ChatGPT

Desktop-Capturing mit Windows 11: Mehrere Bildschirme und virtuelle Desktops streamen

Erfassen mehrerer physischer Bildschirme (Dual-Monitor-Capture)

Um beide physischen Monitore gleichzeitig abzugreifen, kann FFmpeg's **GDI-Grab** oder der neuere **Desktop Duplication API**-Grabber genutzt werden. Mit dem GDI-Grabber (`gdigrab`) lässt sich der gesamte **erweiterte Desktop** (alle Monitore als ein großes Bild) erfassen. Ein einfacher Befehl lautet zum Beispiel:

```
ffmpeg -f gdigrab -framerate 20 -i desktop -vcodec libx264 -preset ultrafast
output.mp4
```

Dieser Befehl nutzt `-i desktop`, was *alle* Displays als eine einzige große Fläche aufnimmt [1]. In einem Setup mit zwei 1920×1080-Monitoren nebeneinander würde das z. B. ein 3840×1080-Bild streamen. Falls FFmpeg die gesamte virtuelle Bildschirmfläche nicht automatisch erkennt, können Sie mit `-video_size 3840x1080 -offset_x 0 -offset_y 0` explizit die Auflösung und Offsets angeben, um den gesamten Bereich abzudecken.

Tipp: Neuere FFmpeg-Versionen (ab ca. v5.x/6.x) unterstützen das **DXGI Desktop Duplication API**-basierte Capturing via `ddagrab`. Dieses ist hardwarebeschleunigt und performanter als GDI-Grabbing [2]. Die Verwendung erfolgt über das Filter-Device `-f lavfi -i ddagrab`. Beispielsweise:

```
# Primären Bildschirm mit Desktop Duplication API erfassen (mit NVIDIA
Encoder):
ffmpeg -f lavfi -i ddagrab -c:v h264_nvenc -framerate 30 -cq 18 output.mp4
```

Hier erfasst `ddagrab` standardmäßig den ersten Monitor (Index 0). Mit der Option `output_idx=<N>` kann ein spezifischer Monitor ausgewählt werden (z. B. `output_idx=1` für den zweiten Bildschirm) [3] [4]. So könnten Sie auch zwei FFmpeg-Prozesse parallel starten – jeweils einen pro Monitor – und zwei separate Streams erzeugen. Standardmäßig erfasst `ddagrab` die volle Auflösung des gewählten Monitors (bzw. bei `gdigrab` des gesamten Desktops). Die Maus wird übrigens mit aufgenommen (standardmäßig an, steuerbar über `-draw_mouse` bzw. bei ddagrab `draw_mouse=1/0` [5]).

Alternative Tools: Neben FFmpeg gibt es Tools wie **OBS Studio** (Open Broadcaster Software), das eine GUI bietet, um mehrere Bildschirme oder Fenster zu capturen. OBS kann z.B. beide Monitore als Quellen hinzufügen und einen virtuellen Bildschirm daraus generieren. Für unsere Zwecke (Weiterleitung über WebSocket) ist FFmpeg jedoch gut geeignet, da es direkt in einen Stream oder in Pipelines ausgeben kann. Ein weiterer Ansatz für Multi-Monitor-Capture ist die Verwendung eines **DirectShow**-Screen-Capture-Filters (wie `screen-capture-recorder`), der in FFmpeg via `-f dshow`

eingebunden werden könnte – allerdings liefert `gdigrab` / `ddagrab` bereits direkt die gewünschten Ergebnisse.

Aufnahme von Fenstern auf virtuellen Desktops (Windows 11 Taskansicht)

Virtuelle Desktops und Aufnahme-Limitierungen: Windows 10/11 bieten zwar mehrere virtuelle Desktops, jedoch wird **immer nur der aktive Desktop tatsächlich gerendert**. Fenster, die auf einem anderen (inaktiven) virtuellen Desktop liegen, werden vom Desktop-Compositor nicht gezeichnet und sind somit für Capture-Tools unsichtbar [6]. Ein Fenster, das nicht auf dem aktuellen virtuellen Desktop ist, erscheint im Capture entweder als *schwarzes Bild* oder es zeigt den zuletzt gerenderten Frame (statisch). In einer Test-Umgebung wurde z.B. beobachtet, dass OBS Studio ein Fenster auf Desktop 2 nicht mehr aktualisiert, sobald man zu Desktop 1 wechselt – es resultiert in einem eingefrorenen oder schwarzen Bild [7]. Diese Einschränkung ist systembedingt: „No amount of BitBlt can capture something that isn't being drawn" [6], d.h. es ist (ohne Tricks) **nicht möglich**, gleichzeitig einen nicht angezeigten virtuellen Desktop abzugreifen [8].

