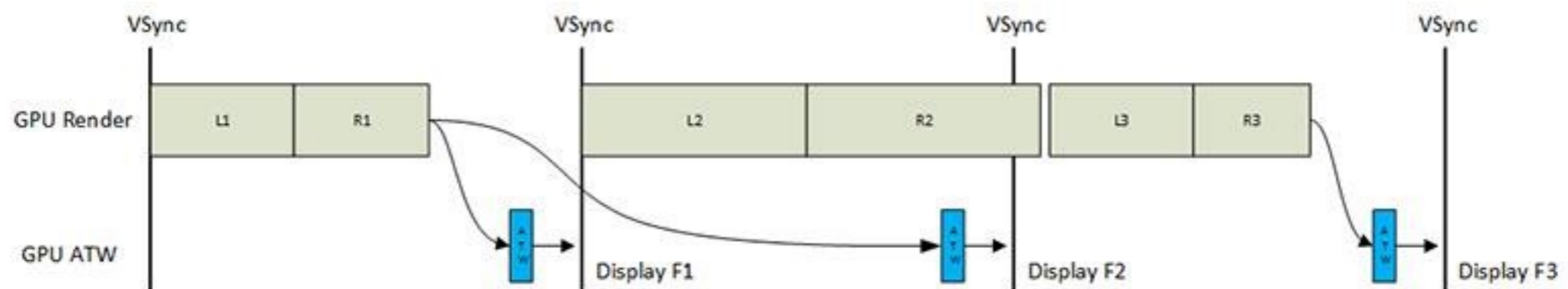
- Reproject rendered frame to account for change in head rotation from when tracking information given to application



- Only uses orientation

# Asynchronous Timewarp (ATM)

- Use Timewarp to deal with lost frames
- Keep frame around, asynchronously Timewarp old frame if a frame dropped



- Called Asynchronous Reprojection in the Steam SDK



# Asynchronous Spacewarp

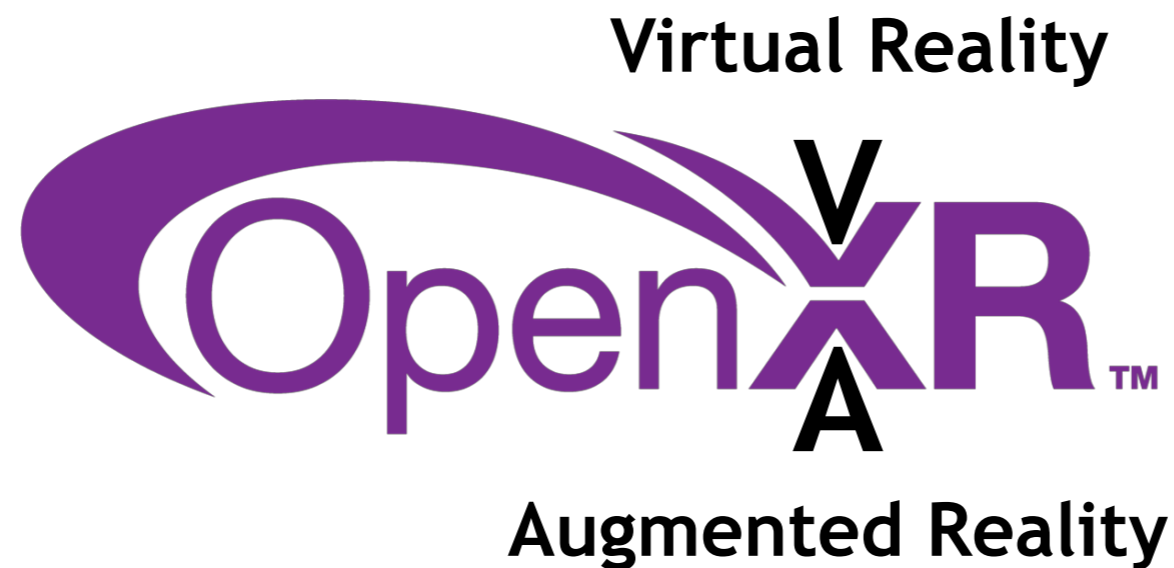
- Tries to account for motion as well as rotation by extrapolating image changes based on motion in previous frames



- ASW 2.0 incorporates depth information for better results

# Device Abstractions are Coming

- OpenXR from Khronos
  - Royalty-free, open standard that provides high-performance access to Augmented Reality (AR) and Virtual Reality (VR) platforms and devices.
  - VI is out, but much left to do



© Khronos® Group Inc. 2019 - Page 10

<https://www.khronos.org/assets/uploads/developers/library/2019-siggraph/OpenXR-BOF-SIGGRAPH-Jul19.pdf>













