Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Интерфейсы и устройства вычислительных машин

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

на тему

Веб-камера

Выполнил
студент гр. 250541 В.Ю. Бобрик

Проверил
ст. преподаватель каф. ЭВМ Д.В. Куприянова

Минск 2025

СОДЕРЖАНИЕ

1 Цель работы

Написать программу, осуществляющую захват изображения с последующим сохранением в файл.

2 Исходные данные к работе

Для написания программы используется язык C/C++ и операционная система Windows 10.

Необходимо вывести информацию об установленной веб-камере. Осуществить захват изображения (фото и видео) с последующим сохранением в файл. Предусмотреть скрытый вариант фотонаблюдения, когда на мониторе и на панели задач не отображается информация о Вашем работающем приложении;

3 Теоретические сведения

Веб-камера - маленькая цифровая модель, встроенная в ноутбук или ПК, передающая изображения в режиме прямого эфира.

Назначение веб-камеры: захватывать свет через маленькую линзу спереди, используя сетку из детекторов света микроскопических размеров.

Датчики встроены в сам микрочип, который отвечает за восприятие картинки.

Чип выполняет роль ядра, снимает видео и фото, преобразуя их в цифровой формат. Этот формат компьютер должен распознавать в режиме реального времени.

Принцип, как работает веб-камера на компьютере, будет единым для всех типов устройств. Памяти у прибора нет, поскольку видео, фото не сохраняются, а сразу передаются на ПК.

Сенсор изображения является важнейшим элементом любой видеокамеры. Сегодня практически во всех камерах используются матрицы изображения CCD или CMOS. Обе матрицы выполняют задачу преобразования изображения, построенного на сенсоре объективом, в электрический сигнал.

CCD является аналоговой матрицей, несмотря на дискретность светочувствительной структуры. Когда свет попадает на матрицу, в каждом пикселе накапливается заряд или пакет электронов, преобразуемый при считывании на нагрузке в напряжение видеосигнала, пропорциональное освещенности пикселей. Минимальное количество промежуточных переходов этого заряда и отсутствие активных устройств обеспечивают высокую идентичность чувствительных элементов CCD.

CMOS-матрица является цифровым устройством с активными чувствительными элементами (Active Pixel Sensor). С каждым пикселем работает свой усилитель, преобразующий заряд чувствительного элемента в

напряжение. Это дает возможность практически индивидуально управлять каждым пикселем.

К преимуществам CCD матриц относятся: низкий уровень шумов; высокий коэффициент заполнения пикселов (около 100%); высокая эффективность (отношение числа зарегистрированных фотонов к их общему числу, попавшему на светочувствительную область матрицы, для CCD - 95%); высокий динамический диапазон (чувствительность).

К недостаткам CCD матриц относятся: сложный принцип считывания сигнала, а следовательно и технология; высокий уровень энергопотребления (до 2-5Вт); дороже в производстве в сравнении с CMOS.

К преимуществам CMOS матриц относятся: высокое быстродействие (до 500 кадров/с); низкое энергопотребление (почти в 100 раз по сравнению с CCD); дешевле и проще в производстве; перспективность технологии.

К недостаткам CMOS матриц относятся: низкий коэффициент заполнения пикселов, что снижает чувствительность (эффективная поверхность пиксела ~75%, остальное занимают транзисторы); высокий уровень шума (он обусловлен так называемыми темповыми токами - даже в отсутствие освещения через фотодиод течет довольно значительный ток), борьба с которым усложняет и удорожает технологию; невысокий динамический диапазон.

4 Выполнение работы

Исходный код программы, выполняющей поставленную задачу, приведен ниже.

CommandLine.h

```
000 #pragma once
001 #include <string>
002 #include <optional>
003
004 struct CmdOptions {
005     bool info = false;
006     bool snap = false;
007     bool capture = false;
008     bool quiet = false;
009     bool verbose = false;
010     std::optional<std::wstring> outputPath;
011     std::optional<int> deviceId;
012     int captureSeconds = 0;
013 };
014
015 class CommandLineParser {
016 public:
017     static std::optional<CmdOptions> Parse(int argc, wchar_t** argv, std::wstring& err);
018 };
```

CommandLine.cpp

```
000 #include "CommandLine.h" // объявление CmdOptions и парсера
001 #include <string>        // std::wstring
002 #include <algorithm>     // std::transform
003
004 // Возвращает нижний регистр строки
005 static std::wstring toLower(const std::wstring& s) {
006     std::wstring r = s; // копируем вход
007     std::transform(r.begin(), r.end(), r.begin(), ::towlower);
008     return r; // возвращаем результат
```

```
009 }
010
011 // Разбор аргументов командной строки
012 std::optional<CmdOptions> CommandLineParser::Parse(int argc, wchar_t** argv, std::wstring&
err) {
013     CmdOptions opt; // структура с результатами парсинга
014     for (int i = 1; i < argc; ++i) { // итерируем аргументы
015         std::wstring a = toLower(argv[i]);
016         if (a == L"--info") {
017             opt.info = true; // режим перечисления устройств
018         }
019         else if (a == L"--snap") {
020             opt.snap = true; // режим снятия одного кадра
021         }
022         else if (a == L"--capture") {
023             opt.capture = true; // режим записи видео
024             if (i + 1 >= argc) { // ожидаем параметр (секунды)
025                 err = L"Неверный вызов: --capture требует аргумент (секунды)";
026                 return std::nullopt; // ошибочный вызов — возвращаем nullopt
027             }
028             opt.captureSeconds = _wtoi(argv[++i]); // парсим целое число
029             if (opt.captureSeconds <= 0) { // проверяем положительность
030                 err = L"Неверный аргумент для --capture: ожидается положительное число";
031                 return std::nullopt; // неверный аргумент — ошибка
032             }
033         }
034         else if (a == L"--output") {
035             if (i + 1 >= argc) { // ожидаем путь после --output
036                 err = L"Неверный вызов: --output требует аргумент (путь)";
037                 return std::nullopt;
038             }
039             opt.outputPath = argv[++i]; // сохраняем строку пути
040         }
041         else if (a == L"--verbose") {
042             opt.verbose = true; // включаем подробный вывод
043         }
044         else if (a == L"--quiet") {
045             opt.quiet = true; // не вызываем консоль
046         }
047         else {
048             err = L"Неподдерживаемый аргумент: " + std::wstring(argv[i]);
049             return std::nullopt; // ошибка парсинга
050         }
051     }
052
053     int modeCount = (int)opt.info + (int)opt.snap + (int)opt.capture;
054     if (modeCount == 0) { // ни один режим не выбран
055         err = L"Не указан режим работы: --info, --snap или --capture";
056         return std::nullopt;
057     }
058     if (modeCount > 1) { // конфликтующие режимы одновременно
059         err = L"Укажите только один режим: --info или --snap или --capture";
060         return std::nullopt;
061     }
062     if (opt.quiet && opt.info) { // --quiet конфликтует с --info
063         err = L"--quiet несовместим с --info";
064         return std::nullopt;
065     }
066     return opt; // возвращаем опции
067 }
```

DeviceEnumerator.h

```
000 #pragma once
001 #include "MFHelpers.h"
002 #include <vector>
003
004 class DeviceEnumerator {
005 public:
006     DeviceEnumerator();
007     ~DeviceEnumerator();
008     std::vector<DeviceInfo> ListDevices();
009 private:
010     bool initialized_ = false;
011 };
```

DeviceEnumerator.cpp

```
000 #include "DeviceEnumerator.h"
001 #include "Logger.h"
002 #include <mfapi.h>      // Media Foundation
003 #include <mfidl.h>      // MF интерфейсы
004 #include <mfreadwrite.h> // SourceReader/SinkWriter
005
006 DeviceEnumerator::DeviceEnumerator() {}
007
008 DeviceEnumerator::~DeviceEnumerator() {}
009
010 // Возвращает список доступных видеоустройств
011 std::vector<DeviceInfo> DeviceEnumerator::ListDevices() {
012     Logger::Instance().Verbose(L"Enumerating devices...");
013     auto devices = EnumerateDevices();
// вызываем перечисление MF устройств в MFHelpers
014     Logger::Instance().Verbose(std::wstring(L"Devices found: ") +
std::to_wstring(devices.size())); // сколько найдено (verbose)
015     return devices; // возвращаем вектор DeviceInfo
016 }
```

FrameGrabber.h

```
000 #pragma once
001
002 #ifndef _WIN32_WINNT
003 #define _WIN32_WINNT 0x0A00
004 #endif
005
006 #include <string>
007 #include "MFHelpers.h"
008
009 class FrameGrabber {
010 public:
011     FrameGrabber(int deviceIndex);
012     ~FrameGrabber();
013     HRESULT CaptureToJpeg(const std::wstring& outPath, UINT quality = 95,
014                           std::wstring* usedDeviceName = nullptr,
015                           VideoFormatInfo* usedFmt = nullptr);
016 private:
017     int deviceIndex_;
018 };
```