Mögliche Workarounds: Da direktes Capturing mehrerer virtueller Desktops gleichzeitig nicht unterstützt wird, muss man kreativ werden:

- **Fenster-Capture auf aktivem Desktop:** Falls Sie bestimmte Programme streamen möchten, platzieren Sie diese auf dem aktuell sichtbaren Desktop (oder ziehen Sie sie auf einen der physischen Monitore). Mit FFmpeg können Sie einzelne **Fenster** statt des ganzen Desktops erfassen, z. B. via Titel: `ffmpeg -f gdigrab -framerate 20 -i title="Fenstertitel" ...`. Dieses Verfahren funktioniert aber nur, solange das Fenster auf dem aktiven Desktop sichtbar ist. Sobald das Fenster in einen virtuellen Desktop wechselt, der gerade nicht aktiv ist, liefert auch die Fensteraufnahme kein Bild mehr.

- **Automatisches Umschalten:** Theoretisch könnte man zwischen virtuellen Desktops hin- und herschalten und jeweils einen Screenshot/Frame aufnehmen. Praktisch ist das für einen Live-Stream aber ungeeignet (hohe Latenz, Flackern am lokalen Bildschirm, Verlust der Gleichzeitigkeit).

- **Virtuelle Maschine oder Sandbox:** Eine bewährte Lösung für parallele Desktops ist der Einsatz von **VMs** oder Containern. Beispielsweise können Sie **Windows Sandbox** oder eine leichte **VM (Hyper-V, VirtualBox)** nutzen, um ein sekundäres Windows-System laufen zu lassen. Auf diesem zweiten System können die gewünschten Anwendungen ausgeführt werden. Die Ausgabe der VM (die ja in einem Fenster auf Ihrem Host angezeigt wird oder per RDP gestreamt werden kann) lässt sich wiederum wie ein normaler Bildschirm oder ein Fenster abgreifen. So haben Sie faktisch eine zweite unabhängige Desktop-Umgebung, die simultan sichtbar ist. Diese VM könnte man auf den zweiten Monitor schieben oder auch im Hintergrund laufen lassen und direkt deren Framebuffer capturen (einige VM-Software bieten APIs für Bildschirmzugriff). Der Overhead ist höher, aber es umgeht die Einschränkungen der Windows-Taskansicht.

- **Weitere Ansätze:** Tools wie **Sysinternals Desktops** (die echte separate Desktop-Instanzen innerhalb einer Session starten) helfen hier leider ebenfalls nicht weiter – auch dort wird ein Desktop nur gerendert, wenn er aktiv ist. Alternativ könnte man überlegen, pro zu überwachende Anwendung eine *Remote Desktop* Sitzung zu öffnen (auf localhost, via RDP/Terminal-Services), um so mehrere Sessions parallel laufen zu haben. Allerdings erfordert das

Windows-Server-Funktionen oder entsprechende Hacks, da Windows 11 normalerweise nur eine gleichzeitige Benutzer-Session zulässt.

Zusammengefasst sind die besten Lösungen: **Programme möglichst auf sichtbaren Desktops/ Monitoren halten** oder **separate Umgebungen verwenden**, wenn simultan gestreamt werden muss. Ein einfacher Trick im Zwei-Monitor-Betrieb kann sein, auf Monitor 1 normal zu arbeiten und Monitor 2 als „Anzeigewand" für die zu streamenden Fenster zu verwenden – beide Monitore werden ja im oben genannten FFmpeg-Befehl erfasst (3840×1080 Gesamtbild). Dadurch könnten Sie z. B. Desktop 1 auf beiden Monitoren verwenden, aber nur bestimmte Fenster auf Monitor 2 darstellen, die für den Stream gedacht sind, während Ihre privaten Fenster auf Monitor 1 bleiben.

