COMPUTER VISION

3D Vision & Multi-Camera Systems

A practical deep dive into stereo vision, depth estimation, point clouds, and synchronized multi-camera pipelines for production computer vision — from camera calibration to real-time 3D reconstruction.

Dr. Farshid Pirahansiah · pirahansiah.com

- MASt3R-SLAM: Real-Time Dense SLAM with 3D Reconstruction Priors
-

Real-Time 3D Point Cloud Generation and Visualization from Depth Data

The fusion of depth sensing and 3D visualization opens remarkable possibilities for interactive applications. By converting 2D depth maps into 3D point clouds, we can build systems that bridge physical and digital realms in real-time.

Depth to 3D Conversion

The foundation of this approach lies in the deprojection process - transforming pixel coordinates and their associated depth values into 3D space. This requires camera intrinsic parameters (focal length, principal point) to perform the perspective transformation:

```
def deproject_point(u, v, depth, camera_matrix):
    fx = camera_matrix[0, 0]  # Focal length X
    fy = camera_matrix[1, 1]  # Focal length Y
    cx = camera_matrix[0, 2]  # Principal point X
    cy = camera_matrix[1, 2]  # Principal point Y

    # Convert to 3D coordinates
    x = (u - cx) * depth / fx
    y = (v - cy) * depth / fy
    z = depth

    return np.array([x, y, z])
```

Real-Time Visualization Strategies

Visualizing 3D data interactively requires threading to prevent blocking the main application loop. A separate thread can handle display updates while maintaining responsive input handling:

```
def start_visualizer_thread():
    global visualizer_thread, visualizer_active
    visualizer_active = True
    visualizer_thread = threading.Thread(target=visualizer_loop)
    visualizer_thread.daemon = True
    visualizer_thread.start()
```

Multi-Camera Fusion

Depth data from multiple cameras can create a more complete 3D representation. This requires transformation matrices to convert points between coordinate systems:

```
def transform_point(point, matrix):
    point_homog = np.append(point, 1.0)  # Convert to homogeneous coordinates
    transformed = np.dot(matrix, point_homog)
    return transformed[:3]  # Return Cartesian coordinates
```

Trajectory Analysis

By maintaining a history of 3D positions, we can analyze trajectories to detect motion patterns. This enables advanced behaviors like distinguishing between stationary and moving objects:

```
def detect_motion_from_point_cloud():
    if len(position_history) < 2:
        return False

    threshold_movement = 0.05
    threshold_time = 0.5

    recent_time, recent_pos = position_history[-1]
    for i in range(len(position_history)-2, -1, -1):
        prev_time, prev_pos = position_history[i]
        time_diff = recent_time - prev_time
        if time_diff > threshold_time:
            break

        position_diff = np.linalg.norm(recent_pos - prev_pos)
        if position_diff > threshold_movement:
            return True

    return False
```

Robust Depth Sampling

Raw depth data often contains noise and gaps. Implementing window-based analysis improves reliability:

```
def get_depth_at_point(depth_image, u, v, depth_scale, window_size=7):
    h, w = depth_image.shape[:2]

    # Extract window around point
    x_min = max(0, u - window_size // 2)
    x_max = min(w, u + window_size // 2 + 1)
    y_min = max(0, v - window_size // 2)
    y_max = min(h, v + window_size // 2 + 1)

    window = depth_image[y_min:y_max, x_min:x_max]

    # Filter valid depths
    valid_depths = window[(window > 100) & (window < 65000)]

    if valid_depths.size > 0:
        return np.median(valid_depths) * depth_scale

    return 0
```

Visualization Options

Different visualization approaches offer tradeoffs between performance and visual fidelity:

1. **Real-time interactive display** using Matplotlib or Open3D
2. **Static image rendering** for systems with limited GUI capabilities
3. **3D file export** (PLY, OBJ) for offline analysis in specialized software

The choice depends on the specific requirements of your application and the computational resources available.

By combining these techniques, we can create systems that understand and respond to motion in three-dimensional space, opening possibilities for contactless interfaces, motion analysis, and spatial computing applications.

🚨 Reality Check: Why This Is Hard

- **Bandwidth**: 100 HD (720p) cameras @ 30 FPS ≈ 6.5 Gbps raw data (uncompressed).
- **CPU/GPU Load**: Decoding 100 video streams simultaneously requires massive parallel compute.
- **USB/PCIe Bottlenecks**: USB controllers share bandwidth. PCIe lanes limit capture cards.
- **Memory**: 100 frames in memory ≈ 2–4 GB just for buffers (depending on resolution).

- **Latency & Sync**: Hardware-level sync (e.g., genlock) is needed for true frame alignment.

✅ Goals Clarification

Define what “sync” means:

1. **Temporal Sync (Frame Alignment)**: All cameras capture frames at the same physical time.
2. **Software Sync (Display/Processing Sync)**: All frames are read/displayed together in your app.
3. **Trigger Sync**: Cameras start/stop recording simultaneously.

We’ll focus on **software sync** for display/processing using OpenCV. True hardware sync requires specialized cameras and capture hardware.

🧩 Architecture Overview

```
[100 Webcams]
     ↓ (USB / Ethernet / PCIe)
[Windows Machine(s) + Capture Hardware]
     ↓
[Multi-threaded Capture Layer (C++/Python)]
     ↓
[Frame Buffer + Synchronization Queue]
     ↓
[Display / Processing Thread (OpenCV GUI / Analysis)]
```

🖥️ Option 1: Single Machine (Theoretical, Not Recommended)

Hardware Requirements

- Multiple PCIe USB 3.0/3.1 expansion cards (each with independent controllers).
- Possibly multiple capture cards if using SDI/HDMI cameras.
- 64+ GB RAM, 32+ logical cores, high-end GPU (RTX 4090 or multi-GPU).
- NVMe SSD for buffering if needed.