FrameGrabber.cpp

```
000 // FrameGrabber.cpp
001 #ifndef _WIN32_WINNT
002 #define _WIN32_WINNT 0x0A00               // требуем Windows 10 API
003 #endif
004
005 #include "FrameGrabber.h"
006 #include "Logger.h"
007 #include "ScopeGuard.h"
008
009 #include <windows.h>                       // базовый WinAPI
010 #include <objbase.h>                       // CoCreateInstance и т.п.
011
012 #include <mfapi.h>                         // Media Foundation
013 #include <mfidl.h>
014 #include <mfobjects.h>
015 #include <mfreadwrite.h>
016 #include <mftransform.h>
017 #include <mferror.h>
018
019 #include <wrl/client.h>                    // ComPtr
020 #include <wincodec.h>                      // WIC для записи JPEG
021 #include <atlbase.h>                       // вспомогательно (COM)
022 #include <vector>                          // std::vector
023 #include <sstream>                         // string streams
024 #include <memory>                          // shared_ptr, unique_ptr
025
026 #pragma comment(lib, "mfplat.lib")         // линковка MF и WIC
027 #pragma comment(lib, "mf.lib")
028 #pragma comment(lib, "mfreadwrite.lib")
029 #pragma comment(lib, "mfuuid.lib")
030 #pragma comment(lib, "windowscodecs.lib")
031 #pragma comment(lib, "shlwapi.lib")
032
033 using Microsoft::WRL::ComPtr;             // умный указатель из COM
```

```
034
035 // Генерирация имени файла
036 static std::wstring MakeTimestampFilename(const std::wstring& ext, const std::wstring&
outputDir) {
037     SYSTEMTIME st;
038     GetLocalTime(&st);                      // текущее локальное время
039     wchar_t buf[128];
040     swprintf_s(buf, L"%04d-%02d-%02d_%02d-%02d-%02d%s",
041         st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond, ext.c_str()); //
формат имени
042     std::wstring filename = buf;            // имя файла
043     std::wstring out = outputDir;           // копируем директорию
044     if (!out.empty()) {
045         if (out.back() != L'\\' && out.back() != L'/') out += L"\\"; // добавляем слэш если
нужно
046     }
047     out += filename;                        // итоговый путь
048     return out;
049 }
050
051 FrameGrabber::FrameGrabber(int deviceIndex) : deviceIndex_(deviceIndex) {} // сохраняем
индекс устройства
052 FrameGrabber::~FrameGrabber() {}               // деструктор ничего не делает
053
054 // Захват одного кадра и сохранение в JPEG
055 HRESULT FrameGrabber::CaptureToJpeg(const std::wstring& outPath, UINT quality, std::wstring*
usedDeviceName, VideoFormatInfo* usedFmt) {
056     Logger::Instance().Verbose(L"Starting capture to JPEG");
057
058     IMFAttributes* pAttr = nullptr;
059     HRESULT hr = MFCreateAttributes(&pAttr, 1); // создаём атрибуты для перечисления
устройств
060     if (FAILED(hr)) { Logger::Instance().Error(L"MFCreateAttributes failed: " +
std::to_wstring((long)hr)); return hr; }
061     ScopeGuard gAttr([&] { if (pAttr) pAttr->Release(); }); // гарантированное освобождение
атрибутов
062
063     hr = pAttr->SetGUID(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID); // запрос на получение видеокамер
064     if (FAILED(hr)) { Logger::Instance().Error(L"SetGUID failed: " +
std::to_wstring((long)hr)); return hr; }
065
066     IMFActivate** ppDevices = nullptr;
067     UINT32 count = 0;
068     hr = MFEnumDeviceSources(pAttr, &ppDevices, &count); // перечисляем устройства
069     if (FAILED(hr)) { Logger::Instance().Error(L"MFEnumDeviceSources failed: " +
std::to_wstring((long)hr)); return hr; }
070     if (count == 0) { CoTaskMemFree(ppDevices); Logger::Instance().Error(L"No devices
found"); return E_FAIL; } // нет камер
071     if (deviceIndex_ < 0 || deviceIndex_ >= static_cast<int>(count)) {
CoTaskMemFree(ppDevices); Logger::Instance().Error(L"Invalid device index"); return E_INVALIDARG;
} // неверный индекс
072
073     IMFActivate* act = ppDevices[deviceIndex_]; // выбираем активацию нужного устройства
074     ScopeGuard gAct([&] { if (act) act->Release(); CoTaskMemFree(ppDevices); }); //
освобождение массива и активации
075
076     WCHAR* friendly = nullptr;
077     hr = act->GetAllocatedString(MF_DEVSOURCE_ATTRIBUTE_FRIENDLY_NAME, &friendly, nullptr);
// получаем имя
078     if (SUCCEEDED(hr) && usedDeviceName) *usedDeviceName = friendly; // возвращаем имя
079     if (friendly) CoTaskMemFree(friendly);    // освобождаем строку
080
081     ComPtr<IMFMediaSource> spSource;
082     hr = act->ActivateObject(IID_PPV_ARGS(&spSource)); // активируем источник (камера)
083     if (FAILED(hr)) { Logger::Instance().Error(L"ActivateObject failed: " +
std::to_wstring((long)hr)); return hr; }
084
085     ComPtr<IMFSourceReader> spReader;
086     hr = MFCreateSourceReaderFromMediaSource(spSource.Get(), nullptr, &spReader); // создаём
SourceReader
087     if (FAILED(hr)) { Logger::Instance().Error(L"MFCreateSourceReaderFromMediaSource failed:
" + std::to_wstring((long)hr)); return hr; }
088
089     ComPtr<IMFMediaType> pTypeOut;
090     hr = MFCreateMediaType(&pTypeOut);         // медиатип для запроса формата
```

```
091     if (FAILED(hr)) { Logger::Instance().Error(L"MFCreateMediaType failed: " +
std::to_wstring((long)hr)); return hr; }
092     hr = pTypeOut->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video); // ставим major type = video
093     if (FAILED(hr)) { Logger::Instance().Error(L"SetGUID major type failed: " +
std::to_wstring((long)hr)); return hr; }
094
095     bool chosenRGB32 = false;
096     bool chosenRGB24 = false;
097     bool chosenNV12 = false;
098
099     // Пробуем получить RGB32 напрямую из SourceReader
100     hr = pTypeOut->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_RGB32);
101     if (SUCCEEDED(hr)) {
102         hr = spReader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pTypeOut.Get()); // применяем запрос
103         if (SUCCEEDED(hr)) {
104             chosenRGB32 = true;
105             Logger::Instance().Verbose(L"Using RGB32 output from SourceReader");
106         }
107         else {
108             Logger::Instance().Verbose(L"RGB32 not supported");
109         }
110     }
111
112     // Если RGB32 не доступен — пробуем RGB24
113     if (!chosenRGB32) {
114         hr = pTypeOut->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_RGB24);
115         if (SUCCEEDED(hr)) {
116             hr = spReader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pTypeOut.Get());
117             if (SUCCEEDED(hr)) {
118                 chosenRGB24 = true;
119                 Logger::Instance().Verbose(L"Using RGB24 output from SourceReader");
120             }
121             else {
122                 Logger::Instance().Verbose(L"RGB24 not supported");
123             }
124         }
125     }
126
127     // Если ни RGB32 ни RGB24 не прошли — пробуем NV12 и делаем софт-конвертацию