## A Brief History of OpenXR

- Among the first VR hardware available 2016

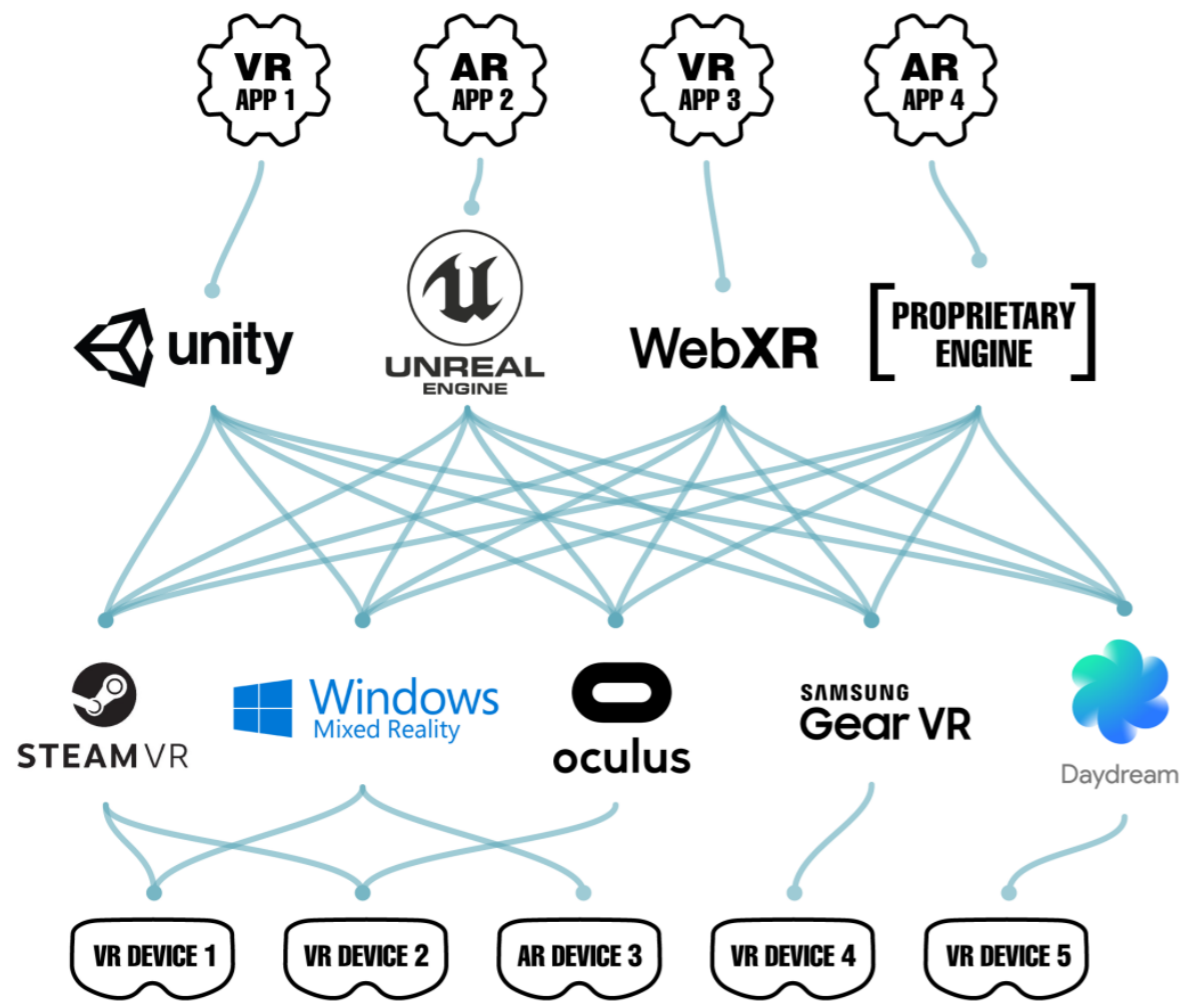


- Need applications...
  - Each platform provided an SDK to interface with the hardware
  - Each was different from the other