Software Stack (C++ Recommended)

Step 1: Multi-threaded Camera Capture

Use `std::thread` per camera (or thread pool) to avoid blocking.

```cpp
#include <opencv2/opencv.hpp>
#include <thread>
#include <vector>
#include <mutex>
#include <queue>
#include <condition_variable>

struct FramePackage {
    int cam_id;
    cv::Mat frame;
    double timestamp;
};

std::mutex queue_mutex;
std::condition_variable frame_cv;
std::queue<FramePackage> frame_queue;
bool shutdown = false;

void capture_thread(int cam_id) {
    cv::VideoCapture cap(cam_id);
    if (!cap.isOpened()) {
        std::cerr << "Cannot open camera " << cam_id << std::endl;
        return;
    }

    // Optional: Set lower resolution for performance
    cap.set(cv::CAP_PROP_FRAME_WIDTH, 640);
    cap.set(cv::CAP_PROP_FRAME_HEIGHT, 480);
    cap.set(cv::CAP_PROP_FPS, 15);

    while (!shutdown) {
        cv::Mat frame;
        if (!cap.read(frame)) continue;

        FramePackage pkg{cam_id, frame.clone(), (double)cv::getTickCount() /
cv::getTickFrequency()};

        {
            std::lock_guard<std::mutex> lock(queue_mutex);
            frame_queue.push(pkg);
        }
        frame_cv.notify_one();
```

```
    }
}
```

Step 2: Synced Frame Display / Processing

Maintain a buffer of latest frame per camera. Wait until all 100 are updated, then display.

```cpp
std::vector<cv::Mat> latest_frames(100);
std::vector<bool> frame_ready(100, false);
std::mutex display_mutex;

void sync_display_thread() {
    while (!shutdown) {
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
            frame_cv.wait(lock, [] { return !frame_queue.empty(); });

            while (!frame_queue.empty()) {
                FramePackage pkg = frame_queue.front();
                frame_queue.pop();

                {
                    std::lock_guard<std::mutex> dlock(display_mutex);
                    latest_frames[pkg.cam_id] = pkg.frame;
                    frame_ready[pkg.cam_id] = true;
                }
            }
        }

        // Check if all 100 frames are ready
        bool all_ready = true;
        {
            std::lock_guard<std::mutex> dlock(display_mutex);
            for (bool r : frame_ready) {
                if (!r) { all_ready = false; break; }
            }
        }

        if (all_ready) {
            // Display or process synced batch
            display_grid(latest_frames); // You implement this

            // Reset for next cycle
            std::lock_guard<std::mutex> dlock(display_mutex);
            std::fill(frame_ready.begin(), frame_ready.end(), false);
        }
    }
}
```

Step 3: Main

```
int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 100; ++i) {
        threads.emplace_back(capture_thread, i);
    }

    std::thread display_thread(sync_display_thread);

    // Let run...
    std::this_thread::sleep_for(std::chrono::minutes(10));

    shutdown = true;
    for (auto& t : threads) t.join();
    display_thread.join();

    return 0;
}
```

🐍 Python Version (Not Recommended for 100 Cameras)

Python's GIL and OpenCV overhead make it unsuitable for 100 real-time streams. But for <20 cameras, you can try:

```python
import cv2
import threading
import queue
import numpy as np

frame_buffers = [None] * 100
frame_ready = [False] * 100
lock = threading.Lock()

def capture(cam_id):
    cap = cv2.VideoCapture(cam_id)
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    while True:
        ret, frame = cap.read()
        if not ret: continue

        with lock:
            frame_buffers[cam_id] = frame.copy()
            frame_ready[cam_id] = True

threads = []
for i in range(100):
    t = threading.Thread(target=capture, args=(i,), daemon=True)
    t.start()
    threads.append(t)

while True:
    with lock:
        if all(frame_ready):
            # Display grid (implement your own)
            # display_grid(frame_buffers)

            # Reset flags
            frame_ready = [False] * 100
```

⚠️ This will likely crash or lag severely beyond 10–20 cameras.

🌐 Option 2: Distributed System (Recommended)

Use multiple PCs (e.g., 10 machines × 10 cameras each).

- Each machine captures and preprocesses 10 cameras.
- Send compressed frames (JPEG/MJPEG/H.264) over network to central machine.
- Central machine decodes and displays synced grid.

Tools:

- **ZMQ** or **gRPC** for low-latency streaming.
- **FFmpeg** for hardware-accelerated encoding/decoding.
- **OpenCV** + **CUDA** for GPU decoding.

🎯 Optimization Tips

1. Reduce Resolution & FPS

```
cap.set(cv::CAP_PROP_FRAME_WIDTH, 320);
cap.set(cv::CAP_PROP_FRAME_HEIGHT, 240);
cap.set(cv::CAP_PROP_FPS, 10);
```

2. Use MJPEG or H.264 if supported

```
cap.set(cv::CAP_PROP_FOURCC, cv::VideoWriter::fourcc('M','J','P','G'));
```

3. Enable Hardware Acceleration (if available)

Use `cv::cudacodec::VideoReader` for GPU decoding (NVIDIA).

4. Use Memory-Efficient Buffers

Reuse `cv::Mat` with `.copyTo()` or pre-allocated buffers.

5. Disable Auto-Focus / Auto-Exposure

Reduces per-frame processing delay.

```
cap.set(cv::CAP_PROP_AUTOFOCUS, 0);
cap.set(cv::CAP_PROP_AUTO_EXPOSURE, 0);
```

🧪 Testing Strategy

1. Start with 5 cameras → profile CPU, memory, USB bandwidth.
2. Scale to 10 → 20 → monitor bottlenecks.
3. Use tools: Windows Task Manager, `USBTreeView`, `GPU-Z`, `RAMMap`.
4. Consider industrial cameras with external trigger (e.g., FLIR, Basler) for true sync.

🧰 Alternative Libraries & Tools

- **DirectShow / Media Foundation (Windows)** — lower-level, more control.
- **Spinnaker SDK (FLIR)** — for industrial cameras with hardware sync.
- **Aravis (GigE Vision)** — Linux but useful conceptually.
- **Pylon (Basler)** — excellent multi-camera sync support.
- **HALCON / LabVIEW** — commercial but robust for multi-cam.

