Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе №8

на тему

Интерфейс сокетов и основы сетевого программирования (*Windows*). Программирование взаимодействия через сеть с использованием интерфейса сокетов. Реализация сетевых протоколов: собственных или стандартных.

Студент Н. В. Климкович

Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 5](#_Toc146752070)

[Заключение 6](#_Toc146752071)

[Список использованных источников 7](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 8](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Целью данной работы является создание многофункционального графического редактора, объединяющее графический редактор и инструменты для взаимодействия через сеть с использованием интерфейса сокетов. Для достижения цели будет создано клиент-серверное приложение для обмена текстовыми сообщениями с использованием *TCP*.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Библиотека *Winsock* предоставляет набор функций и структур, необходимых для создания сетевых приложений в среде *Windows*. Она позволяет приложениям создавать сокеты, устанавливать соединения, отправлять и получать данные через сеть, а также управлять сетевыми настройками.

Сокет является абстракцией, предоставляемая операционной системой, которая позволяет приложениям взаимодействовать через сеть. В *Windows* для сетевого программирования с использованием сокетов предоставляются функции и структуры из библиотеки *Winsock* (*Windows* *Sockets*), которая является стандартом для разработки сетевых приложений. Протоколы, поддерживаемые *Winsock*, включают *TCP*/*IP*, *UDP*, и другие.

В *Winsock* существуют разные типы сокетов, включая: *SOCK\_STREAM* (сокеты *TCP*): Представляют собой потоковые сокеты, которые обеспечивают надежную и упорядоченную передачу данных. Они используют протокол *TCP*. *SOCK\_DGRAM* (сокеты *UDP*): Представляют датаграммные сокеты, которые обеспечивают ненадежную передачу данных, но без установления соединения. Они используют протокол *UDP*. *SOCK\_RAW* (сокеты *RAW*): Позволяют приложению работать на низком уровне с сетевыми пакетами. Они обеспечивают полный доступ к сетевому стеку.

Для создания сокета в Windows используется функция *socket*. Она принимает параметры, такие как домен (*AF\_INET* для *IPv4*), тип сокета (*SOCK\_STREAM* или *SOCK\_DGRAM*), и протокол (обычно 0, чтобы выбрать соответствующий протокол для домена и типа).

Для клиентского приложения в сокетах *TCP* используется функция *connect*, чтобы установить соединение с сервером. Для сервера используется функция *bind*, чтобы связать сокет с конкретным адресом и портом, и *listen*, чтобы начать прослушивание входящих соединений.

После успешного установления соединения, вы можете использовать функции *send* и *recv* для отправки и приема данных через сокет.

После завершения работы с сокетом, его следует закрыть с использованием функции *closesocket*.

Для создания многопоточных серверов, способных обслуживать несколько клиентов одновременно, обычно используются многопоточные или асинхронные подходы. Каждое новое входящее соединение может обрабатываться в отдельном потоке или асинхронно.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В ходе выполнения лабораторной работы было разработано многофункциональное графическое приложение, используя *Win32 API*. Приложение объединяет графический редактор с инструментами для обмена текстовыми сообщениями между клиентами по локальной сети с использованием сокетов и *TCP* протокола. Самописный сервер обрабатывает сразу несколько пользователей, используя многопоточность, где для одного пользователя будет один поток (Рисунок 1).

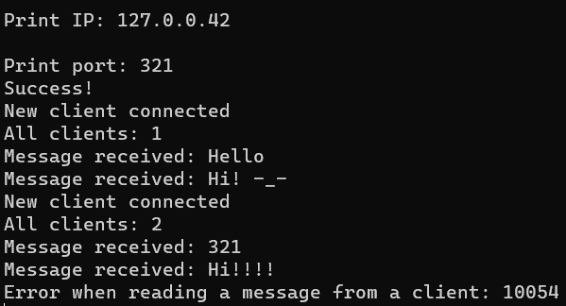


Рисунок 1 – Серверное окно приложения

Также клиентское приложение позволяет взаимодействовать с сервером и получать текстовые сообщения (Рисунок 2).