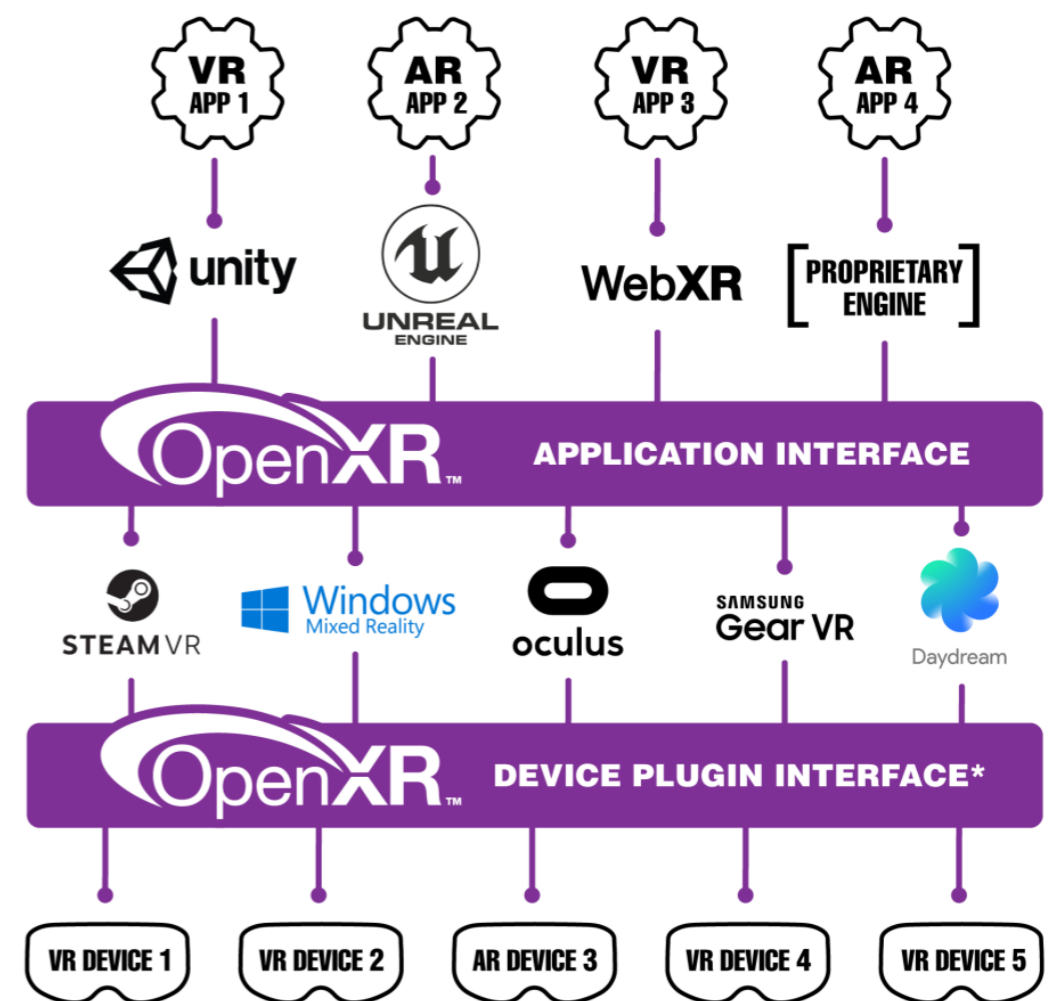
# Major XR Runtimes

	Virtual Reality						Augmented Reality				Console VR
	PC			AIO	Mobile		AIO		Mobile		
	Oculus Rift	SteamVR	Mixed Reality	Oculus Go	Daydream	GearVR	Hololens	ML1	ARKit	ARCore	PSVR
Company	Facebook	Valve	Microsoft	Facebook	Google	Samsung Oculus	Microsoft	Magic Leap	Apple	Google	Sony
OS support		 									