128     if (!chosenRGB32 && !chosenRGB24) {
129         hr = pTypeOut->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_NV12);
130         if (SUCCEEDED(hr)) {
131             hr = spReader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pTypeOut.Get());
132             if (SUCCEEDED(hr)) {
133                 chosenNV12 = true;
134                 Logger::Instance().Verbose(L"Using NV12 fallback from SourceReader");
135             }
136             else {
137                 Logger::Instance().Error(L"NV12 fallback not supported");
138                 return hr;
139             }
140         }
141         else {
142             Logger::Instance().Error(L"Failed to set NV12 subtype");
143             return hr;
144         }
145
146     }
147
148     ComPtr<IMFMediaType> pFinalType;
149     hr = spReader->GetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, &pFinalType); //
получаем фактический тип
150     if (FAILED(hr)) {
151         Logger::Instance().Error(L"GetCurrentMediaType failed: " +
std::to_wstring((long)hr));
152         return hr;
153     }
154
155     VideoFormatInfo vf{};
156     ParseMediaType(pFinalType.Get(), vf);       // парсим ширину/высоту/fps/подтип
157     if (usedFmt) *usedFmt = vf;                 // возвращаем формат при запросе
158     Logger::Instance().Verbose(L"Selected format: " + std::to_wstring(vf.width) + L"x" +
std::to_wstring(vf.height));
```

```
159
160     hr = spReader->SetStreamSelection(MF_SOURCE_READER_FIRST_VIDEO_STREAM, TRUE); // включаем
поток чтения
161     if (FAILED(hr)) {
162         Logger::Instance().Error(L"SetStreamSelection failed: " + std::to_wstring((long)hr));
163         return hr;
164     }
165
166     const DWORD kTimeoutMs = 5000;              // таймаут ожидания кадра
167     const DWORD kPollIntervalMs = 30;           // интервал опроса
168     DWORD waited = 0;
169     DWORD streamIndex = 0, flags = 0;
170     LONGLONG llTimeStamp = 0;
171     ComPtr<IMFSample> spSample;
172
173     // Цикл чтения одного кадра с таймаутом
174     while (waited < kTimeoutMs) {
175         flags = 0;
176         spSample.Reset();
177         hr = spReader->ReadSample(MF_SOURCE_READER_FIRST_VIDEO_STREAM, 0, &streamIndex,
&flags, &llTimeStamp, &spSample);
178         if (FAILED(hr)) {
179             Logger::Instance().Error(L"ReadSample failed: " + std::to_wstring((long)hr));
180             break;
181         }
182         if (spSample) break;
183         if (flags & MF_SOURCE_READERF_ENDOFSTREAM) {
184             Logger::Instance().Error(L"ReadSample signalled EOS");
185             break;
186         }
187         Sleep(kPollIntervalMs);
188         waited += kPollIntervalMs;
189     }
190
191     if (!spSample) {
192         Logger::Instance().Error(L"No sample received after waiting " +
std::to_wstring(waited) + L" ms");
193         return E_FAIL;
194     }
195
196     ComPtr<IMFMediaBuffer> spBuffer;
197     hr = spSample->ConvertToContiguousBuffer(&spBuffer);
198     if (FAILED(hr)) {
199         Logger::Instance().Error(L"ConvertToContiguousBuffer failed: " +
std::to_wstring((long)hr));
200         return hr;
201     }
202
203     BYTE* pData = nullptr; DWORD maxLen = 0, curLen = 0;
204     hr = spBuffer->Lock(&pData, &maxLen, &curLen);
205     if (FAILED(hr)) {
206         Logger::Instance().Error(L"Buffer Lock failed: " + std::to_wstring((long)hr));
207         return hr;
208     }
209     ScopeGuard gUnlock([&] { if (spBuffer) spBuffer->Unlock(); });
210
211     if (vf.width == 0 || vf.height == 0) {
212         Logger::Instance().Error(L"Invalid frame dimensions");
213         return E_FAIL;
214     }
215
216     ComPtr<IWICImagingFactory> spWIC;
217     hr = CoCreateInstance(CLSID_WICImagingFactory, nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&spWIC));
218     if (FAILED(hr)) {
219         Logger::Instance().Error(L"WIC CreateInstance failed: " + std::to_wstring((long)hr));
220         return hr;
221     }
222
223     GUID finalSub = { 0 };
224     pFinalType->GetGUID(MF_MT_SUBTYPE, &finalSub);
225     bool useRgb32 = (finalSub == MFVideoFormat_RGB32);
226     bool useRgb24 = (finalSub == MFVideoFormat_RGB24);
227     bool useNv12 = (finalSub == MFVideoFormat_NV12);
228
229     UINT bpp = useRgb32 ? 4u : 3u;
```

```
230     UINT expectedStride = vf.width * bpp;
231     UINT expectedBytes = expectedStride * vf.height;
232 
233     Logger::Instance().Verbose(L"Frame buffer info: curLen=" + std::to_wstring(curLen));
234 
235     UINT useBytes = (curLen >= expectedBytes) ? expectedBytes : curLen;
236 
237     WICPixelFormatGUID pixfmt = useRgb32 ? GUID_WICPixelFormat32bppBGR :
GUID_WICPixelFormat24bppBGR;
238 
239     std::shared_ptr<std::vector<BYTE>> convPtr;
240     if (useNv12) {
241         Logger::Instance().Verbose(L"NV12 frame captured; converting to BGR24");
242 
243         UINT yPlaneSize = vf.width * vf.height;
244         UINT uvPlaneSize = yPlaneSize / 2;
245         if (curLen < static_cast<DWORD>(yPlaneSize + uvPlaneSize)) {
246             Logger::Instance().Error(L"NV12 buffer too small");
247             return E_FAIL;
248         }
249 
250         UINT bpp_conv = 3;
251         UINT stride_conv = vf.width * bpp_conv;
252         UINT bytes_conv = stride_conv * vf.height;
253         std::vector<BYTE> convBuf(bytes_conv);
254         if (convBuf.empty()) {
255             Logger::Instance().Error(L"Failed to allocate conversion buffer");
256             return E_OUTOFMEMORY;
257         }
258 
259         const BYTE* yPlane = pData;
260         const BYTE* uvPlane = pData + yPlaneSize;
261 
262         for (UINT row = 0; row < vf.height; ++row) {
263             const BYTE* yRow = yPlane + row * vf.width;
264             const BYTE* uvRow = uvPlane + (row / 2) * vf.width;
265             BYTE* outRow = convBuf.data() + row * stride_conv;
266             for (UINT col = 0; col < vf.width; ++col) {
267                 int Y = (int)yRow[col];
268                 int uvIndex = (col & ~1);
269                 int U = (int)uvRow[uvIndex + 0];
270                 int V = (int)uvRow[uvIndex + 1];
271 
272                 int C = Y - 16;
273                 int D = U - 128;
274                 int E = V - 128;
275 
276                 int R = (298 * C + 409 * E + 128) >> 8;
277                 int G = (298 * C - 100 * D - 208 * E + 128) >> 8;
278                 int B = (298 * C + 516 * D + 128) >> 8;
279 
280                 if (R < 0) R = 0; else if (R > 255) R = 255;
281                 if (G < 0) G = 0; else if (G > 255) G = 255;
282                 if (B < 0) B = 0; else if (B > 255) B = 255;
283 
284                 outRow[col * 3 + 0] = (BYTE)B;
285                 outRow[col * 3 + 1] = (BYTE)G;
286                 outRow[col * 3 + 2] = (BYTE)R;