🖼️ Displaying 100 Frames

Use `cv::imshow` with a tiled grid:

```
cv::Mat create_grid(const std::vector<cv::Mat>& frames, int grid_w = 10) {
    int grid_h = (frames.size() + grid_w - 1) / grid_w;
    int tile_h = frames[0].rows;
    int tile_w = frames[0].cols;

    cv::Mat grid = cv::Mat::zeros(tile_h * grid_h, tile_w * grid_w, CV_8UC3);

    for (int i = 0; i < frames.size(); ++i) {
        int r = i / grid_w;
        int c = i % grid_w;
        cv::Rect roi(c * tile_w, r * tile_h, tile_w, tile_h);
        frames[i].copyTo(grid(roi));
    }
    return grid;
}
```

Then: `cv::imshow("Synced Grid", grid);`

✅ Final Recommendations

Task	Recommendation
< 10 cameras	Python + OpenCV (threaded)
10–50 cameras	C++ + OpenCV + Multi-threading
50–100+ cameras	Distributed system + industrial cameras + hardware sync
True frame sync	Use cameras with genlock/trigger input (e.g., Basler/FLIR)
Display only	Downscale, use MJPEG, drop frames if needed
Processing	Offload to GPU or separate machines

🧠 Why OpenCV VideoCapture Fails at Scale

- Uses **generic DirectShow or MF backend** with no control.
- Forces **RGB conversion** (expensive for 100 streams).
- No access to **compressed buffers** (MJPEG/H.264).
- No **zero-copy** or **GPU texture sharing**.
- No **hardware sync or buffer timestamps**.

✅ What You Should Do Instead

🎯 GOAL: Capture 100 cameras → Keep in compressed YUV/MJPEG → Decode on GPU → Sync timestamps → Display/Process

---

🧩 PART 1: Use Media Foundation (MF) Directly — NOT OpenCV

OpenCV's `cv::VideoCapture` is a black box. You need **direct MF Source Reader** access.

✅ Advantages of MF Source Reader:

- Enumerate cameras with `MFEnumDeviceSources`
- Query **native formats**: MJPEG, YUY2, NV12, H.264
- Request **compressed samples** (avoid RGB conversion)
- Get **precise timestamps** (`IMFSample::GetSampleTime`)
- Use **hardware MJPEG decoder** via `IMFTransform`
- Zero-copy to GPU via `IMFDXGIBuffer`

🧰 Step 1: Initialize MF and Enumerate Cameras

```cpp
#include <mfapi.h>
#include <mfidl.h>
#include <mfreadwrite.h>
#pragma comment(lib, "mfplat.lib")
#pragma comment(lib, "mfreadwrite.lib")
#pragma comment(lib, "mfuuid.lib")

HRESULT EnumerateCameras(std::vector<IMFActivate*>& devices) {
    IMFAttributes* pConfig = nullptr;
    IMFActivate** ppDevices = nullptr;
    UINT32 count = 0;

    MFCreateAttributes(&pConfig, 1);
    pConfig->SetGUID(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);

    MFEnumDeviceSources(pConfig, &ppDevices, &count);

    for (UINT32 i = 0; i < count; ++i) {
        devices.push_back(ppDevices[i]); // Caller must Release()
    }

    CoTaskMemFree(ppDevices);
    pConfig->Release();
    return S_OK;
}
```

🎞️ Step 2: Create Source Reader + Request MJPEG/YUV

```
IMFSourceReader* CreateReader(IMFActivate* pActivate) {
    IMFSourceReader* pReader = nullptr;
    IMFMediaSource* pSource = nullptr;

    pActivate->ActivateObject(IID_PPV_ARGS(&pSource));
    MFCreateSourceReaderFromMediaSource(pSource, nullptr, &pReader);

    // Set output format to MJPEG (if supported)
    IMFMediaType* pType = nullptr;
    MFCreateMediaType(&pType);
    pType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
    pType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_MJPG); // or MFVideoFormat_YUY2 / NV12
    pType->SetUINT32(MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
    pType->SetUINT32(MF_MT_FRAME_RATE, 30 << 16); // 30 FPS (QWORD = 30 * 2^16)

    pReader->SetCurrentMediaType((DWORD)MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pType);
    pType->Release();

    pSource->Release();
    return pReader;
}
```

> 💡 Tip: Use `MFVideoFormat_YUY2` if MJPEG not available — still better than RGB.

📦 Step 3: Read Compressed Sample + Get Timestamp

```cpp
struct CameraFrame {
    int cam_id;
    LONGLONG timestamp; // 100ns units
    std::vector<uint8_t> data; // compressed MJPEG or raw YUV
    DWORD data_size;
    GUID format; // e.g., MFVideoFormat_MJPG
};

void ReadCameraLoop(int cam_id, IMFSourceReader* pReader, std::queue<CameraFrame>& q,
std::mutex& mtx) {
    while (true) {
        DWORD streamIndex, flags;
        LONGLONG llTimestamp;
        IMFSample* pSample = nullptr;

        HRESULT hr = pReader->ReadSample(
            MF_SOURCE_READER_FIRST_VIDEO_STREAM,
            0, &streamIndex, &flags, &llTimestamp, &pSample);

        if (flags & MF_SOURCE_READERF_ENDOFSTREAM) break;
        if (!pSample) continue;

        IMFMediaBuffer* pBuffer = nullptr;
        pSample->ConvertToContiguousBuffer(&pBuffer);

        BYTE* pData = nullptr;
        DWORD cbMaxLength, cbCurrentLength;
        pBuffer->Lock(&pData, &cbMaxLength, &cbCurrentLength);

        CameraFrame frame;
        frame.cam_id = cam_id;
        frame.timestamp = llTimestamp;
        frame.data.assign(pData, pData + cbCurrentLength);
        frame.data_size = cbCurrentLength;

        // Get subtype to know if it's MJPEG/YUY2/etc.
        IMFMediaType* pType = nullptr;
        pReader->GetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, &pType);
        pType->GetGUID(MF_MT_SUBTYPE, &frame.format);
        pType->Release();

        pBuffer->Unlock();
```

```
        pBuffer->Release();
        pSample->Release();

        {
            std::lock_guard<std::mutex> lock(mtx);
            q.push(frame);
        }

        // Optional: throttle or drop frames if queue too large
    }
}
```