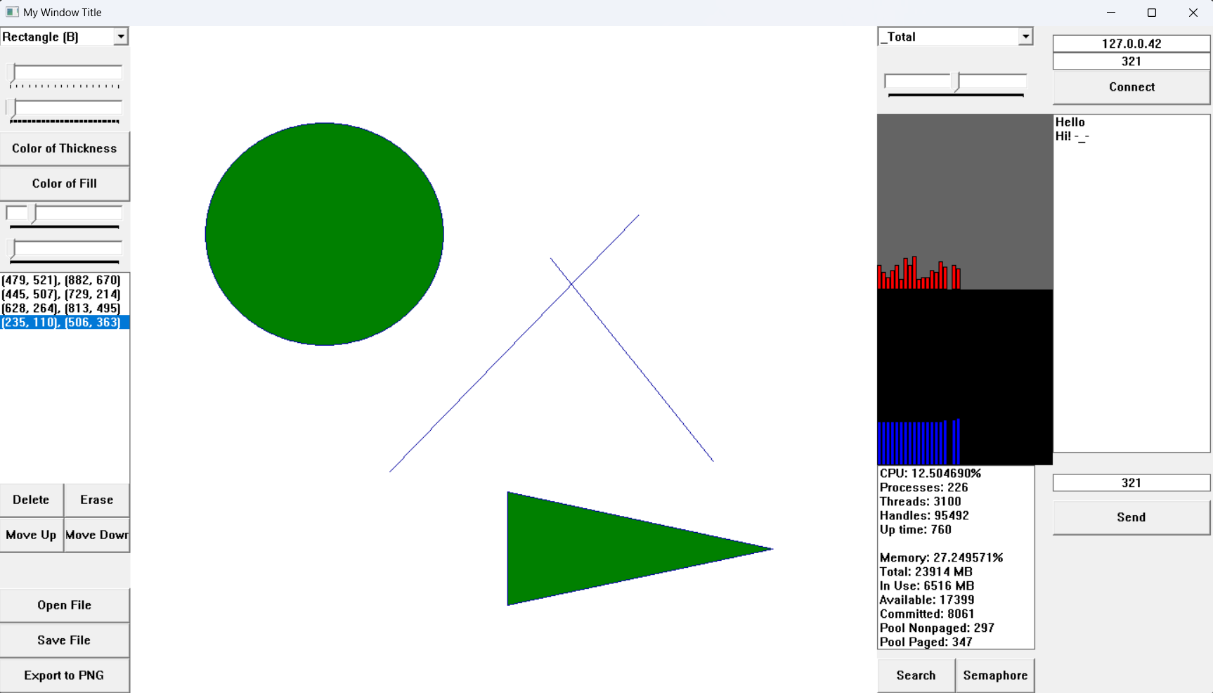


Рисунок 2 – Клиентское окно приложения

# ЗАКЛЮЧЕНИЕ

В данной лабораторной работе были изучены и применены основы сетевого программирования в среде *Windows* с использованием сокетов. Мы разработали клиент-серверное приложение, позволяющее обмениваться текстовыми сообщениями через сеть с использованием протокола *TCP*. Клиентское приложение было адаптировано для ввода адреса сервера и порта, что обеспечивает более гибкую настройку подключения.

Изучение и использование функций, предоставляемых *Winsock* *API* для создания сокетов клиента и сервера.