287             }
288         }
289 
290         convPtr = std::make_shared<std::vector<BYTE>>(std::move(convBuf));
291         pData = convPtr->data();
292         curLen = static_cast<DWORD>(bytes_conv);
293         useBytes = bytes_conv;
294         useNv12 = false;
295         useRgb24 = true;
296         useRgb32 = false;
297         expectedStride = stride_conv;
298         expectedBytes = bytes_conv;
299         pixfmt = GUID_WICPixelFormat24bppBGR;
300 
301         ScopeGuard holdConv([convPtr]() {});
302         Logger::Instance().Verbose(L"Conversion done");
303     }
304     ComPtr<IWICBitmap> spBitmap;
```

```
305     bool createdFromMemory = false;
306     if (!useNv12) {
307         hr = spWIC->CreateBitmapFromMemory(vf.width, vf.height, pixfmt, expectedStride,
useBytes, pData, &spBitmap);
308         if (SUCCEEDED(hr)) {
309             createdFromMemory = true;
310             Logger::Instance().Verbose(L"CreateBitmapFromMemory succeeded");
311         }
312         else {
313             Logger::Instance().Verbose(L"CreateBitmapFromMemory failed, fallback will be
used");
314         }
315     }
316
317     if (!createdFromMemory) {
318         hr = spWIC->CreateBitmap(vf.width, vf.height, pixfmt, WICBitmapCacheOnLoad,
&spBitmap);
319         if (FAILED(hr)) {
320             Logger::Instance().Error(L"Fallback CreateBitmap failed");
321             return hr;
322         }
323         ComPtr<IWICBitmapLock> lock;
324         WICRect rect = { 0,0,(INT)vf.width,(INT)vf.height };
325         hr = spBitmap->Lock(&rect, WICBitmapLockWrite, &lock);
326         if (SUCCEEDED(hr)) {
327             UINT cbBufferSize = 0;
328             BYTE* pv = nullptr;
329             hr = lock->GetDataPointer(&cbBufferSize, &pv);
330             if (SUCCEEDED(hr)) {
331                 UINT toCopy = min(cbBufferSize, useBytes);
332                 memcpy(pv, pData, toCopy);
333                 if (cbBufferSize > toCopy) memset(pv + toCopy, 0, cbBufferSize - toCopy);
334                 lock->Release();
335                 Logger::Instance().Verbose(L"Copied pixel data into WIC bitmap");
336             }
337             else {
338                 Logger::Instance().Error(L"BitmapLock GetDataPointer failed");
339                 lock->Release();
340                 return hr;
341             }
342         }
343         else {
344             Logger::Instance().Error(L"Bitmap Lock failed");
345             return hr;
346         }
347     }
348
349     ComPtr<IWICStream> spStream;
350     hr = spWIC->CreateStream(&spStream);
351     if (FAILED(hr)) { Logger::Instance().Error(L"CreateStream failed"); return hr; }
352
353     hr = spStream->InitializeFromFilename(outPath.c_str(), GENERIC_WRITE);
354     if (FAILED(hr)) { Logger::Instance().Error(L"InitializeFromFilename failed"); return hr;
}
355
356     ComPtr<IWICBitmapEncoder> spEncoder;
357     hr = spWIC->CreateEncoder(GUID_ContainerFormatJpeg, nullptr, &spEncoder);
358     if (FAILED(hr)) { Logger::Instance().Error(L"CreateEncoder failed"); return hr; }
359
360     hr = spEncoder->Initialize(spStream.Get(), WICBitmapEncoderNoCache);
361     if (FAILED(hr)) { Logger::Instance().Error(L"Encoder Initialize failed"); return hr; }
362
363     ComPtr<IWICBitmapFrameEncode> spFrame;
364     hr = spEncoder->CreateNewFrame(&spFrame, nullptr);
365     if (FAILED(hr)) { Logger::Instance().Error(L"CreateNewFrame failed"); return hr; }
366
367     hr = spFrame->Initialize(nullptr);
368     if (FAILED(hr)) { Logger::Instance().Error(L"Frame Initialize failed"); return hr; }
369
370     hr = spFrame->SetSize(vf.width, vf.height);
371     if (FAILED(hr)) { Logger::Instance().Error(L"SetSize failed"); return hr; }
372
373     WICPixelFormatGUID targetFmt = pixfmt;
374     hr = spFrame->SetPixelFormat(&targetFmt);
375     if (FAILED(hr)) { Logger::Instance().Error(L"SetPixelFormat failed"); return hr; }
376
```

```
377     bool needConversion = (targetFmt != pixfmt);
378     ComPtr<IWICBitmap> spBitmapToWrite = spBitmap;
379     if (needConversion) {
380         Logger::Instance().Verbose(L"Pixel format conversion required by encoder");
381         ComPtr<IWICFormatConverter> spConv;
382         hr = spWIC->CreateFormatConverter(&spConv);
383         if (FAILED(hr)) { Logger::Instance().Error(L"CreateFormatConverter failed"); return
hr; }
384         hr = spConv->Initialize(spBitmap.Get(), targetFmt, WICBitmapDitherTypeNone, nullptr,
0.0, WICBitmapPaletteTypeCustom);
385         if (FAILED(hr)) { Logger::Instance().Error(L"FormatConverter Initialize failed");
return hr; }
386         ComPtr<IWICBitmap> spConvBitmap;
387         hr = spWIC->CreateBitmapFromSource(spConv.Get(), WICBitmapCacheOnLoad,
&spConvBitmap);
388         if (FAILED(hr)) { Logger::Instance().Error(L"CreateBitmapFromSource failed"); return
hr; }
389         spBitmapToWrite = spConvBitmap;
390     }
391
392     hr = spFrame->WriteSource(spBitmapToWrite.Get(), nullptr);
393     if (FAILED(hr)) {
394         Logger::Instance().Verbose(L"WriteSource failed, trying WritePixels fallback");
395         ComPtr<IWICBitmapLock> lock;
396         WICRect rect = { 0,0,(INT)vf.width,(INT)vf.height };
397         hr = spBitmapToWrite->Lock(&rect, WICBitmapLockRead, &lock);
398         if (FAILED(hr)) { Logger::Instance().Error(L"Bitmap Lock for Read failed"); return
hr; }
399         UINT cbBufferSize = 0;
400         BYTE* pv = nullptr;
401         hr = lock->GetDataPointer(&cbBufferSize, &pv);
402         if (FAILED(hr)) { Logger::Instance().Error(L"GetDataPointer failed"); lock-
>Release(); return hr; }
403         UINT rowStride = expectedStride;
404         UINT totalToWrite = min(cbBufferSize, useBytes);
405         hr = spFrame->WritePixels(vf.height, rowStride, totalToWrite, pv);
406         lock->Release();
407         if (FAILED(hr)) { Logger::Instance().Error(L"WritePixels failed"); return hr; }
408     }
409
410     hr = spFrame->Commit();
411     if (FAILED(hr)) { Logger::Instance().Error(L"Frame Commit failed"); return hr; }
412
413     hr = spEncoder->Commit();
414     if (FAILED(hr)) { Logger::Instance().Error(L"Encoder Commit failed"); return hr; }
415
416     spFrame.Reset();
417     spEncoder.Reset();
418     spStream.Reset();
419     spBitmap.Reset();
420     spBitmapToWrite.Reset();
421
422     WIN32_FILE_ATTRIBUTE_DATA fad;
423     if (GetFileAttributesExW(outPath.c_str(), GetFileExInfoStandard, &fad)) {
424         Logger::Instance().Verbose(L"Saved image: " + outPath);
425         return S_OK;
426     }
427     else {
428         DWORD gle = GetLastError();
429         Logger::Instance().Error(L"Failed to write file: " + outPath);
430         return HRESULT_FROM_WIN32(gle ? gle : ERROR_FILE_NOT_FOUND);
431     }
432 }
```