# OpenXR - Solving XR Fragmentation

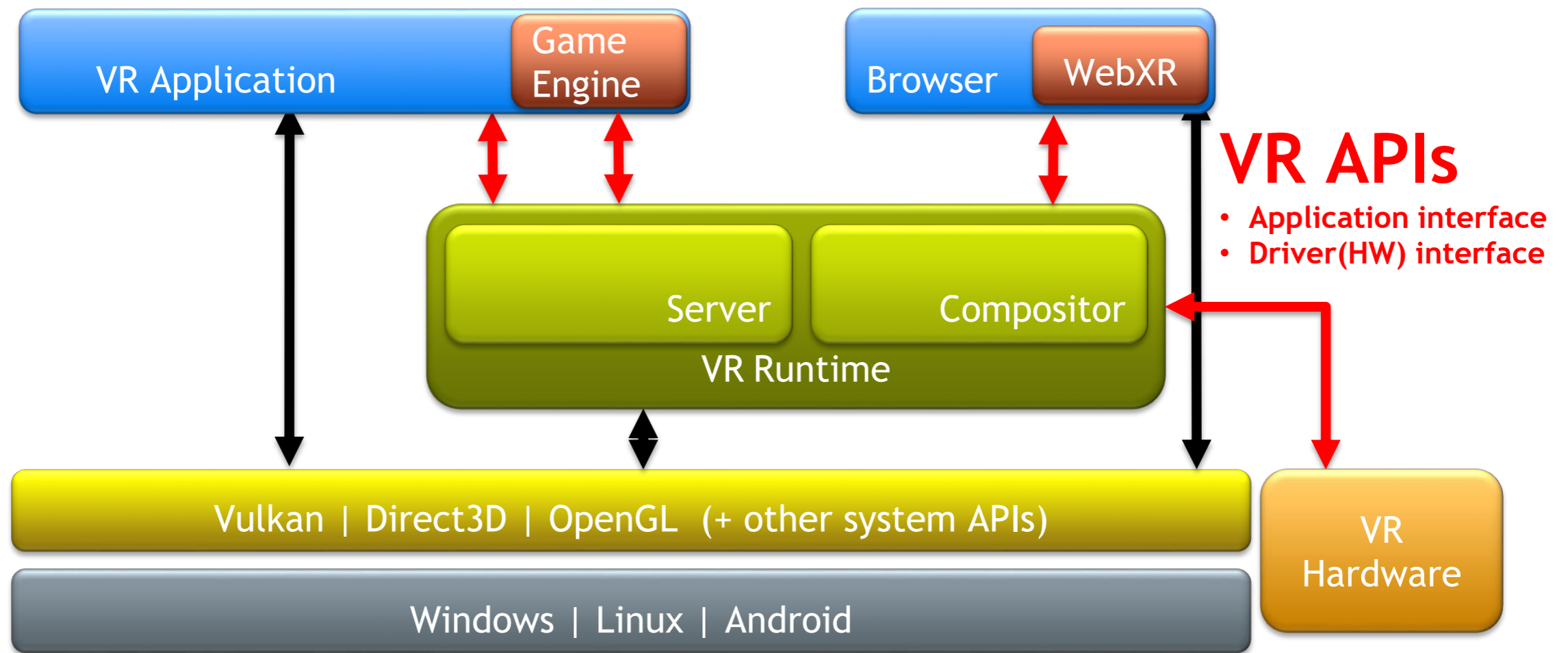


**Before OpenXR**  
XR Market Fragmentation



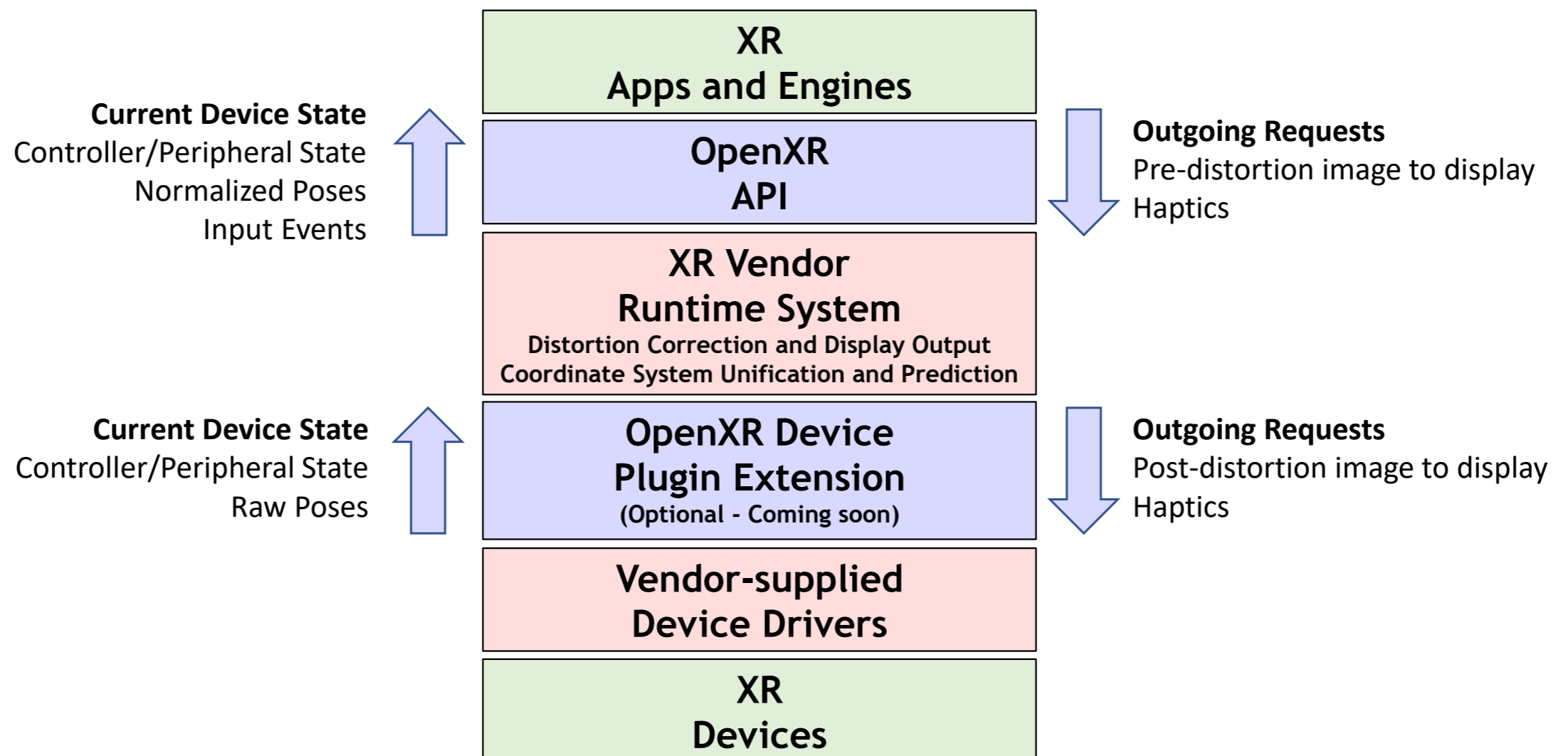
**After OpenXR**  
Wide interoperability of XR apps and devices

# VR Software Stack (Example)

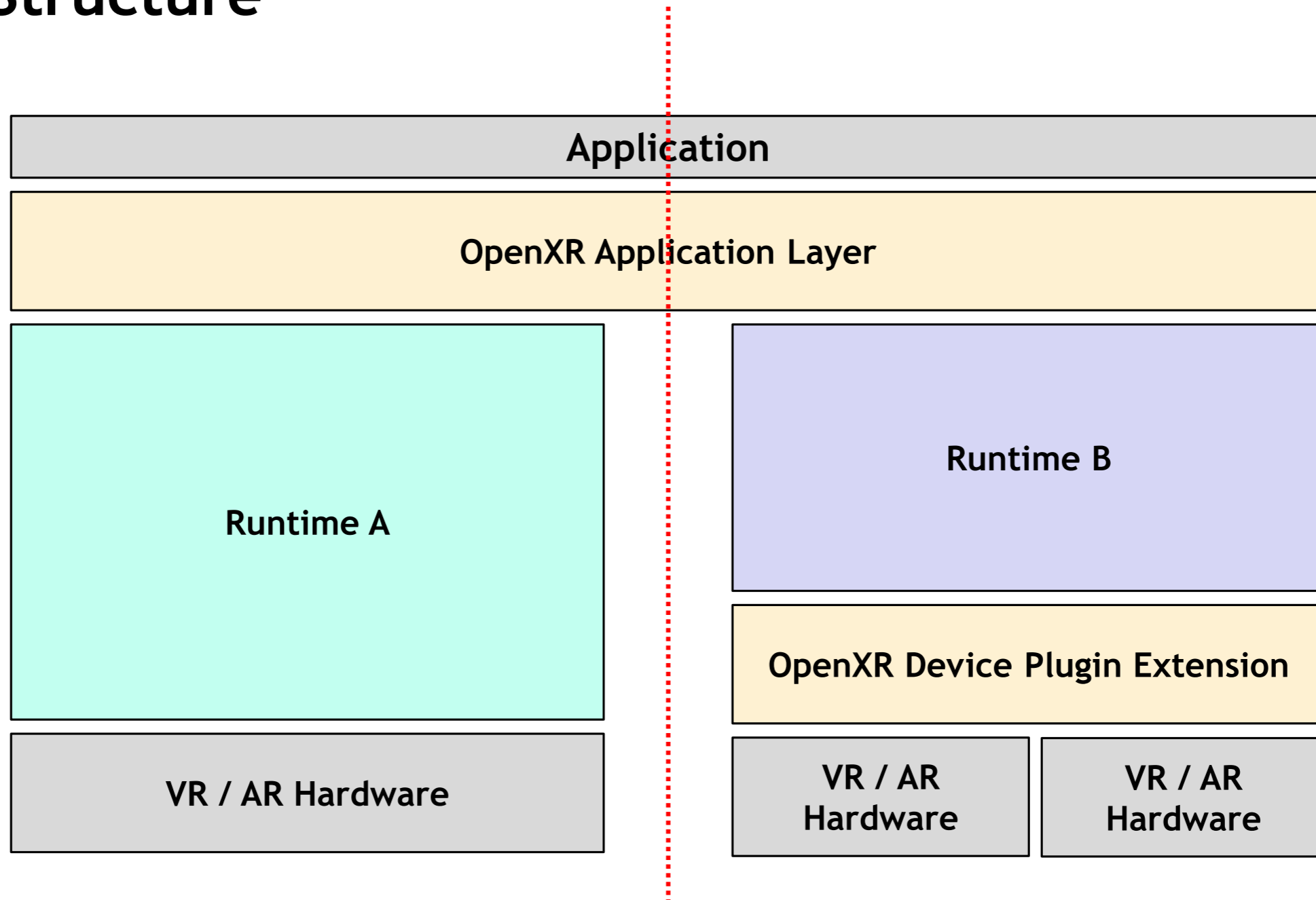


# OpenXR Architecture

OpenXR does not replace XR Runtime Systems!  
It enables any XR Runtime to expose CROSS-VENDOR APIs to access their functionality