Данная лабораторная работа позволила ознакомиться с основами сетевого программирования, сокетами, и применить полученные знания для разработки сетевого приложения. Мы также рассмотрели создание серверов, способных обслуживать несколько клиентов одновременно, что поднимает понимание сетевых взаимодействий на новый уровень.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Щупак Ю. *Win32 API*. Разработка приложений для *Windows*. ─ СПБ: Питер, 2008. ─ 592 с.: ип.
2. Создание классических приложений для *Windows* с использованием *API Win32* [Электронный ресурс]. ─ Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – Файл StartServer.cpp

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdio.h>

#include <process.h>

#include <vector>

#include <mutex>

#include <iostream>

#include <string>

#pragma comment(lib, "ws2\_32.lib")

std::vector<SOCKET> clients;

std::mutex clientsMutex;

void ClientHandler(void\* socketPtr) {

SOCKET clientSocket = \*((SOCKET\*)socketPtr);

free(socketPtr);

char buffer[250];

int bytesRead;

do {

bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);

if (bytesRead > 0) {

buffer[bytesRead] = '\0';

std::cout << "Message received: " << buffer << '\n';

{

std::lock\_guard<std::mutex> lock(clientsMutex);

for (SOCKET& otherClient : clients) {

send(otherClient, buffer, bytesRead, 0);

}

}

}

else if (bytesRead == 0) {

std::cout << "The client disconnected\n";

}

else {

std::cout << "Error when reading a message from a client: " << WSAGetLastError() << '\n';

}

} while (bytesRead > 0);

closesocket(clientSocket);

{

std::lock\_guard<std::mutex> lock(clientsMutex);

clients.erase(std::remove(clients.begin(), clients.end(), clientSocket), clients.end());

}

\_endthread();

}

int main() {

WSADATA wsaData;

int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

std::cout << "Error during Winsock initialization: " << iResult;

return 1;

}

SOCKET serverSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (serverSocket == INVALID\_SOCKET) {

std::cout << "Error when creating a socket: " << WSAGetLastError();

WSACleanup();

return 1;

}

std::string ip;

int port;

sockaddr\_in serverAddr;

while (true) {

std::cout << "\nPrint IP: ";

std::cin >> ip;

std::cout << "\nPrint port: ";

std::cin >> port;

serverAddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, ip.c\_str(), &serverAddr.sin\_addr);

serverAddr.sin\_port = htons(port);

if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {

std::cout << "Error during socket binding: " << WSAGetLastError() << '\n';

continue;

}

break;

}

std::cout << "Success!\n";

if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR) {

std::cout << "Error on socket listening: " << WSAGetLastError();

closesocket(serverSocket);

WSACleanup();

return 1;

}

while (1) {

SOCKET clientSocket = accept(serverSocket, NULL, NULL);

if (clientSocket == INVALID\_SOCKET) {

std::cout << "Error accepting connection: " << WSAGetLastError();

closesocket(serverSocket);

WSACleanup();

return 1;

}

std::cout << "New client connected\n";

{

std::lock\_guard<std::mutex> lock(clientsMutex);

clients.push\_back(clientSocket);

}

std::cout << "All clients: " << clients.size() << '\n';

SOCKET\* clientSocketPtr = (SOCKET\*)malloc(sizeof(SOCKET));

\*clientSocketPtr = clientSocket;

\_beginthread(ClientHandler, 0, clientSocketPtr);

}

closesocket(serverSocket);

WSACleanup();

return 0;

}

Листинг 2 – Файл Lab1.cpp

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#include <vector>

#include <CommCtrl.h>

#include <commdlg.h>

#include <string>

#include <GdiPlus.h>

#include <iostream>

#include <pdh.h>

#include <thread>

#include <Wininet.h>

#include <deque>

#include <chrono>

#include <Psapi.h>

#include <future>

#pragma comment(lib, "ws2\_32.lib")

#pragma comment(lib, "pdh.lib")

#pragma comment(lib, "gdiplus.lib")

#pragma comment(lib, "user32.lib")

#pragma comment(lib, "wininet.lib")

#define M\_PI 3.141592653589793238462643383279

#define WM\_SEARCH\_REG (WM\_APP + 1)