🖥️ Step 4: GPU-Accelerated MJPEG → RGB Decoding (Optional)

If you must display/process in RGB — **DO NOT USE CPU**. Use **Direct3D11 + DXVA** or **NVDEC** via `IMFTransform`.

```
// Create MJPEG decoder MFT
IMFTransform* pDecoder = nullptr;
CoCreateInstance(CLSID_MJPEGDecoderMFT, nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&pDecoder));

// Set input type (MJPEG)
IMFMediaType* pInputType = nullptr;
MFCreateMediaType(&pInputType);
pInputType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
pInputType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_MJPG);
pInputType->SetUINT32(MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
pDecoder->SetInputType(0, pInputType, 0);

// Set output type (NV12 or RGB32 — GPU-friendly)
IMFMediaType* pOutputType = nullptr;
MFCreateMediaType(&pOutputType);
pOutputType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
pOutputType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_NV12); // or RGB32
pOutputType->SetUINT32(MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
pDecoder->SetOutputType(0, pOutputType, 0);
```

Then feed `IMFSample` into `pDecoder->ProcessInput()`, get `ProcessOutput()` → GPU texture.

> 🚀 You can even share `ID3D11Texture2D` with OpenCV via `cv::cuda::GpuMat` or DirectX interop.

🔄 PART 2: Sync Strategy — “Good Enough” Software Sync

Since you’re getting **hardware timestamps** (`llTimestamp` in 100ns units), you can:

1. Collect frames from all 100 cams.
2. Wait until you have one frame from each cam within a **±16ms window** (for 60Hz).
3. Pick the closest matching set → display together.

```cpp
struct SyncedBatch {
    std::vector<cv::Mat> frames; // or GPU textures
    LONGLONG ref_timestamp;
};

std::vector<std::queue<CameraFrame>> per_cam_queues(100);
std::mutex queue_mutex;

SyncedBatch WaitForSyncedBatch() {
    SyncedBatch batch;
    batch.frames.resize(100);

    while (true) {
        std::vector<LONGLONG> latest_ts(100, -1);

        {
            std::lock_guard<std::mutex> lock(queue_mutex);
            for (int i = 0; i < 100; ++i) {
                if (!per_cam_queues[i].empty()) {
                    latest_ts[i] = per_cam_queues[i].back().timestamp;
                }
            }
        }

        // Find median or mode timestamp
        auto valid_ts = latest_ts | std::views::filter([](auto t) { return t >= 0; });
        if (valid_ts.size() < 100) continue;

        std::vector<LONGLONG> ts_vec(valid_ts.begin(), valid_ts.end());
        std::sort(ts_vec.begin(), ts_vec.end());
        LONGLONG median_ts = ts_vec[ts_vec.size()/2];

        // Check if all frames are within ±1 frame time (e.g., ±33ms for 30fps)
        bool all_in_window = true;
        for (int i = 0; i < 100; ++i) {
            if (std::abs(latest_ts[i] - median_ts) > 3300000) { // 33ms in 100ns units
                all_in_window = false;
                break;
            }
        }

        if (all_in_window) {
```

```
            batch.ref_timestamp = median_ts;
            for (int i = 0; i < 100; ++i) {
                CameraFrame& f = per_cam_queues[i].front();
                // Decode MJPEG → GPU → cv::cuda::GpuMat or upload to texture
                batch.frames[i] = DecodeFrameOnGPU(f); // You implement
                per_cam_queues[i].pop();
            }
            return batch;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

🖼️ PART 3: Display — Use DirectX, Not OpenCV HighGUI

OpenCV's `imshow` is **not designed for 100 streams**. Use **Direct3D11** to render 100 textures in a grid.

```
// Create 10x10 texture grid
// Upload each decoded frame as shader resource
// Render full-screen quad with pixel shader sampling grid

// Bonus: Use compute shader to process frames on GPU without CPU round-trip!
```

> 🎮 Example: Render using `ID3D11DeviceContext::PSSetShaderResources()` + custom HLSL.

🚀 PART 4: GStreamer Interop (Optional)

If you prefer GStreamer (e.g., for network streaming or advanced pipelines):

```
gst-launch-1.0 ksvideosrc device-path="\\?\usb#vid_XXXX&pid_YYYY#..." !
image/jpeg,framerate=30/1 ! ...
```

You can:

- Use `appsrc` to feed MF buffers into GStreamer pipeline.
- Use `GstBuffer` with `GST_BUFFER_DTS` for sync.
- Use `nvjpegdec` or `d3d11h264dec` for GPU decoding.

But **on Windows, MF is more native and performant** for local cameras.

📊 Bandwidth & Performance Estimates

Format	640x480 @ 30 FPS	100 Cameras
Uncompressed RGB24	~26.4 MB/frame → 7.9 GB/s	❌ Impossible
YUY2 (4:2:2)	~0.6 MB/frame → 1.8 GB/s	✅ Possible with PCIe 3.0 x8
MJPEG (5:1 compressed)	~0.12 MB/frame → 360 MB/s	✅✅ Easily fits USB 3.2 Gen 2

> 💡 Use **MJPEG** if available — reduces bandwidth 5–10x.

🧰 Tools to Debug & Profile

- **GraphEdit / GraphStudioNext** — visualize DirectShow/MF graphs.
- **MediaFoundation.NET** — C# wrapper if you want rapid prototyping.
- **Windows Performance Analyzer (WPA)** — profile MF pipeline stalls.
- **USBTreeView** — check which USB controller each cam is on.
- **GPUView** — analyze GPU decode/display latency.

✅ Final Architecture Summary