Logger.h

```
000 #pragma once
001 #include <string>
002
003 class Logger {
004 public:
005     static Logger& Instance();
006
007     void InitConsole(bool enableConsole, bool verbose);
008
009     void Info(const std::wstring& msg);
010     void Warn(const std::wstring& msg);
```

```
011     void Error(const std::wstring& msg);
012     void Verbose(const std::wstring& msg);
013 private:
014     Logger();
015     ~Logger();
016     Logger(const Logger&) = delete;
017     Logger& operator=(const Logger&) = delete;
018
019     struct Impl;
020     Impl* pImpl;
021 };
```

Logger.cpp

```
000 #include "Logger.h"
001 #include <mutex>                    // std::mutex, lock_guard
002 #include <windows.h>                // WinAPI (Console, WriteConsole, etc.)
003 #include <iostream>
004
005 // Внутренняя реализация логгера — хранит состояние и мьютекс
006 struct Logger::Impl {
007     std::mutex mtx;                 // защищает одновременный вывод из потоков
008     bool console = true;            // разрешён вывод в консоль
009     bool verbose = false;           // флаг подробного вывода
010     Impl() = default;               // конструктор по умолчанию
011 };
012
013 // Возвращает текущий таймстамп в виде строки "YYYY-MM-DD HH:MM:SS"
014 static std::wstring TimestampNow() {
015     SYSTEMTIME st;
016     GetLocalTime(&st);              // получаем локальное время
017     wchar_t buf[64];
018     swprintf_s(buf, sizeof(buf) / sizeof(buf[0]), L"%04d-%02d-%02d %02d:%02d:%02d",
019         st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond); // форматируем
строку
020     return std::wstring(buf);
021 }
022
023 // Пытается безопасно записать wide-строку в консоль/файл/пайп
024 static void SafeWriteConsoleWide(HANDLE h, const std::wstring& s) {
025     if (!h || h == INVALID_HANDLE_VALUE) return; // ничего не делаем при невалидном
дескрипторе
026     DWORD written = 0;
027     if (WriteConsoleW(h, s.c_str(), (DWORD)s.size(), &written, nullptr)) return; // если
WriteConsoleW сработал — готово
028
029     // Если WriteConsoleW не поддерживается (например, дескриптор не консоль) — конвертим в
UTF-8 и пишем через WriteFile
030     int needed = WideCharToMultiByte(CP_UTF8, 0, s.c_str(), (int)s.size(), nullptr, 0,
nullptr, nullptr); // размер буфера в байтах
031     if (needed <= 0) return;           // не удалось посчитать размер
032     std::string buf(needed, '\0');     // выделяем строку нужного размера
033     WideCharToMultiByte(CP_UTF8, 0, s.c_str(), (int)s.size(), &buf[0], needed, nullptr,
nullptr); // конвертация в UTF-8
034     DWORD out = 0;
035     WriteFile(h, buf.data(), (DWORD)buf.size(), &out, nullptr); // пишем байты в дескриптор
036 }
037
038 // Singleton: возвращает единственный экземпляр Logger
039 Logger& Logger::Instance() {
040     static Logger inst;               // статический единственный объект
041     return inst;                      // возвращаем ссылку
042 }
043
044 Logger::Logger() : pImpl(new Impl()) {} // конструктор: создаём Impl
045 Logger::~Logger() { delete pImpl; }     // деструктор: удаляем Impl
046
047 // Инициализация поведения логгера
048 void Logger::InitConsole(bool enableConsole, bool verbose) {
049     if (!pImpl) pImpl = new Impl();    // на всякий случай создаём Impl если нет
050     std::lock_guard<std::mutex> g(pImpl->mtx); // блокируем настройки
051     pImpl->console = enableConsole;    // включаем/выключаем вывод в консоль
052     pImpl->verbose = verbose;          // включаем/выключаем verbose
053     if (pImpl->console) {              // если вывод в консоль разрешён
054         SetConsoleOutputCP(CP_UTF8);   // ставим кодировку вывода UTF-8
055         SetConsoleCP(CP_UTF8);         // ставим кодировку ввода UTF-8
056     }
```

```
057 }
058
059 // Информационное сообщение
060 void Logger::Info(const std::wstring& msg) {
061     if (!pImpl) return;                // защита от неинициализированного pImpl
062     std::lock_guard<std::mutex> g(pImpl->mtx); // синхронизуем доступ
063     if (!pImpl->console) return;       // если консоль отключена — не выводим
064     std::wstring line;
065     if (pImpl->verbose) line = TimestampNow() + L" [INFO] " + msg + L"\n"; // с таймстампом
при verbose
066     else line = msg + L"\n";           // иначе только сообщение
067     SafeWriteConsoleWide(GetStdHandle(STD_OUTPUT_HANDLE), line); // вывод в STDOUT
068 }
069
070 // Предупреждение
071 void Logger::Warn(const std::wstring& msg) {
072     if (!pImpl) return;
073     std::lock_guard<std::mutex> g(pImpl->mtx); // синхронизация
074     if (!pImpl->console) return;
075     std::wstring line = (pImpl->verbose ? (TimestampNow() + L" [WARN] " + msg + L"\n") :
(L"[WARN] " + msg + L"\n")); // формат
076     SafeWriteConsoleWide(GetStdHandle(STD_OUTPUT_HANDLE), line); // вывод в STDOUT
077 }
078
079 // Ошибка
080 void Logger::Error(const std::wstring& msg) {
081     if (!pImpl) return;
082     std::lock_guard<std::mutex> g(pImpl->mtx);
083     if (!pImpl->console) return;
084     std::wstring line = (pImpl->verbose ? (TimestampNow() + L" [ERROR] " + msg + L"\n") :
(L"[ERROR] " + msg + L"\n")); // формат
085     SafeWriteConsoleWide(GetStdHandle(STD_ERROR_HANDLE), line); // вывод в STDERR
086 }
087
088 // Подробный лог (выводится только при verbose)
089 void Logger::Verbose(const std::wstring& msg) {
090     if (!pImpl) return;
091     std::lock_guard<std::mutex> g(pImpl->mtx);
092     if (!pImpl->console) return;
093     if (!pImpl->verbose) return;       // пропускаем если verbose отключён
094     std::wstring line = TimestampNow() + L" [VERBOSE] " + msg + L"\n"; // формат с
таймстампом
095     SafeWriteConsoleWide(GetStdHandle(STD_OUTPUT_HANDLE), line); // вывод в STDOUT
096 }
```