Streaming des Captures im Frontend (WebSocket, MJPEG, WebRTC)

Wenn die Bildschirminhalte per WebSocket an ein Frontend (z. B. eine React-App) gesendet werden sollen, gibt es verschiedene Ansätze, den Videostream im Browser darzustellen:

- **MJPEG-Stream (Multipart JPEG)**: Ein einfacher Weg ist, die aufgenommenen Frames als JPEG-Bilder der Reihe nach an den Browser zu senden. FFmpeg kann z.B. mit `-c:v mjpeg -f mpjpeg` einen MJPEG-Stream ausgeben. Über einen WebSocket lässt sich dieser streamen, indem jede JPEG-Datei als Binärnachricht geschickt wird. Im Frontend könnte man einen `<canvas>` verwenden und per JavaScript die ankommenden JPEG-Blobs regelmäßig darstellen. Alternativ kann man einen MJPEG-Stream auch als HTTP-Multipart-Stream an einen `<img>`-Tag senden (das `<img>` lädt dann kontinuierlich neue Bilder nach). Über WebSocket muss man diesen Mechanismus allerdings selbst nachbilden. Vorteil: MJPEG ist einfach zu handhaben (jeder Frame ist unabhängig). Nachteil: die Bandbreite ist relativ hoch, da JPEG weniger effizient ist als Videocodecs.

- **MPEG-1 + WebSocket (JSMpeg)**: Ein bewährter Ansatz für Live-Streaming im Browser ist die Verwendung der JavaScript-Bibliothek **JSMpeg** [9] [10]. Dabei encodiert man den Stream mit FFmpeg als MPEG-TS mit MPEG-1 Video (und optional MP2 Audio) und schickt diesen kontinuierlich an einen WebSocket-Relay-Server. FFmpeg könnte z.B. so gestartet werden:

```
ffmpeg -f gdigrab -framerate 20 -i desktop -f mpegts -codec:v mpeg1video -s
1920x1080 -b:v 1500k -bf 0 -codec:a mp2 -ar 44100 -ac 1 -b:a 128k http://
localhost:8081/feed1
```

Hierbei würde ein einfacher Node.js-WebSocket-Relay (wie im JSMpeg-Projekt mitgeliefert) den MPEG-TS-Datenstrom von FFmpeg empfangen und an verbundene Browser weiterverteilen [11]. Im Browser rendert dann JSMpeg den Stream in ein Canvas-Element. Dieses Vorgehen hat relativ geringe Latenz (einige hundert Millisekunden) und ist leichter umzusetzen als eigenes WebRTC. Beachte, dass MPEG1 bei höheren Auflösungen mehr Bitrate braucht, um gute Qualität zu liefern [12] – ggf. muss man die Auflösung oder Qualität anpassen.

- **WebRTC**: Für **sehr geringe Latenzen** und effizientere Kompression kann WebRTC eingesetzt werden. Hierfür muss allerdings ein WebRTC Peer-Verbindung aufgebaut werden. FFmpeg selbst kann keinen kompletten WebRTC-Handshake durchführen, aber man könnte FFmpeg einen RTP-Stream senden lassen und mit einem WebRTC-Gateway koppeln. Zum Beispiel könnte man einen

Medienserver (Janus, mediasoup o. Ä.) oder eine Library (wie Pion in Go, aiortc in Python) einsetzen, die einen WebRTC-Peer stellt. FFmpeg würde dann den Desktop an diesen Peer via RTP streamen (H.264 oder VP8/VP9 encodiert). Der Browser könnte den Strom mit `<video>` Tag direkt abspielen, da es ein natives WebRTC-Medienstream ist. Dieses Setup ist am komplexesten, bietet aber die besten Ergebnisse, wenn Interaktivität in Echtzeit gefordert ist. Für erste Tests ist es oft einfacher, MJPEG oder JSMpeg zu verwenden, da diese kein spezielles Signaling benötigen.

- **Video-Element mit MSE/HLS**: Ein weiterer Ansatz (wenngleich mit höherer Latenz) ist, FFmpeg H.264-Segmente erzeugen zu lassen und per HTTP (HLS/DASH) bereitzustellen. Der Browser könnte diese mit einem `<video>` Element oder über **Media Source Extensions (MSE)** abspielen. Dies ist aber eher für Streaming mit einigen Sekunden Verzögerung sinnvoll und weniger für eine Analyseumgebung, die nah am Echtzeit-Bild sein soll.