```
[100 Webcams (MJPEG/YUY2)]
       ↓
[Media Foundation Source Reader — 1 thread per cam]
       ↓
[Compressed Samples + HW Timestamps → Queue]
       ↓
[Sync Engine: Match frames within ±1 frame time]
       ↓
[GPU MJPEG Decoder (DXVA/NVDEC) → Direct3D11 Texture]
       ↓
[Direct3D11 Renderer: 10x10 Grid of Textures]
       ↓
[Display @ 30 FPS — Zero CPU copy, GPU-accelerated]
```

🧪 Sample Project Structure

```
/src
  /capture
    MFSourceReaderWrapper.h/cpp
    CameraManager.h/cpp
  /sync
    TimestampSync.h/cpp
  /decode
    MJPEGDecoderD3D11.h/cpp
  /render
    D3D11GridRenderer.h/cpp
  main.cpp
```

OpenCV Interop with Direct3D Textures
If you *must* use OpenCV for processing:
``cpp cv::cuda::GpuMat gpu_mat; // Use CUDA External Memory to wrap ID3D11Texture2D // Requires CUDA 10+ and WDDM 2.0
// OR — slower but simpler: cv::Mat cpu_mat = cv::Mat(h, w, CV_8UC3); CopyTextureToCPU(d3d_texture, cpu_mat.data); // via staging texture gpu_mat.upload(cpu_mat); // then process on GPU ```
✅ GOAL: Real-Time Sync of 100 Cameras — Minimal Delay, Deterministic Latency
✔ What “Sync” Really Means Here

- All 100 cameras capture frames within ≤ 1ms of each other (software sync). - No buffering delay — frame N from cam0 to cam99 are displayed/processed together. - Minimal end-to-end latency (capture → decode → display ≤ 33ms @ 30fps). - Deterministic pipeline — no random stalls or buffer bloat.

🆚 MF vs GStreamer for 100-Cam Sync

Feature	Media Foundation (MF)	GStreamer (Windows)
Latency Control	Medium — buffering tunable but not always exposed	✅ **High — full pipeline control**
Buffering	Auto-buffers (can be reduced)	✅ **Can be disabled or set to 0**
Hardware Timestamps	✅ Yes (IMFSample)	✅ Yes (GST_BUFFER_PTS/DTS)
Zero-Copy	✅ Yes (IMFDXGIBuffer → D3D11)	✅ Yes (d3d11, nvdec, dmabuf)
MJPEG/H.264 HW Decode	✅ DXVA	✅ d3d11h264dec, nvjpegdec
Pipeline Determinism	❌ OS/Driver dependent	✅ Fully scriptable & tunable
Cross-Platform	❌ Windows only	✅ Linux/macOS/Windows
Ease of Tuning	❌ COM APIs, complex	✅ gst-launch, caps, properties

> 🏆 **Winner for Real-Time Sync: GStreamer** — if you tune it right.

🚀 BEST SOLUTION: GStreamer with Zero Buffering + Hardware Sync + GPU Decode

Here's how to build a **real-time, low-latency, synced 100-camera pipeline** using **GStreamer on Windows**.

🥾 STEP 1: Use `ksvideosrc` with `do-timestamp=true` + `num-buffers=1`

```
gst-launch-1.0 ksvideosrc device-path="\\?\usb#vid_XXXX&pid_YYYY#..." do-timestamp=true ! \
    image/jpeg,framerate=30/1,width=640,height=480 ! \
    queue max-size-buffers=1 max-size-bytes=0 max-size-time=0 leaky=2 ! \
    jpegparse ! d3d11jpegdec ! \
    videoconvert ! video/x-raw,format=BGRA ! \
    appsink name=sink emit-signals=true max-buffers=1 drop=true sync=false
```

🔑 Key Flags for Real-Time Sync:

- `do-timestamp=true` → Uses **hardware timestamps** from camera driver.
- `queue max-size-buffers=1 leaky=2` → Only keep newest frame, drop old ones.
- `max-buffers=1 drop=true` on `appsink` → Never buffer, drop if not consumed.
- `sync=false` → Don't wait for clock, push immediately.

> 💡 This ensures **no buffering delay** — each frame is processed as soon as captured.

🖥️ STEP 2: C++ Integration — Pull Frames via `appsink`

```cpp
#include <gst/gst.h>
#include <gst/app/gstappsink.h>

struct CameraStream {
    int id;
    GstElement* pipeline;
    GstElement* appsink;
};

GstFlowReturn on_new_sample(GstAppSink* sink, gpointer user_data) {
    CameraStream* cam = (CameraStream*)user_data;
    GstSample* sample = gst_app_sink_pull_sample(sink);
    if (!sample) return GST_FLOW_OK;

    GstBuffer* buffer = gst_sample_get_buffer(sample);
    GstCaps* caps = gst_sample_get_caps(sample);

    // Get hardware timestamp
    GstClockTime pts = GST_BUFFER_PTS(buffer); // ← 🔥 THIS IS YOUR SYNC KEY

    // Map buffer (zero-copy if possible)
    GstMapInfo map;
    if (gst_buffer_map(buffer, &map, GST_MAP_READ)) {
        // Copy or zero-copy to GPU/D3D11 texture
        // Use format from caps (e.g., BGRA, NV12)

        CameraFrame frame;
        frame.cam_id = cam->id;
        frame.timestamp_ns = pts; // ← Use this for sync across 100 cams
        frame.data.assign(map.data, map.data + map.size);
        frame.format = parse_format_from_caps(caps);