#define WM\_DELETE\_REG (WM\_APP + 2)

#define WM\_SERVER (WM\_APP + 3)

#define MYBYTES 1048576

#pragma region const

struct Shape

{

RECT rect;

bool isCorrect;

int n;

int selectedShape;

// 1 - circle

// 2 - square

// 3 - Triangle

// 4 - Ellipse

int Thickness;

COLORREF selectedColorThickness;

COLORREF selectedColorBrush;

std::vector<Gdiplus::Point> pen;

int scale = 100;

int rotation = 0;

};

struct PaintWindow

{

int x1 = 150;

int y1 = 0;

int x2 = 1000;

int y2 = 800;

int Width = x2 - x1;

int Height = y2 - y1;

};

struct RegistryEntry {

HKEY parentKey;

std::wstring subkeyName;

TCHAR\* szPath;

};

// Global variables

HINSTANCE hInst;

HDC hdcBuffer;

HBITMAP hBitmap;

HWND hwndMain, hwndListServer, hwndEditServer, hwndEditPort, hwndEditIP, hButtonPort, hButtonSend, hwndComboBox, hSlider, hSliderCP, hSliderThickness, hSliderScale, hSliderRotation, hwndList, hwndListCP, hwndDeleteItem, hwndUpItem, hwndDownItem, hwndSave, hwndComboBoxCP, hButton6;

COLORREF customColorsThickness[16]{ 0 };

COLORREF customColorsBrush[16]{ 0 };

CHOOSECOLOR ccThickness, ccBrush;

COLORREF selectedColorThickness, selectedColorBrush;

HHOOK MouseHook;

PaintWindow PW;

int selectedItemIndex = -1;

std::vector<Shape> shapes;

RECT currentShape;

bool isDrawing = false;

bool isMove = false;

bool isTaskManager = false;

int selectedShape = 1;

int n = 3;

int speed = 1000;

int Thickness = 1;

std::vector<Gdiplus::Point> pen;

Gdiplus::Point startPos;

std::vector<float> cpuLoadHistory;

std::vector<float> memoryLoadHistory;

std::vector<std::wstring> StarusCP;

std::vector<RegistryEntry> emptyValues;

HANDLE Search, Delete, hMutex, hSemaphore, ServerThread;

long long deleteValues = 0;

bool isConnected = false;

SOCKET clientSocket;

#pragma endregion

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK MouseHookProc(int nCode, WPARAM wParam, LPARAM lParam);

void DrawShape(Gdiplus::Graphics& graphics, Gdiplus::Pen\* penPlus, Gdiplus::SolidBrush\* brush, bool isCorrect, int scale = 100, int rotation = 0);

void FillRectWindow();

void RePaint(bool ctrlZ, bool del, bool list = false);

void UpdateData();

void DrawGraph(HDC hdc, const std::vector<float>& data, int y, COLORREF color);

int GetEncoderClsid(const WCHAR\* format, CLSID\* pClsid);

void Monitoring(HMONITOR hMonitor, HDC hdcMonitor, LPRECT lprcMonitor, LPARAM dwData) {

DEVMODE prevDevMode, currentDevMode;

SYSTEM\_POWER\_STATUS prevPowerStatus, currentPowerStatus;

BOOL prevInternet, currentInternet;

prevInternet = InternetGetConnectedState(NULL, 0);

EnumDisplaySettings(nullptr, ENUM\_CURRENT\_SETTINGS, &prevDevMode);

GetSystemPowerStatus(&prevPowerStatus);

while (true) {

EnumDisplaySettings(nullptr, ENUM\_CURRENT\_SETTINGS, &currentDevMode);

GetSystemPowerStatus(&currentPowerStatus);

currentInternet = InternetGetConnectedState(NULL, 0);

if (prevDevMode.dmPelsWidth != currentDevMode.dmPelsWidth || prevDevMode.dmPelsHeight != currentDevMode.dmPelsHeight) {

std::wstring message2 = L"Display resolution changed: ";

message2 += std::to\_wstring(currentDevMode.dmPelsWidth);

message2 += L" x ";

message2 += std::to\_wstring(currentDevMode.dmPelsHeight);