MFHelpers.h

```
000 #pragma once
001
002 #include <windows.h>
003 #include <objbase.h>
004
005 #include <mfapi.h>
006 #include <mfidl.h>
007 #include <mfobjects.h>
008 #include <mfreadwrite.h>
009 #include <mftransform.h>
010 #include <mferror.h>
011
012 #include <shlwapi.h>
013 #include <wincodec.h>
014
015 #include <wrl/client.h>
016 #include <string>
017 #include <vector>
018
019 #include "ScopeGuard.h"
020
021 struct VideoFormatInfo {
022     UINT32 width{};
023     UINT32 height{};
024     UINT32 fpsNumerator{};
025     UINT32 fpsDenominator{};
026     GUID subtype{};
027     UINT32 bitDepth{};
028 };
029
```

```
030 struct DeviceInfo {
031     std::wstring id;
032     std::wstring name;
033     std::wstring vendor;
034     std::vector<VideoFormatInfo> formats;
035 };
036
037 std::vector<DeviceInfo> EnumerateDevices();
038 std::wstring GuidToString(const GUID& g);
039 void ParseMediaType(IMFMediaType* pType, VideoFormatInfo& out);
```

MFHelpers.cpp

```
000 // MFHelpers.cpp
001 #ifndef _WIN32_WINNT
002 #define _WIN32_WINNT 0x0A00                // требуем Windows 10 API
003 #endif
004
005 #include "MFHelpers.h"
006
007 #include <windows.h>                       // базовый WinAPI
008 #include <objbase.h>                       // COM
009 #include <mfapi.h>                         // Media Foundation
010 #include <mfidl.h>
011 #include <mfobjects.h>
012 #include <mfreadwrite.h>
013 #include <mftransform.h>
014 #include <mferror.h>
015
016 #include <wrl/client.h>                    // ComPtr
017 #include <shlwapi.h>                       // Path утилиты
018 #include <propvarutil.h>                   // PROPVARIANT helpers
019 #include <comdef.h>                        // _com_error
020 #include <sstream>                         // stringstream для форматирования
021 #include <iomanip>                         // манипуляторы вывода
022
023 #pragma comment(lib, "mfplat.lib")         // линковка необходимых библиотек MF и shlwapi
024 #pragma comment(lib, "mf.lib")
025 #pragma comment(lib, "mfreadwrite.lib")
026 #pragma comment(lib, "mfuuid.lib")
027 #pragma comment(lib, "shlwapi.lib")
028
029 using Microsoft::WRL::ComPtr;              // умный указатель COM
030
031 // Преобразует GUID в строку вида {XXXXXXXX-...}
032 std::wstring GuidToString(const GUID& g) {
033     wchar_t buf[64] = {};                  // буфер для строки GUID
034     if (0 == StringFromGUID2(g, buf, (int)std::size(buf))) { // форматируем GUID
035         return std::wstring(L"{?}");       // при ошибке возвращаем placeholder
036     }
037     return std::wstring(buf);              // возвращаем строковое представление
038 }
039
040 // Читает 64-битный или 32-битный целочисленный атрибут из IMFAttributes
041 static bool GetAttributeUINT64(IMFAttributes* attr, const GUID& key, UINT64& out) {
042     if (!attr) return false;               // защита от nullptr
043     PROPVARIANT var;
044     PropVariantInit(&var);                  // инициализация PROPVARIANT
045     HRESULT hr = attr->GetItem(key, &var);  // получаем значение по ключу
046     if (FAILED(hr)) return false;           // нет атрибута или ошибка
047     if (var.vt == VT_UI8) out = var.uhVal.QuadPart; // 64-битное значение
048     else if (var.vt == VT_UI4) out = var.ulVal;     // 32-битное значение
049     else { PropVariantClear(&var); return false; }  // неподдерживаемый тип
050     PropVariantClear(&var);                 // очищаем PROPVARIANT
051     return true;                            // успешно прочитано
052 }
053
054 // Разбирает IMFMediaType и заполняет VideoFormatInfo (ширина/высота/fps/subtype/bitDepth)
055 void ParseMediaType(IMFMediaType* pType, VideoFormatInfo& out) {
056     if (!pType) return;                    // защита от nullptr
057     UINT32 width = 0, height = 0;
058     if (SUCCEEDED(MFGetAttributeSize(pType, MF_MT_FRAME_SIZE, &width, &height))) {
059         out.width = width; out.height = height; // читаем размер кадра
060     }
061     else {
062         out.width = out.height = 0;          // нет информации о размере
063     }
```

```
064
065     UINT32 num = 0, den = 0;
066     if (SUCCEEDED(MFGetAttributeRatio(pType, MF_MT_FRAME_RATE, &num, &den))) {
067         out.fpsNumerator = num; out.fpsDenominator = den; // читаем FPS (num/den)
068     }
069     else {
070         out.fpsNumerator = out.fpsDenominator = 0; // нет информации о частоте
071     }
072
073     GUID subtype = { 0 };
074     if (SUCCEEDED(pType->GetGUID(MF_MT_SUBTYPE, &subtype))) {
075         out.subtype = subtype;                // читаем подтип (формат пикселей)
076     }
077     else {
078         out.subtype = GUID_NULL;              // неизвестный подтип
079     }
080
081 #ifdef MF_MT_BITS_PER_SAMPLE
082     UINT32 bitDepth = 0;
083     if (SUCCEEDED(pType->GetUINT32(MF_MT_BITS_PER_SAMPLE, &bitDepth))) out.bitDepth =
bitDepth; // битность, если доступна
084     else out.bitDepth = 0;
085 #else
086     out.bitDepth = 0;                         // если ключ не определён — 0
087 #endif
088 }
089
090 // Внутренняя реализация перечисления устройств — возвращает вектор DeviceInfo
091 static std::vector<DeviceInfo> EnumerateDevicesInternal() {
092     std::vector<DeviceInfo> list;             // результат
093
094     ComPtr<IMFAttributes> spAttr;
095     if (FAILED(MFCreateAttributes(&spAttr, 2))) return list; // создаём атрибуты
096
097     // Фильтрируем только видеоустройства (видеокамеры)
098     if (FAILED(spAttr->SetGUID(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID))) return list;
099
100     IMFActivate** ppDevices = nullptr;
101     UINT32 count = 0;
102     HRESULT hr = MFEnumDeviceSources(spAttr.Get(), &ppDevices, &count); // перечисляем
устройства
103     if (FAILED(hr) || count == 0) {
104         if (ppDevices) CoTaskMemFree(ppDevices); // освобождаем при необходимости
105         return list;                           // пустой список при ошибке или отсутствии
устройств
106     }
107
108     for (UINT32 i = 0; i < count; ++i) {
109         IMFActivate* act = ppDevices[i];      // берем IMFActivate для i-го устройства
110         DeviceInfo di;                        // структура для заполнения
111
112         WCHAR* friendlyName = nullptr;
113         if (SUCCEEDED(act->GetAllocatedString(MF_DEVSOURCE_ATTRIBUTE_FRIENDLY_NAME,
&friendlyName, nullptr))) {
114             di.name = friendlyName;           // читаем имя
115             CoTaskMemFree(friendlyName);      // освобождаем строку, выделенную MF
116         }
117
118         WCHAR* symId = nullptr;
119         if (SUCCEEDED(act-
>GetAllocatedString(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK, &symId, nullptr))) {
120             di.id = symId;                    // присваиваем символическую ссылку
(идентификатор)
121             CoTaskMemFree(symId);
122         }
123
124 #ifdef MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_HW_VENDOR_GUID
125         WCHAR* vendor = nullptr;
126         if (SUCCEEDED(act-
>GetAllocatedString(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_HW_VENDOR_GUID, &vendor, nullptr)))
{
127             di.vendor = vendor;               // читаем vendor GUID как строку (если
доступно)
128             CoTaskMemFree(vendor);
129         }
```

```
130 #else
131         (void)0;
132 #endif
133
134         // Опционально активируем источник чтобы прочитать нативные форматы
135         ComPtr<IMFMediaSource> spSource;
136         if (SUCCEEDED(act->ActivateObject(IID_PPV_ARGS(&spSource)))) {
137             ComPtr<IMFSourceReader> spReader;
138             if (SUCCEEDED(MFCreateSourceReaderFromMediaSource(spSource.Get(), nullptr,
&spReader))) {
139                 DWORD idx = 0;
140                 while (true) {
141                     ComPtr<IMFMediaType> spType;
142                     HRESULT hr2 = spReader-
>GetNativeMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, idx, &spType); // читаем нативный тип по
индексу
143                     if (FAILED(hr2)) break; // выход при исчерпании типов
144                     VideoFormatInfo vfi{};
145                     ParseMediaType(spType.Get(), vfi); // парсим формат
146                     di.formats.push_back(vfi);         // сохраняем формат в список
устройства
147                     ++idx;
148                 }
149             }
150         }
151
152         list.push_back(std::move(di));          // добавляем DeviceInfo в результирующий
вектор
153         act->Release();// явно релизим IMFActivate
154     }
155
156     CoTaskMemFree(ppDevices);         // освобождаем массив IMFActivate*
157     return list;      // возвращаем список устройств
158 }
159
160 // Публичная обёртка — вызывает внутреннюю реализацию
161 std::vector<DeviceInfo> EnumerateDevices() {
162     return EnumerateDevicesInternal();
163 }
```