        {
            std::lock_guard<std::mutex> lock(global_frame_mutex);
            global_frame_queue[cam->id].push(frame);
        }

        gst_buffer_unmap(buffer, &map);
    }

    gst_sample_unref(sample);
    return GST_FLOW_OK;
```

```
}

void setup_pipeline(CameraStream& cam, const std::string& device_path) {
    std::string pipeline_str = "ksvideosrc device-path=\"" + device_path + "\" do-
timestamp=true ! "
        "image/jpeg,framerate=30/1,width=640,height=480 ! "
        "queue max-size-buffers=1 max-size-bytes=0 max-size-time=0 leaky=2 ! "
        "jpegparse ! d3d11jpegdec ! "
        "videoconvert ! video/x-raw,format=BGRA ! "
        "appsink name=sink emit-signals=true max-buffers=1 drop=true sync=false";

    cam.pipeline = gst_parse_launch(pipeline_str.c_str(), nullptr);
    cam.appsink = gst_bin_get_by_name(GST_BIN(cam.pipeline), "sink");

    GstAppSinkCallbacks callbacks = { nullptr, nullptr, on_new_sample };
    gst_app_sink_set_callbacks(GST_APP_SINK(cam.appsink), &callbacks, &cam, nullptr);

    gst_element_set_state(cam.pipeline, GST_STATE_PLAYING);
}
```

🔄 STEP 3: Sync Engine — Match Frames by PTS

Same as before, but now using `GstClockTime` (nanoseconds):

```cpp
struct SyncedBatch {
    std::vector<cv::Mat> frames;
    GstClockTime ref_pts;
};

SyncedBatch WaitForSyncedBatch() {
    while (true) {
        std::vector<GstClockTime> latest_pts(100, GST_CLOCK_TIME_NONE);

        {
            std::lock_guard<std::mutex> lock(global_frame_mutex);
            for (int i = 0; i < 100; ++i) {
                if (!global_frame_queue[i].empty()) {
                    latest_pts[i] = global_frame_queue[i].back().timestamp_ns;
                }
            }
        }

        // Filter valid timestamps
        std::vector<GstClockTime> valid_pts;
        for (auto pt : latest_pts) if (pt != GST_CLOCK_TIME_NONE) valid_pts.push_back(pt);
        if (valid_pts.size() < 100) continue;

        // Find median PTS
        std::sort(valid_pts.begin(), valid_pts.end());
        GstClockTime median_pts = valid_pts[valid_pts.size()/2];

        // Check sync window: ±1 frame (33ms = 33,000,000 ns)
        bool all_synced = true;
        for (int i = 0; i < 100; ++i) {
            if (latest_pts[i] == GST_CLOCK_TIME_NONE ||
                std::abs((int64_t)(latest_pts[i] - median_pts)) > 33000000) {
                all_synced = false;
                break;
            }
        }

        if (all_synced) {
            SyncedBatch batch;
            batch.ref_pts = median_pts;

            for (int i = 0; i < 100; ++i) {
```

```
                auto& q = global_frame_queue[i];
                CameraFrame& f = q.front();

                // Upload to GPU texture or decode if needed
                batch.frames[i] = UploadToGPUTexture(f.data, f.format); // Zero-copy ideal

                q.pop();
            }
            return batch;
        }

        std::this_thread::sleep_for(1ms);
    }
}
```

🎯 STEP 4: Disable ALL Buffering — Force Real-Time

In Pipeline:

```
queue max-size-buffers=1 leaky=2
appsink max-buffers=1 drop=true sync=false
```

Also Set:

```
g_object_set(G_OBJECT(ksvideosrc), "device", device_path, nullptr);
g_object_set(G_OBJECT(ksvideosrc), "do-timestamp", TRUE, nullptr);

// Reduce latency in decoder
g_object_set(G_OBJECT(decoder), "low-latency", TRUE, nullptr);
g_object_set(G_OBJECT(decoder), "output-buffers", 1, nullptr); // if supported
```

🚫 STEP 5: Kill Windows Camera Frame Server (If Needed)

Windows 10/11 sometimes routes cameras through `Windows Camera Frame Server` — adds 1–2 frame delay.

Disable it:

1. Open **Task Manager** → Startup tab → Disable “Windows Camera Frame Server”.
2. Or via Registry/Group Policy if in enterprise.

> ⚠️ This may break Windows Camera app — acceptable for dedicated capture machines.

🖥️ STEP 6: Display — Use Direct3D11, Not OpenCV

Same as before — render 100 textures in grid using `ID3D11DeviceContext`.

GStreamer can output directly to D3D11 texture:

```
... ! d3d11jpegdec ! d3d11videosink
```

But for sync control, better to use `appsink` → upload to your own D3D11 texture pool.

🧪 BENCHMARK: Expected Latency

Stage	Latency
Camera Exposure → HW Timestamp	~0ms (hardware)
USB Transfer → GStreamer	~1–2ms
MJPEG Decode (GPU)	~2–5ms
Sync Wait (max)	~33ms (1 frame)
Display (D3D11 Flip)	~3ms
Total End-to-End	**≤ 40ms**

> ✅ This is **real-time** for 30 FPS (33ms/frame).

⚙️ GStreamer vs MF — Final Recommendation

Scenario	Recommendation
You need **maximum control, lowest latency, deterministic sync**	✅ **GStreamer**
You're stuck in pure Windows COM/MF environment	✅ MF (with `IMFSourceReader`, `SetStreamSelection`, `Flush`, low-latency profile)
You want **zero-copy to GPU**	✅ Both (MF: `IMFDXGIBuffer`, GStreamer: `d3d11` caps)
You want **cross-platform**	✅ GStreamer
You want **easiest integration with Python/ML**	✅ GStreamer + `gst-python` + `appsink` → NumPy

🛠️ HOW TO BUILD GSTREAMER ON WINDOWS

1. Download **GStreamer MinGW 64-bit** from https://gstreamer.freedesktop.org/download/
2. Install: `gstreamer-1.0`, `gst-plugins-good`, `gst-plugins-bad`, `gst-libav`
3. Enable: `ksvideosrc`, `d3d11`, `jpegparse`, `d3d11jpegdec`
4. Link in C++: `-lgstreamer-1.0 -lgstapp-1.0 -lgstvideo-1.0`

CMake:

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(GST REQUIRED gstreamer-1.0>=1.18 gstreamer-app-1.0 gstreamer-video-1.0)
target_link_libraries(your_app ${GST_LIBRARIES})
target_include_directories(your_app PRIVATE ${GST_INCLUDE_DIRS})
```

```
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, GLib
import threading
import numpy as np

Gst.init(None)

class CameraStream:
    def __init__(self, cam_id, device_path):
        self.id = cam_id
        self.pipeline = Gst.parse_launch(
            f'ksvideosrc device-path="{device_path}" do-timestamp=true ! '
            'image/jpeg,framerate=30/1,width=640,height=480 ! '
            'queue max-size-buffers=1 leaky=2 ! '
            'jpegparse ! jpegdec ! '
            'videoconvert ! video/x-raw,format=BGR ! '
            'appsink name=sink emit-signals=true max-buffers=1 drop=true sync=false'
        )
        self.appsink = self.pipeline.get_by_name('sink')
        self.appsink.connect('new-sample', self.on_new_sample)
        self.pipeline.set_state(Gst.State.PLAYING)

    def on_new_sample(self, sink):
        sample = sink.emit('pull-sample')
        buf = sample.get_buffer()
        caps = sample.get_caps()

        # Get PTS
        pts = buf.pts  # ← SYNC KEY

        # Extract to NumPy
        success, map_info = buf.map(Gst.MapFlags.READ)
        if success:
            h = caps.get_structure(0).get_value('height')
            w = caps.get_structure(0).get_value('width')
            arr = np.ndarray((h, w, 3), buffer=map_info.data, dtype=np.uint8).copy()
            buf.unmap(map_info)

            # Push to global sync queue with PTS
            with frame_lock:
                frame_queues[self.id].append((pts, arr))
```

```
        return Gst.FlowReturn.OK
```

Then sync in main thread same as C++ version.

✅ FINAL ANSWER: YES — Use GStreamer for Real-Time Sync

> ✅ **GStreamer is superior to MF for real-time, low-latency, synced 100-camera capture on Windows** — if you: - Use `ksvideosrc` with `do-timestamp=true` - Set `queue` and `appsink` to `max-buffers=1`, `drop=true`, `sync=false` - Use hardware timestamps (`PTS`) for frame alignment - Decode on GPU (`d3d11jpegdec`) - Render via Direct3D11 - Disable Windows Camera Frame Server

---

🚀 I can provide:

- Complete **Visual Studio 2022 C++ project** with GStreamer + D3D11 + sync engine
- Prebuilt **GStreamer pipelines for 100 cameras** with device enumeration
- **Python prototype** with `gi.repository.Gst`
- **Latency profiling tool** (measures PTS → display time)

✅ SHORT ANSWER

> ❗ **Media Foundation does NOT have a direct "no buffer" mode like GStreamer's `max-buffers=1 drop=true` — but you can simulate it by:**
>
> 1. **Flushing the stream before each read** → `pReader->Flush(...)`
> 2. **Using `MF_SOURCE_READER_IGNORE_CLOCK` + `MF_SOURCE_READER_DISABLE_DXVA`** for deterministic timing
> 3. **Reading in blocking mode + discarding all but the latest sample**
> 4. **Setting low-latency media types + disabling internal buffering where possible**

🧠 WHY MF BUFFERS BY DEFAULT

- MF Source Reader uses an **internal queue** to smooth out delivery (good for playback, bad for real-time sync).
- It respects **presentation clock** → adds latency to align with system time.
- USB camera drivers may buffer 1–3 frames internally (driver-dependent).

✅ STEP-BY-STEP: MINIMAL BUFFER / "NO BUFFER" MODE IN MF

✅ STEP 1: CREATE SOURCE READER WITH LOW-LATENCY FLAGS

```
IMFAttributes* pAttributes = nullptr;
MFCreateAttributes(&pAttributes, 2);

// ⚡ Critical: Ignore system clock → read samples as soon as available
pAttributes->SetUINT32(MF_READWRITE_ENABLE_HARDWARE_TRANSFORMS, TRUE);
pAttributes->SetUINT32(MF_SOURCE_READER_ASYNC_CALLBACK, FALSE); // Sync mode for control

// ⚠️ This is KEY: Ignore presentation clock → no waiting for "scheduled" time
pAttributes->SetUINT32(MF_SOURCE_READER_IGNORE_CLOCK, TRUE);

// Optional: Disable DXVA if you want CPU control (or keep if using GPU)
// pAttributes->SetUINT32(MF_SOURCE_READER_DISABLE_DXVA, TRUE);

MFCreateSourceReaderFromMediaSource(pSource, pAttributes, &pReader);
pAttributes->Release();
```

✅ STEP 2: SET LOW-LATENCY MEDIA TYPE (MJPEG/YUY2 + NO RGB)

```
IMFMediaType* pType = nullptr;
MFCreateMediaType(&pType);

pType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
pType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_MJPG); // or MFVideoFormat_YUY2
pType->SetUINT32(MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
pType->SetUINT32(MF_MT_FRAME_RATE, 30 << 16); // 30 FPS
pType->SetUINT32(MF_MT_FRAME_SIZE, (640 << 32) | 480); // width << 32 | height

// ⚡ Optional: Set low-latency flag (driver may respect it)
pType->SetUINT32(MF_MT_VIDEO_NOMINAL_RANGE, MFNominalRange_0_255); // Sometimes helps

pReader->SetCurrentMediaType((DWORD)MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr, pType);
pType->Release();
```

✅ STEP 3: FLUSH BEFORE EVERY READ → SIMULATE “NO BUFFER”