# The Structure





# OpenXR Philosophies

1

## Enable both VR and AR applications

The OpenXR standard will unify common VR and AR functionality to streamline software and hardware development for a wide variety of products and platforms

## Be future-proof

2

While OpenXR 1.0 is focused on enabling the current state-of-the-art, the standard is built around a flexible architecture and extensibility to support rapid innovation in the software and hardware spaces for years to come

## Do not try to predict the future of XR technology

3

While trying to predict the future details of XR would be foolhardy, OpenXR uses forward-looking API design techniques to enable engineers to easily harness new and emerging technologies

## Unify performance-critical concepts in XR application development

4

Developers can optimize to a single, predictable, universal target rather than add application complexity to handle a variety of target platforms

# Abstractions Still Very Low Level

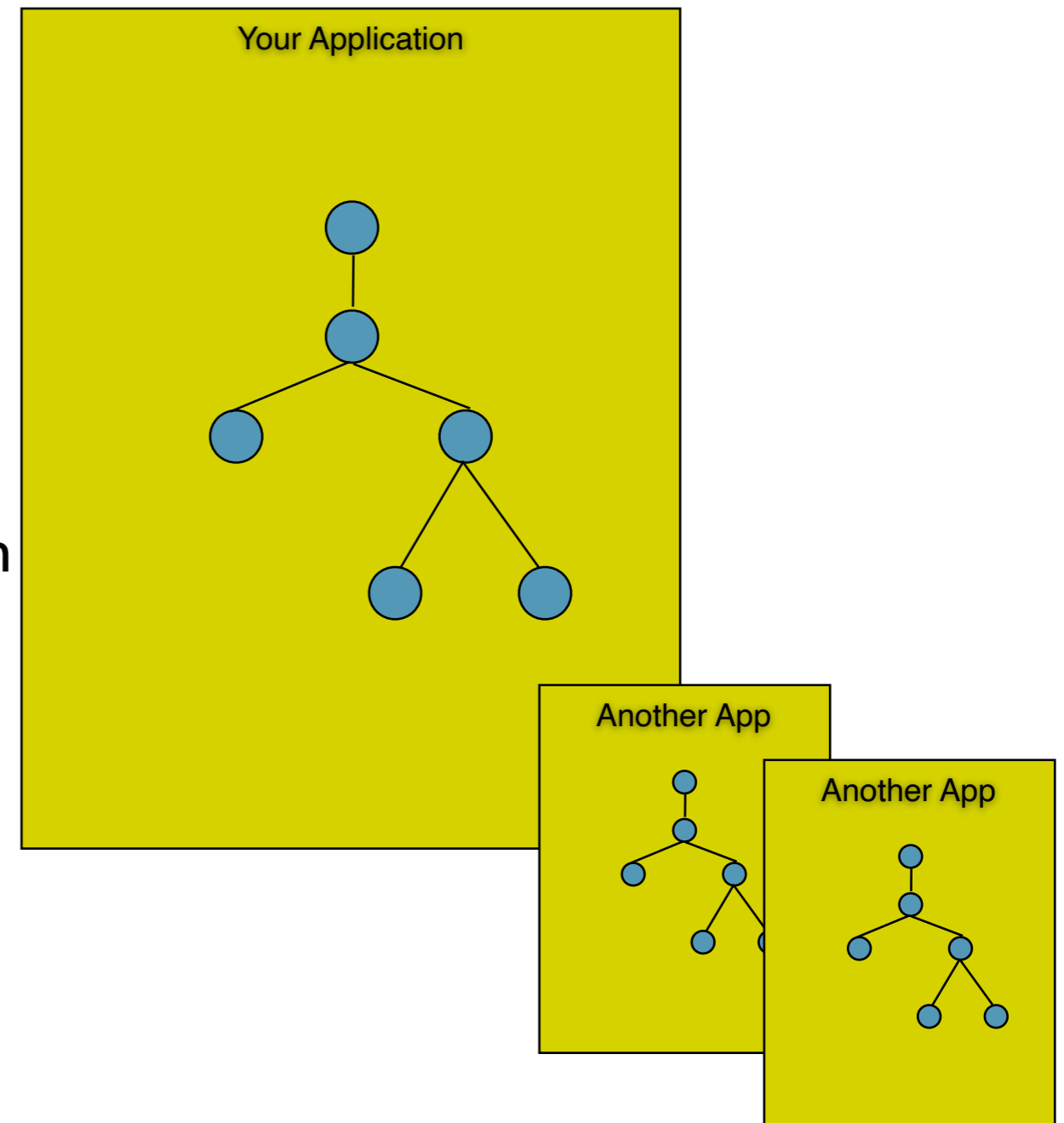
- No user-level abstractions, apps implement their own UI
- System-level interactions require “popping out” to the Shell/system
  - On desktop devices, go back to main display to deal with prompts
  - On all-in-one devices, pops out to “3D home” / “Shell” to deal with prompts
- Apps need to know “too much” about the platform
  - e.g., controller models
  - OpenXR is trying to fix this part
  - WebXR is hoping to follow suit



# The Toolkit Layer

# Finally, we get to the application

- All of the 2D and 3D components you use are a part of your application
  - i.e., in your application's process
- Toolkit code gets linked into your app.
- Multiple apps each have their own hierarchy of components, in their own address spaces
- Lots of Toolkits:
  - UIKit (MacOS X/iOS), SWT (Java), Windows Presentation Foundation (Win), Qt (Linux)
  - Babylon, three.js, Unity, Unreal, Godot, ...
    - 3D “components” less uniform