VideoRecorder.h

```
000 #pragma once
001 #include <string>
002 #include "MFHelpers.h"
003
004 class VideoRecorder {
005 public:
006     VideoRecorder(int deviceIndex);
007     ~VideoRecorder();
008
009     HRESULT RecordToFile(const std::wstring& tmpPath,
010                          const std::wstring& finalPath,
011                          int seconds,
012                          std::wstring* usedDeviceName = nullptr,
013                          VideoFormatInfo* usedFmt = nullptr);
014 };
```

VideoRecorder.cpp

```
000 #include "VideoRecorder.h"
001 #include "Logger.h"
002 #include <mfapi.h>                      // Media Foundation API
003 #include <mfreadwrite.h>                // SourceReader / SinkWriter
004 #include <mfidl.h>                      // MF интерфейсы
005 #include <comdef.h>                     // _com_error
006 #include <chrono>                       // измерение времени записи
007 #include <thread>                       // sleep_for
008 #pragma comment(lib, "mfplat.lib")      // линковка MF
009 #pragma comment(lib, "mf.lib")
010 #pragma comment(lib, "mfreadwrite.lib")
011 using Microsoft::WRL::ComPtr;           // ComPtr для удобного управления COM-указателями
012
013 // Конструктор сохраняет индекс устройства
014 VideoRecorder::VideoRecorder(int deviceIndex) {}
015 VideoRecorder::~VideoRecorder() {}
016
017 // Основная функция записи видео в файл
```

```
018 HRESULT VideoRecorder::RecordToFile(const std::wstring& tmpPath, const std::wstring&
finalPath, int seconds,
019                                     std::wstring* usedDeviceName, VideoFormatInfo* usedFmt) {
020     Logger::Instance().Verbose(L"Starting recording");
021 
022     IMFAttributes* pAttr = nullptr;
023     HRESULT hr = MFCreateAttributes(&pAttr, 1);   // создаём атрибуты для перечисления
устройств
024     if (FAILED(hr)) {
025         Logger::Instance().Error(L"MFCreateAttributes failed: " + std::to_wstring((long)hr));
026         return hr;
027     }
028     ScopeGuard gAttr([&] { if (pAttr) pAttr->Release(); }); // релиз атрибутов при выходе
029 
030     pAttr->SetGUID(MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);
031     IMFActivate** ppDevices = nullptr;
032     UINT32 count = 0;
033     hr = MFEnumDeviceSources(pAttr, &ppDevices, &count);
034     if (FAILED(hr) || count == 0) {
035         Logger::Instance().Error(L"MFEnumDeviceSources failed or no devices");
036         CoTaskMemFree(ppDevices);
037         return E_FAIL;
038     }
039 
040     IMFActivate* act = ppDevices[0];               // берём первое устройство
041     WCHAR* friendly = nullptr;
042     if (SUCCEEDED(act->GetAllocatedString(MF_DEVSOURCE_ATTRIBUTE_FRIENDLY_NAME, &friendly,
nullptr))) {
043         if (usedDeviceName) *usedDeviceName = friendly;
044         CoTaskMemFree(friendly);
045     }
046 
047     ComPtr<IMFMediaSource> spSource;
048     hr = act->ActivateObject(IID_PPV_ARGS(&spSource));
049     if (FAILED(hr)) {
050         Logger::Instance().Error(L"ActivateObject failed: " + std::to_wstring((long)hr));
051         act->Release(); CoTaskMemFree(ppDevices);
052         return hr;
053     }
054 
055     ComPtr<IMFAttributes> readerAttr;
056     hr = MFCreateAttributes(&readerAttr, 1);
057     if (FAILED(hr)) {
058         Logger::Instance().Error(L"MFCreateAttributes(reader) failed: " +
std::to_wstring((long)hr));
059         spSource->Shutdown(); spSource.Reset(); act->Release(); CoTaskMemFree(ppDevices);
060         return hr;
061     }
062     readerAttr->SetUINT32(MF_READWRITE_ENABLE_HARDWARE_TRANSFORMS, TRUE);
063 
064     ComPtr<IMFSourceReader> reader;
065     hr = MFCreateSourceReaderFromMediaSource(spSource.Get(), readerAttr.Get(), &reader);
066     if (FAILED(hr)) {
067         Logger::Instance().Error(L"MFCreateSourceReaderFromMediaSource failed: " +
std::to_wstring((long)hr));
068         spSource->Shutdown(); spSource.Reset(); act->Release(); CoTaskMemFree(ppDevices);
069         return hr;
070     }
071 
072     ComPtr<IMFMediaType> pNativeType;
073     hr = reader->GetCurrentMediaType((DWORD)MF_SOURCE_READER_FIRST_VIDEO_STREAM,
&pNativeType);
074     if (FAILED(hr) || !pNativeType) {
075         hr = reader->GetNativeMediaType((DWORD)MF_SOURCE_READER_FIRST_VIDEO_STREAM, 0,
&pNativeType);
076     }
077     if (FAILED(hr) || !pNativeType) {
078         Logger::Instance().Error(L"Could not get native media type: " +
std::to_wstring((long)hr));
079         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
080         return hr;
081     }
082 
083     UINT32 width = 0, height = 0;
```

```
084     MFGetAttributeSize(pNativeType.Get(), MF_MT_FRAME_SIZE, &width, &height);
085     UINT32 num = 0, den = 0;
086     MFGetAttributeRatio(pNativeType.Get(), MF_MT_FRAME_RATE, &num, &den);
087     if (num == 0) { num = 30; den = 1; }
088
089     GUID preferredSub = MFVideoFormat_NV12;
090     ComPtr<IMFMediaType> pTryType;
091     hr = MFCreateMediaType(&pTryType);
092     if (SUCCEEDED(hr)) {
093         pTryType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
094         pTryType->SetGUID(MF_MT_SUBTYPE, preferredSub);
095         MFSetAttributeSize(pTryType.Get(), MF_MT_FRAME_SIZE, width, height);
096         MFSetAttributeRatio(pTryType.Get(), MF_MT_FRAME_RATE, num, den);
097         hr = reader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pTryType.Get());
098     }
099     bool usingNV12 = SUCCEEDED(hr);
100
101     if (!usingNV12) {
102         Logger::Instance().Verbose(L"NV12 not available, trying RGB32");
103         hr = MFCreateMediaType(&pTryType);
104         if (SUCCEEDED(hr)) {
105             pTryType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
106             pTryType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_RGB32);
107             MFSetAttributeSize(pTryType.Get(), MF_MT_FRAME_SIZE, width, height);
108             MFSetAttributeRatio(pTryType.Get(), MF_MT_FRAME_RATE, num, den);
109             hr = reader->SetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, nullptr,
pTryType.Get());
110         }
111         if (FAILED(hr)) {
112             Logger::Instance().Error(L"Failed to set reader output type to NV12 or RGB32: " +
std::to_wstring((long)hr));
113             reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
114             return hr;
115         }
116     }
117
118     ComPtr<IMFSinkWriter> sinkWriter;
119     std::wstring tmpForSink = tmpPath;
120     WCHAR ext[_MAX_EXT]{};
121     _wsplitpath_s(tmpPath.c_str(), nullptr, 0, nullptr, 0, nullptr, 0, ext, _MAX_EXT);
122     if (_wcsicmp(ext, L".mp4") != 0) {
123         tmpForSink = tmpPath + L".mp4";
124     }
125
126     hr = MFCreateSinkWriterFromURL(tmpForSink.c_str(), nullptr, nullptr, &sinkWriter);
127     if (FAILED(hr)) {
128         Logger::Instance().Error(L"MFCreateSinkWriterFromURL failed: " +
std::to_wstring((long)hr));
129         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
130         return hr;
131     }
132
133     ComPtr<IMFMediaType> pOutMediaType;
134     hr = MFCreateMediaType(&pOutMediaType);
135     if (FAILED(hr)) { Logger::Instance().Error(L"MFCreateMediaType(out) failed: " +
std::to_wstring((long)hr)); return hr; }
136     pOutMediaType->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
137     pOutMediaType->SetGUID(MF_MT_SUBTYPE, MFVideoFormat_H264);
138     MFSetAttributeSize(pOutMediaType.Get(), MF_MT_FRAME_SIZE, width, height);
139     MFSetAttributeRatio(pOutMediaType.Get(), MF_MT_FRAME_RATE, num, den);
140     pOutMediaType->SetUINT32(MF_MT_AVG_BITRATE, 4000000);
141     pOutMediaType->SetUINT32(MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
142
143     DWORD outStreamIndex = 0;
144     hr = sinkWriter->AddStream(pOutMediaType.Get(), &outStreamIndex);
145     if (FAILED(hr)) {
146         Logger::Instance().Error(L"AddStream failed: " + std::to_wstring((long)hr));
147         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
148         return hr;
149     }
150
151     ComPtr<IMFMediaType> pReaderType;
```

```
152     hr = reader->GetCurrentMediaType(MF_SOURCE_READER_FIRST_VIDEO_STREAM, &pReaderType);
153     if (FAILED(hr) || !pReaderType) {
154         Logger::Instance().Error(L"GetCurrentMediaType failed: " +
std::to_wstring((long)hr));
155         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
156         return hr;
157     }
158
159     hr = sinkWriter->SetInputMediaType(outStreamIndex, pReaderType.Get(), nullptr);
160     if (FAILED(hr)) {
161         Logger::Instance().Error(L"SetInputMediaType failed: " + std::to_wstring((long)hr));
162         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
163         return hr;
164     }
165
166     hr = sinkWriter->BeginWriting();
167     if (FAILED(hr)) {
168         Logger::Instance().Error(L"BeginWriting failed: " + std::to_wstring((long)hr));
169         reader.Reset(); spSource->Shutdown(); spSource.Reset(); act->Release();
CoTaskMemFree(ppDevices);
170         return hr;
171     }
172
173     auto start = std::chrono::steady_clock::now();
174     while (true) {
175         ComPtr<IMFSample> pSample;
176         DWORD dwStreamIndex = 0, dwFlags = 0;
177         LONGLONG llTimeStamp = 0;
178         HRESULT r = reader->ReadSample((DWORD)MF_SOURCE_READER_FIRST_VIDEO_STREAM, 0,
179                                        &dwStreamIndex, &dwFlags, &llTimeStamp, &pSample);
180         if (FAILED(r)) {
181             Logger::Instance().Error(L"ReadSample failed during recording: " +
std::to_wstring((long)r));
182             break;
183         }
184         if (dwFlags & MF_SOURCE_READERF_ENDOFSTREAM) {
185             Logger::Instance().Verbose(L"Source signalled EOS");
186             break;
187         }
188         if (pSample) {
189             hr = sinkWriter->WriteSample(outStreamIndex, pSample.Get());
190             if (FAILED(hr)) {
191                 Logger::Instance().Error(L"WriteSample failed: " +
std::to_wstring((long)hr));
192                 break;
193             }
194         }
195         auto elapsed = std::chrono::steady_clock::now() - start;
196         if (std::chrono::duration_cast<std::chrono::seconds>(elapsed).count() >= seconds)
break;
197         std::this_thread::sleep_for(std::chrono::milliseconds(5));
198     }
199
200     hr = sinkWriter->Finalize();
201     if (FAILED(hr)) Logger::Instance().Error(L"Finalize failed: " +
std::to_wstring((long)hr));
202
203     reader.Reset();
204     spSource->Shutdown();
205     spSource.Reset();
206     act->Release();
207     CoTaskMemFree(ppDevices);
208
209     if (!MoveFileExW(tmpForSink.c_str(), finalPath.c_str(), MOVEFILE_COPY_ALLOWED |
MOVEFILE_REPLACE_EXISTING)) {
210         if (CopyFileW(tmpForSink.c_str(), finalPath.c_str(), FALSE)) {
211             DeleteFileW(tmpForSink.c_str());
212         }
213         else {
214             MoveFileExW(tmpPath.c_str(), finalPath.c_str(), MOVEFILE_COPY_ALLOWED |
MOVEFILE_REPLACE_EXISTING);
215         }
216     }
217
```

```
218     Logger::Instance().Verbose(L"Recording saved: " + finalPath);
219     return S_OK;
220 }
```

ScopeGuard.h

```
000 #pragma once
001 #include <functional>
002
003 class ScopeGuard {
004     std::function<void()> fn_;
005     bool active_{ true };
006 public:
007     explicit ScopeGuard(std::function<void()> f) : fn_(std::move(f)) {}
008     ~ScopeGuard() { if (active_) fn_(); }
009     void dismiss() { active_ = false; }
010 };
```