MessageBox(NULL, message2.c\_str(), L"", MB\_ICONINFORMATION | MB\_OK);

prevDevMode = currentDevMode;

}

if (prevPowerStatus.ACLineStatus != currentPowerStatus.ACLineStatus) {

if (currentPowerStatus.ACLineStatus == 0) {

MessageBox(NULL, L"Power source switched to battery", L"", MB\_ICONINFORMATION | MB\_OK);

}

else {

MessageBox(NULL, L"Power source switched to AC power", L"", MB\_ICONINFORMATION | MB\_OK);

}

prevPowerStatus = currentPowerStatus;

}

if (currentInternet != prevInternet) {

if (currentInternet == TRUE) {

MessageBox(NULL, L"Internet connection established", L"", MB\_ICONINFORMATION | MB\_OK);

}

else {

MessageBox(NULL, L"No internet connection", L"", MB\_ICONINFORMATION | MB\_OK);

}

prevInternet = currentInternet;

}

std::this\_thread::sleep\_for(std::chrono::seconds(1));

}

}

void SemaforTask() {

DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(hSemaphore, 100000L);

switch (dwWaitResult)

{

case WAIT\_OBJECT\_0: {

std::this\_thread::sleep\_for(std::chrono::seconds(2));

MessageBox(NULL, L"Semaphore", L"", MB\_ICONINFORMATION | MB\_OK);

ReleaseSemaphore(hSemaphore, 1, NULL);

break;

}

}

}

void GetMessages() {

while (true) {

char buffer[250];

int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);

if (bytesReceived > 0) {

buffer[bytesReceived] = '\0';

int wBufferSize = MultiByteToWideChar(CP\_ACP, 0, buffer, -1, NULL, 0);

wchar\_t\* wBuffer = new wchar\_t[wBufferSize];

MultiByteToWideChar(CP\_ACP, 0, buffer, -1, wBuffer, wBufferSize);

SendMessage(hwndListServer, LB\_ADDSTRING, 0, (LPARAM)wBuffer);

delete[] wBuffer;

}

}

}

void GetEmptyValues(HKEY hRootKey, TCHAR\* szPath)

{

HKEY hKey = NULL;

LONG lResult = RegOpenKeyEx(hRootKey, szPath, 0, KEY\_READ, &hKey);

if (lResult != ERROR\_SUCCESS)

return;

DWORD dwSubKeys = 0;

DWORD dwValues = 0;

lResult = RegQueryInfoKey(hKey, NULL, NULL, NULL, &dwSubKeys, NULL, NULL, &dwValues, NULL, NULL, NULL, NULL);

if (lResult != ERROR\_SUCCESS) {

RegCloseKey(hKey);

return;

}

for (DWORD i = 0; i < dwSubKeys; i++)

{

DWORD lpdwMAX\_PATH = MAX\_PATH;

TCHAR szSubKeyName[MAX\_PATH] = { 0 };

lResult = RegEnumKeyEx(hKey, i, szSubKeyName, &lpdwMAX\_PATH, NULL, NULL, NULL, NULL);

if (lResult != ERROR\_SUCCESS)

return;

GetEmptyValues(hKey, szSubKeyName);

}

bool check = true;

for (DWORD i = 0; i < dwValues; i++)

{

DWORD lpdwMAX\_PATH = MAX\_PATH;

TCHAR szValueName[MAX\_PATH] = { 0 };

lResult = RegEnumValue(hKey, i, szValueName, &lpdwMAX\_PATH, NULL, NULL, NULL, NULL);

if (lResult == ERROR\_SUCCESS)

{

DWORD dwSize = MAX\_PATH;