# Object-oriented abstractions for drawing

- Most modern toolkits provide an object that provides uniform access to all graphical output capabilities / devices
  - Treated as abstract drawing surface
    - “Canvas” abstraction
      - Macintosh: grafPort
      - Windows: device context
      - X Windows-based Toolkits: GC (GraphicsContext)
      - Java: Graphics/Graphics2D classes
      - HTML/CSS: Canvas, GraphicsContext
- 3D toolkits typically rely on a 3D version of these surfaces
  - VR runtimes typically use a specific variation of these
  - But the general approach uses the same 3D surface for 3D-in-a-window and 3D-in-a-device

# Object-oriented abstractions for drawing

- 2D toolkit abstraction provides set of drawing primitives
  - Might be drawing on...
    - Window, direct to screen, in-memory bitmap, printer, ...
  - Key point is that you can write code that doesn't have to know which one
- 3D abstraction provides a set of 3D primitives that are geared toward efficient use of the GPU
  - Direct3D, OpenGL, Vulkan, WebGL, etc
  - Stream-of-commands-to-GPU model
  - Content (e.g., textures) might be managed for you

# Object-oriented abstractions for drawing

- Generally don't want to depend on details of device but sometimes need some:
  - How big is it
  - Is it resizable
  - Color depth (e.g., B/W vs. full color)
  - Pixel resolution (for fine details only)
  - Depth, Stencil, Texture buffers

# How does the Toolkit interact with the Window System or AR/VR Shell?

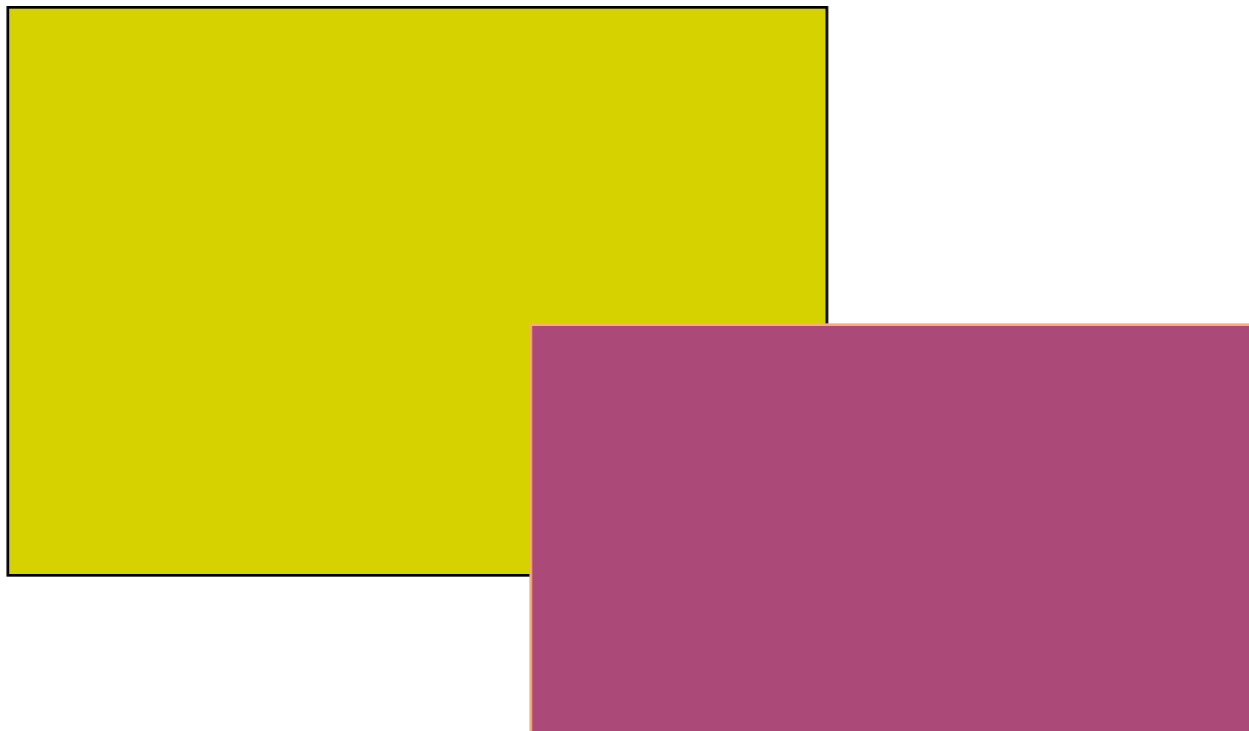


# Window Systems v. GUI Toolkits

- GUI Toolkit: what goes on *inside* a window
  - Components, object models for constructing applications
  - Dispatching events among all of the various listeners in an application
  - Drawing controls, etc.
- Window System: from the top-level window *out*
  - Creates/manages the “desktop” background
  - Creates top-level windows, which are “owned” by applications
  - Manages communication between windows (drag-and-drop, copy-and-paste)
  - Interface w/ the Operating System, hardware devices, renders cursor
- AR/VR Shell currently provides few abstractions, but this will change
- GUI toolkits are frameworks used inside applications to create their GUIs.
- Window systems are used as a system service by multiple applications (at the same time) to carve out regions of screen real estate, and handle communication. **In essence, the window system handles all the stuff that can’t be handled by a single application.**