Wichtig: Da FFmpeg nicht direkt als WebSocket-Server fungieren kann, ist bei WebSocket-Lösungen immer ein kleiner **Relay-Server** nötig [13]. In der Praxis würde Ihre Anwendung also folgendermaßen aussehen:

1. **Capture starten** – z.B. FFmpeg auf dem Windows-Rechner ausführen, welches den Bildschirm aufzeichnet und den codierten Stream an einen Socket/HTTP schreibt.
2. **Relay/Server** – ein Node.js-Server oder ein anderes WebSocket-fähiges Backend empfängt diesen Stream (z.B. via HTTP POST oder TCP) und verteilt die Daten als WebSocket-Broadcast an die Browser-Clients. (Im JSMpeg-Beispiel übernimmt `websocket-relay.js` genau diese Aufgabe [14].)
3. **Frontend** – Die React-Webanwendung verbindet sich per WebSocket und empfängt den Videostrom. Je nach gewähltem Format dekodiert sie ihn: Bei MJPEG könnten ankommende Frames direkt in `<img>` /Canvas geladen werden, bei JSMpeg übernimmt die Library die Decodierung ins Canvas, bei WebRTC würde der Browser die Decodierung nativ im `<video>` Element machen.

Bekannte Einschränkungen von Windows 11 beim Capturing: Abschließend sei betont, dass Windows' Sicherheitseinstellungen gewisse Aufnahmen verhindern können. In Unternehmensumgebungen gibt es z.B. eine *"Screen Capture Protection"*, die Aufzeichnungen von bestimmten Fenstern oder Anwendungen blockiert. Für den hier beschriebenen Anwendungsfall sollte das aber normalerweise nicht greifen, solange Sie Anwendungen auf Ihrem eigenen Desktop aufnehmen. Achten Sie dennoch darauf, dass keine DRM-geschützten Inhalte gestreamt werden, da diese von Windows evtl. als schwarzer Bildschirm erscheinen.

Zusammenfassung der Empfehlungen: Nutzen Sie FFmpeg's Desktop-Capture (ggf. mit dem DXGI/`ddagrab`-Input für bessere Performance) um beide Monitore als großes Bild zu erfassen [1]. Stellen Sie sicher, dass die zu analysierenden Inhalte auch tatsächlich sichtbar sind – aufgrund von Windows-Restrictions können versteckte virtuelle Desktops nicht parallel gestreamt werden [6]. Planen Sie gegebenenfalls mit einem zweiten System/VM, wenn mehrere isolierte Umgebungen gebraucht werden. Für den Stream zum Browser sind MJPEG (einfach) oder ein WebSocket + Canvas-Lösung (z.B. mit JSMpeg) gute Startpunkte, während WebRTC für fortgeschrittene Anforderungen die professionellste (aber aufwändigere) Option ist. Mit dieser Kombination aus Tools und Einstellungen sollte ein funktionierendes Setup möglich sein, um alle gewünschten Bildschirminhalte gleichzeitig ins Frontend zu übertragen und dort weiter zu analysieren.

Quellen: FFmpeg Doku und Beispiele [1] [2], Microsoft/Community-Erfahrungen zu virtuellen Desktops [6] [8], sowie JSMpeg-Dokumentation für WebSocket-Streaming [11].

1 Capture Windows screen with ffmpeg - Stack Overflow
https://stackoverflow.com/questions/6766333/capture-windows-screen-with-ffmpeg

2 Requesting tips and tricks : r/ffmpeg
https://www.reddit.com/r/ffmpeg/comments/x2eetl/requesting_tips_and_tricks/

3 4 5 [FFmpeg-cvslog] avfilter: add vsrc_ddagrab
https://lists.ffmpeg.org/pipermail/ffmpeg-cvslog/2022-July/133050.html

6 8 c++ - Screenshot in Windows 10 and multiple desktops - Stack Overflow
https://stackoverflow.com/questions/45143835/screenshot-in-windows-10-and-multiple-desktops

7 Window capture on windows 10 virtual desktop | OBS Forums
https://obsproject.com/forum/threads/window-capture-on-windows-10-virtual-desktop.132275/

9 10 11 12 13 14 GitHub - phoboslab/jsmpeg: MPEG1 Video Decoder in JavaScript
https://github.com/phoboslab/jsmpeg