TCHAR szValueData[MAX\_PATH] = { 0 };

DWORD dwType = 0;

lResult = RegQueryValueEx(hKey, szValueName, 0, &dwType, (LPBYTE)szValueData, &dwSize);

// if (lResult == ERROR\_SUCCESS && dwType == REG\_SZ && szValueData[0] == '\0')

if (lResult == ERROR\_SUCCESS && szValueData[0] == '\0')

{

check = false;

RegistryEntry buf;

buf.parentKey = hKey;

buf.subkeyName = szValueName;

buf.szPath = szPath;

emptyValues.push\_back(buf);

}

}

}

if (check)

RegCloseKey(hKey);

}

void DeleteEmptyValues()

{

deleteValues = 0;

for (RegistryEntry hKey : emptyValues)

{

HKEY key = NULL;

LONG lResult = RegOpenKeyEx(hKey.parentKey, hKey.szPath, 0, KEY\_READ | KEY\_WRITE, &key);

if (lResult != ERROR\_SUCCESS)

continue;

lResult = RegDeleteValue(key, hKey.subkeyName.c\_str());

RegCloseKey(key);

if (lResult == ERROR\_SUCCESS)

deleteValues++;

}

PostMessage(hwndMain, WM\_DELETE\_REG, 0, 0);

}

void StartSearch() {

DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(hMutex, 200L);

switch (dwWaitResult)

{

case WAIT\_OBJECT\_0: {

MessageBox(NULL, L"Search started", L"", MB\_ICONINFORMATION | MB\_OK);

emptyValues.clear();

HKEY baseKeys[] = { HKEY\_LOCAL\_MACHINE, HKEY\_CURRENT\_USER, HKEY\_USERS, HKEY\_CURRENT\_CONFIG, HKEY\_CLASSES\_ROOT };

for (HKEY baseKey : baseKeys) {

TCHAR szPath[MAX\_PATH] = { 0 };

GetEmptyValues(baseKey, szPath);

}

PostMessage(hwndMain, WM\_SEARCH\_REG, 0, 0);

break;

}

case WAIT\_ABANDONED:

MessageBox(NULL, L"Search was not initiated", L"Error", MB\_ICONERROR | MB\_OK);

return;

}

ReleaseMutex(hMutex);

}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

Gdiplus::GdiplusStartupInput gdiplusStartupInput;

ULONG\_PTR gdiplusToken;

Gdiplus::GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

//MouseHook = SetWindowsHookEx(WH\_MOUSE\_LL, MouseHookProc, NULL, 0);

WSADATA wsaData;

int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

MessageBox(NULL, L"Error Winsock", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

ServerThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)GetMessages, NULL, 0, NULL);

hMutex = CreateMutex(NULL, FALSE, NULL);

if (hMutex == NULL)

{

MessageBox(NULL, L"Mutex creation failed", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

hSemaphore = CreateSemaphore(NULL, 2, 2, NULL);

if (hSemaphore == NULL)

{

MessageBox(NULL, L"Semaphore creation failed", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS\_HREDRAW | CS\_VREDRAW, WndProc, 0, 0, GetModuleHandle(NULL), NULL, NULL, NULL, NULL, L"MyWindowClass", NULL };

RegisterClassEx(&wc);

hwndMain = CreateWindow(L"MyWindowClass", L"My Window Title", WS\_OVERLAPPEDWINDOW | WS\_CLIPCHILDREN, NULL, NULL, PW.x2 + 400, PW.y2, NULL, NULL, hInstance, NULL);

SetWindowLong(hwndMain, GWL\_STYLE, GetWindowLong(hwndMain, GWL\_STYLE) & ~WS\_THICKFRAME);

HDC hdc = GetDC(hwndMain);

hdcBuffer = CreateCompatibleDC(hdc);

hBitmap = CreateCompatibleBitmap(hdc, PW.Width, PW.Height);