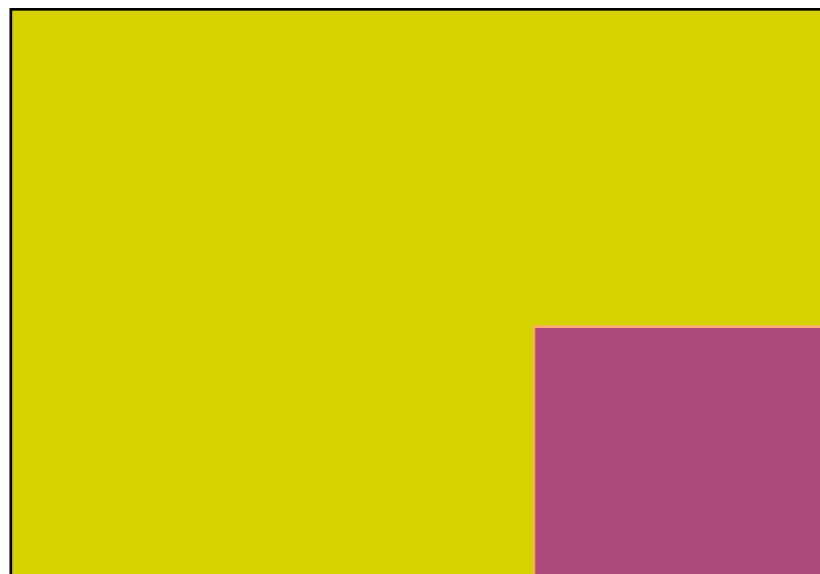
# Example: 2D damage / redraw mechanism

- In 2D, only redraw when necessary
- Windows suffer “damage” when they are obscured then exposed (and when resized)



# Damage / redraw mechanism

- Windows suffer “damage” when they are obscured then exposed (and when resized)
- At some level, the window system *must* be involved in this, since only it “knows” about multiple windows



Wrong contents,  
needs redraw

# Damage / redraw, how much is exposed?

- One option: Window System itself does the redraw
  - Example: Window System may retain (and restore) obscured portions of windows
  - “Retained Contents” model
- Another option: Window System just detects the damage region, and notifies the application that owns the uncovered window (via a Window System-level “Exposure” event)
  - Toolkit gets the message from the Window System and begins its own, internal redraw process of your application
  - Applications draw into the shared framebuffer, with the Window System ensuring they don’t trample on each other
  - This is what typically happens these days...

# Damage / redraw, how much is exposed?

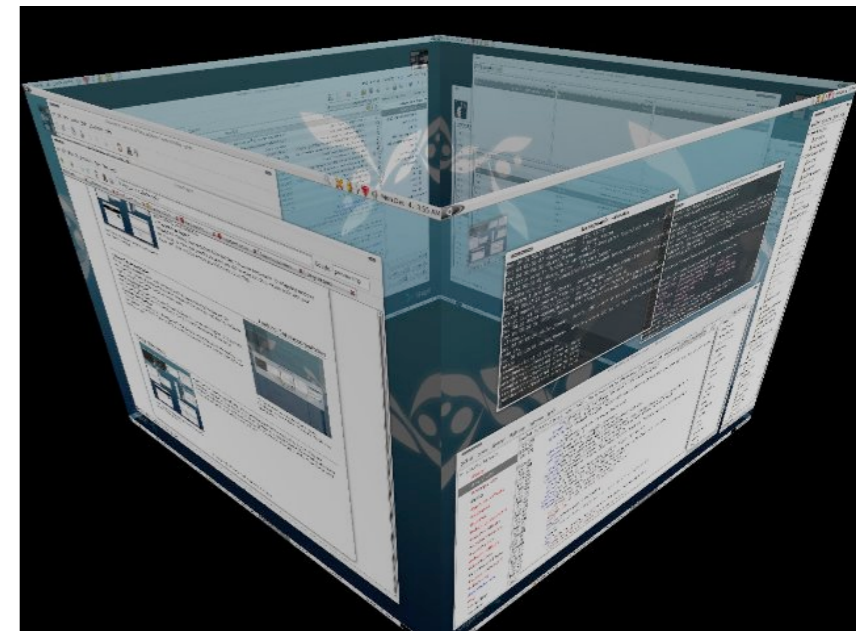
- In many toolkits, you can still optionally use the “retained contents” model
  - Can use it when you know your application contents are not going to change--just let the Window System manage it for you
  - Very efficient
- In general:
  - Redraw can happen because the Window System requests it, or the application decides that it needs to do it
  - After that point, redrawing happens internally to the application with the toolkit’s help

# But there's a twist...

- What if you could avoid involving applications in the need to redraw just because their windows get exposed? Is there a way to do this?
- Some modern window systems actually have a way around this
- Leverage the powerful 3D cards in today's computers
- Basic idea:
  - Let applications draw into their own buffer area, with no interaction from other applications
  - Use the video card hardware to quickly copy and stitch these together at interactive speeds; window system is the “visual mixing board”
  - The trick: the “buffer area” for applications is the video card's texture memory, and the “desktop” is actually a 3D scene created by the Window System
- Benefits:
  - Applications don't get asked to redraw themselves due to exposure events from the Window System: they just draw into their “virtual” frame buffers without care for whether they're covered or not
  - Once these virtual framebuffers are on the video card, the card can do fancy effects with them.

# How it works

- The Window System is now a 3D application that uses the video card
- Each application draws its window contents to a buffer that's then copied into the video card's texture memory
- The Window System then *composits* these individual areas together into a 3D scene (to control Z-ordering of windows)
  - Hence the term *compositing window manager* versus *stacking window manager*
  - This takes care of occlusion, overlapping windows
  - Apps just draw into their buffers as if they're always fully exposed, but only in response to application state changes



# Window Systems Examples: 4

- The Windows Vista (and later) Window System: Desktop Window Manager (DWM): January, 2007
  - Traditionally, apps were asked by Windows to paint their visible regions, and then they painted directly to video card buffer.
  - With the Windows Vista/7 Desktop Window Manager (DWM), all window drawing is redirected to separate memory bitmaps and composited in the video card, and only then finally sent to the display.
  - To leverage the capabilities of the video card and modern graphics technology generally, all of this compositing goodness is done through Windows' low level 3D graphics API, Direct3D.
- (MacOS X window system, called *Quartz Compositor*, is basically similar to this.)