main.cpp

```
000 // main.cpp
001 #include <windows.h>                // WinAPI базовые функции и типы
002 #include <objbase.h>                // COM инициализация/типы
003 #include <iostream>
004 #include <string>
005 #include <vector>
006 #include <optional>
007 #include <locale>                   // локали
008 #include <codecvt>                  // преобразования кодировок
009
010 #include <shlwapi.h>                // PathIsRelative, PathRemoveFileSpec
011 #pragma comment(lib, "shlwapi.lib") // линковка shlwapi
012
013 #include <mfapi.h>                  // Media Foundation API
014 #include <mfidl.h>                  // MF интерфейсы
015 #include <mfobjects.h>              // MF объекты
016 #include <mfreadwrite.h>            // source/sink reader/writer
017 #include <mftransform.h>            // MFT типы
018 #include <mferror.h>                // HRESULT MF коды
019 #pragma comment(lib, "mfplat.lib")  // линковка MF
020 #pragma comment(lib, "mf.lib")
021 #pragma comment(lib, "mfreadwrite.lib")
022 #pragma comment(lib, "mfuuid.lib")
023
024 #include <wincodec.h>               // WIC (запись JPEG)
025 #pragma comment(lib, "windowscodecs.lib")
026
027 #include <wrl/client.h>             // ComPtr
028 using Microsoft::WRL::ComPtr;       // удобный тип для COM указателей
029
030 #include "CommandLine.h"            // парсер аргументов
031 #include "Logger.h"                 // централизованный логгер
032 #include "DeviceEnumerator.h"       // перечисление камер
033 #include "FrameGrabber.h"           // снимок в JPEG
034 #include "VideoRecorder.h"          // запись видео в MP4
035 #include "MFHelpers.h"              // вспомогательные MF утилиты
036 #include "ScopeGuard.h"             // RAII для очистки
037
038 using namespace std;
039
040 // Возвращает полный путь к исполняемому файлу
041 static wstring GetExePath() {
042     wchar_t buf[MAX_PATH]{};
043     GetModuleFileNameW(nullptr, buf, MAX_PATH);
044     return wstring(buf);
045 }
046
047 // Нормализует и создаёт выходную директорию
048 static wstring NormalizeOutputPath(const optional<wstring>& outArg, const wstring& exePath) {
049     wchar_t cur[MAX_PATH]{};
050     GetCurrentDirectoryW(MAX_PATH, cur);
051     wstring out;
052     if (!outArg.has_value()) out = wstring(cur);
053     else {
054         wstring v = outArg.value();
055         if (v == L".") {
056             wchar_t exedir[MAX_PATH];
057             wcscpy_s(exedir, exePath.c_str());
```

```
058             PathRemoveFileSpecW(exedir);
059             out = exedir;
060         }
061         else if (PathIsRelativeW(v.c_str())) {
062             out = wstring(cur) + L"\\" + v;
063         }
064         else out = v;
065     }
066     CreateDirectoryW(out.c_str(), nullptr);
067     return out;
068 }
069
070 // Формирует имя файла с таймстампом и расширением
071 static wstring MakeFilename(const wstring& dir, const wstring& ext) {
072     SYSTEMTIME st;
073     GetLocalTime(&st);
074     wchar_t buf[128];
075     swprintf_s(buf, L"%04d-%02d-%02d_%02d-%02d-%02d%s",
076         st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond, ext.c_str());
077     wstring path = dir;
078     if (!path.empty() && path.back() != L'\\' && path.back() != L'/') path += L"\\";
079     path += buf;
080     return path;
081 }
082
083 // У процесса нет консоли — аллоцируем и перенаправляем stdio
084 static bool EnsureConsole() {
085     if (GetConsoleWindow()) return false;
086     if (!AllocConsole()) return false;
087     FILE* fOut = nullptr;
088     freopen_s(&fOut, "CONOUT$", "w", stdout);
089     freopen_s(&fOut, "CONOUT$", "w", stderr);
090     FILE* fIn = nullptr;
091     freopen_s(&fIn, "CONIN$", "r", stdin);
092     std::ios::sync_with_stdio(false);
093     SetConsoleOutputCP(CP_UTF8);
094     SetConsoleCP(CP_UTF8);
095     HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
096     if (hOut && hOut != INVALID_HANDLE_VALUE) {
097         DWORD mode = 0;
098         if (GetConsoleMode(hOut, &mode)) {
099             mode |= ENABLE_PROCESSED_OUTPUT;
100             SetConsoleMode(hOut, mode);
101         }
102     }
103     return true;
104 }
105
106 // Если консоль была выделена этой программой — ждём нажатия Enter
107 static void PauseIfConsoleAllocated(bool allocated) {
108     if (!allocated) return;
109     Logger::Instance().Info(L"\nНажмите Enter для выхода...");
110     std::wcout.flush();
111     HANDLE hStdin = GetStdHandle(STD_INPUT_HANDLE);
112     if (hStdin && hStdin != INVALID_HANDLE_VALUE) {
113         wchar_t buf[16] = { 0 };
114         DWORD read = 0;
115         ReadConsoleW(hStdin, buf, (DWORD)(std::size(buf) - 1), &read, nullptr);
116     }
117     else {
118         std::wstring dummy;
119         std::getline(std::wcin, dummy);
120     }
121 }
122
123 int wmain(int argc, wchar_t** argv) {
124     std::wstring parseErr;
125     auto opt = CommandLineParser::Parse(argc, argv, parseErr);
126     if (!opt) {
127         MessageBoxW(nullptr, parseErr.c_str(), L"Ошибка запуска", MB_ICONERROR);
128         return -1;
129     }
130
131     bool wantConsole = !opt->quiet;
132     bool consoleAllocated = false;
133     if (wantConsole) consoleAllocated = EnsureConsole();
```

```
134
135     auto exePath = GetExePath();
136
137     Logger::Instance().InitConsole(wantConsole, opt->verbose);
138     Logger::Instance().Info(L"Программа запущена");
139
140     Logger::Instance().Verbose(L"Parse success");
141
142     HRESULT hr = MFStartup(MF_VERSION);
143     if (FAILED(hr)) {
144         Logger::Instance().Error(L"MFStartup failed: HRESULT=" + to_wstring((long)hr));
145         PauseIfConsoleAllocated(consoleAllocated);
146         return -1;
147     }
148     ScopeGuard mfGuard([&] { MFShutdown(); });
149
150     DeviceEnumerator de;
151
152     if (opt->info) {
153         auto devices = de.ListDevices();
154         if (devices.empty()) {
155             Logger::Instance().Info(L"Не найдено ни одной веб-камеры");
156         }
157         else {
158             for (size_t i = 0; i < devices.size(); ++i) {
159                 const auto& d = devices[i];
160                 if (wantConsole) {
161                     std::wcout << L"[" << i << L"] " << d.name << L"\n";
162                     std::wcout << L"  id: " << d.id << L"\n";
163                     std::wcout << L"  vendor: " << (d.vendor.empty() ? L"(unknown)" :
d.vendor) << L"\n";
164                     std::wcout << L"  formats:\n";
165                     for (const auto& f : d.formats) {
166                         wchar_t buf[256];
167                         swprintf_s(buf, L"    %ux%u @ %u/%u fps subtype: %ls bitDepth:%u\n",
168                             f.width, f.height, f.fpsNumerator, f.fpsDenominator,
169                             GuidToString(f.subtype).c_str(), f.bitDepth);
170                         std::wcout << buf;
171                     }
172                     std::wcout << L"\n";
173                 }
174                 Logger::Instance().Verbose(L"Device[" + std::to_wstring(i) + L"] " + d.name);
175             }
176         }
177         PauseIfConsoleAllocated(consoleAllocated);
178         return 0;
179     }
180
181     wstring outDir = NormalizeOutputPath(opt->outputPath, exePath);
182     int devIdx = opt->deviceId.value_or(0);
183
184     if (opt->snap) {                                  // режим --snap: сделать снимок
185         wstring filePath = MakeFilename(outDir, L".jpg"); // имя выходного файла
186         FrameGrabber fg(devIdx);                       // создаём объект FrameGrabber
187         std::wstring usedDevName;
188         VideoFormatInfo usedFmt{};
189         Logger::Instance().Verbose(L"Starting CaptureToJpeg: device=" + to_wstring(devIdx) +
L" out=" + filePath);
190
191         HRESULT r = fg.CaptureToJpeg(filePath, 95, &usedDevName, &usedFmt); // основной вызов
захвата
192         Logger::Instance().Verbose(L"CaptureToJpeg returned HRESULT=" + to_wstring((long)r));
193
194         if (FAILED(r)) {
195             Logger::Instance().Error(L"CaptureToJpeg failed. HRESULT=" +
to_wstring((long)r));
196             PauseIfConsoleAllocated(consoleAllocated);
197             return (int)r;
198         }
199
200         Logger::Instance().Info(L"Фото сохранено: " + filePath);
201         PauseIfConsoleAllocated(consoleAllocated);
202         return 0;
203     }
204
205     if (opt->capture) {
```

```
206         wstring tmpPath = MakeFilename(outDir, L".tmp");
207         wstring finalPath = MakeFilename(outDir, L".mp4");
208         VideoRecorder vr(devIdx);
209         std::wstring usedDevName;
210         VideoFormatInfo usedFmt{};
211         Logger::Instance().Verbose(L"Starting RecordToFile: device=" + to_wstring(devIdx) +
L" final=" + finalPath);
212
213         HRESULT r = vr.RecordToFile(tmpPath, finalPath, opt->captureSeconds, &usedDevName,
&usedFmt);
214         Logger::Instance().Verbose(L"RecordToFile returned HRESULT=" + to_wstring((long)r));
215
216         if (FAILED(r)) {
217             Logger::Instance().Error(L"RecordToFile failed. HRESULT=" + to_wstring((long)r));
218             DeleteFileW(tmpPath.c_str());
219             PauseIfConsoleAllocated(consoleAllocated);
220             return (int)r;
221         }
222
223         Logger::Instance().Info(L"Видео сохранено: " + finalPath);
224         PauseIfConsoleAllocated(consoleAllocated);
225         return 0;
226     }
227
228     Logger::Instance().Info(L"Не выбрано действие. Укажите --info, --snap или --capture");
229     PauseIfConsoleAllocated(consoleAllocated);
230     return 0;
231 }
```

5 Вывод

В ходе выполнения лабораторной работы написана программа, выводящая список всех устройств, подключенных к шине PCI. Дополнительно к программе написан драйвер для шины PCI, осуществляющий низкоуровневое взаимодействие с консольной программой.