> 💡 This is the **most important trick** — flush the stream so you only get the *latest* frame.

```cpp
void ReadLatestFrameOnly(IMFSourceReader* pReader, CameraFrame& outFrame) {
    // 🚫 FLUSH stream → discard all buffered frames
    pReader->Flush(MF_SOURCE_READER_FIRST_VIDEO_STREAM);

    // 🚦 Now read — will block until next frame arrives
    DWORD streamIndex, flags;
    LONGLONG llTimestamp;
    IMFSample* pSample = nullptr;

    HRESULT hr = pReader->ReadSample(
        MF_SOURCE_READER_FIRST_VIDEO_STREAM,
        0, &streamIndex, &flags, &llTimestamp, &pSample);

    if (FAILED(hr) || !pSample) {
        if (pSample) pSample->Release();
        return;
    }

    // Extract buffer
    IMFMediaBuffer* pBuffer = nullptr;
    pSample->ConvertToContiguousBuffer(&pBuffer);

    BYTE* pData = nullptr;
    DWORD cbCurrentLength = 0;
    pBuffer->Lock(&pData, nullptr, &cbCurrentLength);

    // Copy or zero-copy to your buffer
    outFrame.data.assign(pData, pData + cbCurrentLength);
    outFrame.timestamp = llTimestamp;
    outFrame.format = MFVideoFormat_MJPG; // or detect from media type

    pBuffer->Unlock();
    pBuffer->Release();
    pSample->Release();
}
```

> ✅ This ensures you **always get the latest frame**, and **discard any queued frames** — simulating `drop=true` in GStreamer.

✅ STEP 4: RUN IN DEDICATED THREAD PER CAMERA — WITH FLUSH

```
void CameraThread(int cam_id, IMFSourceReader* pReader) {
    while (!g_shutdown) {
        CameraFrame frame;
        ReadLatestFrameOnly(pReader, frame); // ← Flush + block until new frame

        if (!frame.data.empty()) {
            std::lock_guard<std::mutex> lock(g_frame_mutex);
            g_latest_frames[cam_id] = frame; // Overwrite previous — no queue!
            g_frame_ready[cam_id] = true;
        }
    }
}
```

🔄 This creates a **"latest-frame-only"** system — no buffering, no backlog.

✅ STEP 5 (Optional): SET USB CAMERA TO "SYNC START" MODE

Some UVC cameras support "**Sync Start**" — where exposure begins at the same USB SOF (Start of Frame).

You can try enabling it via:

```
// After setting media type, try to set this:
IMFMediaType* pCurrentType = nullptr;
pReader->GetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, &pCurrentType);

// Try to set sync-start if supported
// (Not all cameras support this — check with USBView or device specs)
pCurrentType->SetUINT32(MF_MT_UVC_SYNC_START, 1); // Experimental

pReader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr, pCurrentType);
pCurrentType->Release();
```

🔍 Use **USBView** or **UVC Extension Property** tools to check if your camera supports hardware-level sync.

✅ STEP 6: DISABLE WINDOWS CAMERA FRAME SERVER (CRITICAL!)

Windows 10/11 routes some cameras through a "Frame Server" — adds 1–2 frame delay.

How to disable:

1. Press `Ctrl+Shift+Esc` → Task Manager → Startup tab
2. Find "**Windows Camera Frame Server**" → Right-click → Disable
3. Reboot

> ⚠️ This may break Windows Camera app — acceptable for dedicated capture rigs.

✅ STEP 7: TUNE USB DRIVER LATENCY (REGISTRY HACK)

Some USB host controllers buffer to "smooth" delivery. Force minimal buffering:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{36fc9e60-c465-11cf-8056-444553540000}\XXXX]
"LatencyTimer"=dword:00000001
```

> 🛑 **WARNING**: Only do this if you understand USB driver risks. Test on non-critical machine.

Use **USBTreeView** to find your camera's Host Controller instance ID.

📊 COMPARISON: MF "No Buffer" vs GStreamer

Feature	MF (with Flush)	GStreamer
True "no buffer"	❌ Simulated via Flush	✅ Native (`drop=true`)
Deterministic latency	⚠️ Driver-dependent	✅ Fully scriptable
Timestamp accuracy	✅ `IMFSample` timestamp	✅ `GST_BUFFER_PTS`
Zero-copy to GPU	✅ `IMFDXGIBuffer`	✅ `d3d11` caps
Ease of tuning	❌ COM APIs, complex	✅ Pipeline text
Max camera count	~50 per machine (USB BW)	Same

> ✅ **MF + Flush is usable** — but **GStreamer is simpler and more deterministic** for real-time sync.

🚨 LIMITATIONS OF MF "NO BUFFER"

- `Flush()` adds small overhead (~0.1–1ms).
- USB driver may still buffer 1 frame internally.

- No guarantee frames from all 100 cams are captured at *exactly* same time — unless cameras support **hardware trigger** or **UVC Sync Start**.
- MF Source Reader not designed for hard real-time systems.

✅ RECOMMENDED FINAL MF PIPELINE FOR SYNC

```
// 1. Create reader with IGNORE_CLOCK
// 2. Set MJPEG/YUY2 media type
// 3. In thread loop:
//      → pReader->Flush(stream)
//      → pReader->ReadSample(...) // blocks until new frame
//      → Extract sample timestamp + data
//      → Push to global "latest frame" array (overwrite old)
// 4. Sync engine waits until all 100 “latest” frames have timestamps within ±1 frame
// 5. Display/process together
```

🧪 BONUS: MEASURE LATENCY

Add high-res timer to see actual capture→process delay:

```
LARGE_INTEGER freq, start, end;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start);

ReadLatestFrameOnly(pReader, frame);

QueryPerformanceCounter(&end);
double latency_ms = (double)(end.QuadPart - start.QuadPart) * 1000.0 / freq.QuadPart;
```

🏁 CONCLUSION

> ✅ **Yes, you can simulate “no buffer” in Media Foundation** by:
>
> - Using `MF_SOURCE_READER_IGNORE_CLOCK`
> - Calling `Flush()` before every `ReadSample()`
> - Overwriting “latest frame only” in a global array
> - Disabling Windows Camera Frame Server
> - Tuning USB driver latency