# Basic Pathway

- First, each app gets two memory bitmaps: the first is in system memory and the second is in graphics memory.
- Drawing operations by the app are rendered on the system memory buffer
- Eventually, when the app has finished redrawing its window, the DWM will copy that window's system memory buffer into the graphics card's memory.
  - *Double-buffering* ensures nothing ever gets to the graphics card half-drawn.
- Now, using Direct3D (Windows' 3D API), the DWM takes each window's image in the graphics card, and uses it as a 3D texture object to texture a rectangle in a 3D scene; rectangles are positioned according to how windows on the screen are arranged

# Advantages of this System

- Can easily do cool window distortions (like Flip3D, “genie effect” on MacOS X).
- Can access continually-updated window images (“live previews” in the taskbar, etc).
- Dragging a window doesn’t force all the windows behind it to re-render, thus preventing “trails” as you drag a window.
- Easy to do window scaling to compensate for naive apps on high DPI displays.
- Different performance characteristics: doesn’t involve apps in redraw process just because their windows are exposed; only when their actual contents change

# A Possible Concern

- Doesn't all of this need a lot of memory? Yes.
- But:
  - In recent generations of DirectX (which underlies Direct3D), the drivers actually virtualize graphics memory and allow interruptibility of the GPU. Result: graphics memory allows for paging. So when the video memory runs out, the system sends unneeded pages out to normal system memory.
- Does it need some beefy hardware? Yes.
  - E.g., full Vista compositing effects required 1GB RAM, video card with 128MB texture memory, pixel shaders, etc etc etc.
- But:
  - Moore's Law makes most problems go away

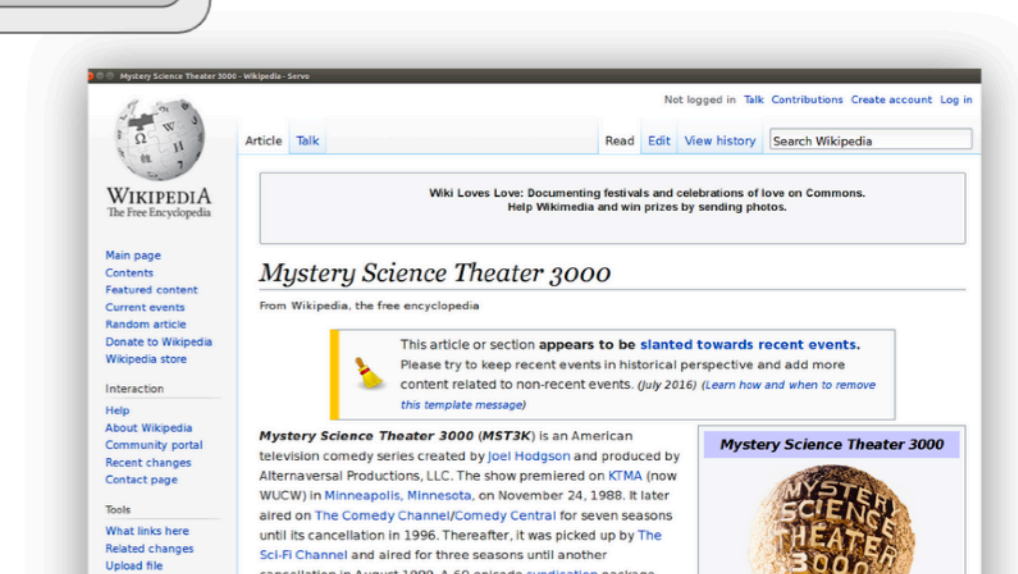
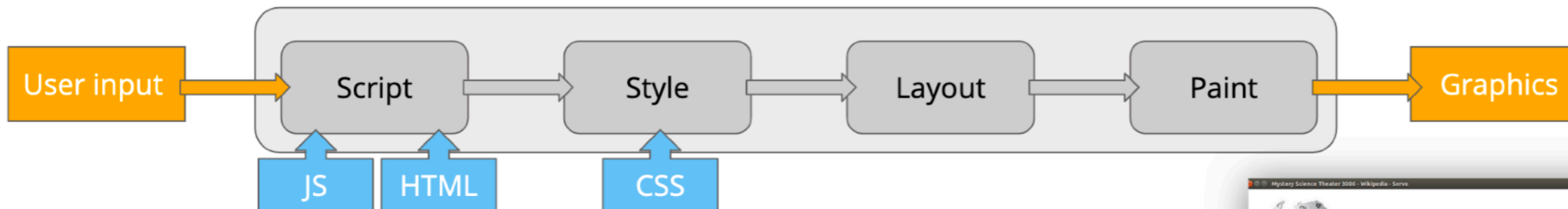
# Balance of Responsibility

- Over the past few years, the balance of what happens in the toolkit versus what happens in the Window System has been changing
- Lots of complex tree walks, querying of object state, etc., in many applications
  - Means that you don't want to have to do a process switch, or inter-process communication for each: so much of this functionality migrated into more complex toolkits in the '80's and '90's
  - These local (i.e., within the application's address space) operations are much faster than having to communicate millions of times with an external window system
- But Window Systems have gotten more complicated too
  - Introduction of compositing window managers is a “trick” that means applications may no longer have to redraw as much, also allows fancier graphics effects
- Having all 2D Windows in GPU allows Windows MR, HoloLens to easily get them into VR/AR
  - Enables screen sharing and VR apps like BigScreen, etc., as well

# Same Approach Useful in Apps: Consider Browsers

## Browser architecture

Highly simplified (i.e. contains lies)



# Servo uses GPU for most rendering, elements are textures

## Multi-threaded/process browser architecture

Modern architectures have multiple cores and a GPU