SelectObject(hdcBuffer, hBitmap);

ShowWindow(hwndMain, nCmdShow);

RECT rectSlider{ 0,0,PW.x1,PW.y2 };

FillRect(hdc, &rectSlider, (HBRUSH)(COLOR\_WINDOW));

RECT rectSlider2{ PW.x2,0,PW.x2 + 400,PW.y2 };

FillRect(hdc, &rectSlider2, (HBRUSH)(COLOR\_WINDOW));

ReleaseDC(hwndMain, hdc);

UpdateWindow(hwndMain);

HANDLE hThreadUpdateData = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)UpdateData, NULL, 0, NULL);

HANDLE hThreadMonitoring = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)Monitoring, NULL, 0, NULL);

MSG msg;

while (GetMessage(&msg, NULL, 0, 0))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

// UnhookWindowsHookEx(MouseHook);

Gdiplus::GdiplusShutdown(gdiplusToken);

DeleteObject(hBitmap);

DeleteDC(hdcBuffer);

TerminateThread(hThreadUpdateData, 0);

CloseHandle(hThreadUpdateData);

TerminateThread(hThreadMonitoring, 0);

CloseHandle(hThreadMonitoring);

closesocket(clientSocket);

WSACleanup();

return msg.wParam;

}

LRESULT CALLBACK MouseHookProc(int nCode, WPARAM wParam, LPARAM lParam)

{

if (nCode == HC\_ACTION)

{

if (wParam == WM\_MOUSEMOVE)

{

if (isDrawing || isMove) {

POINT mousePos;

GetCursorPos(&mousePos);

RECT windowRect;

GetWindowRect(hwndMain, &windowRect);

if (mousePos.x < windowRect.left + PW.x1 || mousePos.x > windowRect.right - 200 ||

mousePos.y < windowRect.top || mousePos.y > windowRect.bottom)

{

if (isDrawing) {

isDrawing = false;

InvalidateRect(hwndMain, NULL, TRUE);

if (selectedShape == 4) {

pen.push\_back({ LOWORD(lParam) ,HIWORD(lParam) });

}

shapes.push\_back({ currentShape, bool(GetKeyState(VK\_SHIFT) & 0x8000),n,selectedShape, Thickness,selectedColorThickness,selectedColorBrush,pen });

pen.clear();

wchar\_t buffer[30];

swprintf(buffer, 30, L"(%d, %d), (%d, %d)", currentShape.left, currentShape.top, currentShape.right, currentShape.bottom);

SendMessage(hwndList, LB\_INSERTSTRING, (WPARAM)0, (LPARAM)buffer);

currentShape = { 0, 0, 0, 0 };

}

if (isMove) {

isMove = false;

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

selectedIndex = shapes.size() - selectedIndex - 1;

int x = LOWORD(lParam) - startPos.X;

int y = HIWORD(lParam) - startPos.Y;

if (shapes[selectedIndex].selectedShape == 4)

{

for (long a = 0; a < shapes[selectedIndex].pen.size(); a++)

{

shapes[selectedIndex].pen[a].X += x;

shapes[selectedIndex].pen[a].Y += y;

}

RePaint(false, false);

}

else

{

shapes[selectedIndex].rect.left += x;

shapes[selectedIndex].rect.right += x;

shapes[selectedIndex].rect.top += y;

shapes[selectedIndex].rect.bottom += y;

RePaint(false, false);

}

}

SetFocus(hwndMain);

}

}

}

}

}

return CallNextHookEx(MouseHook, nCode, wParam, lParam);

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_SEARCH\_REG:

{

std::wstring message = L"Found ";

message += std::to\_wstring(emptyValues.size());

message += L" empty values.\nDelete?";

int result = MessageBox(NULL, message.c\_str(), L"Warning", MB\_YESNO);

if (result == IDYES)

{

HANDLE Delete = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)DeleteEmptyValues, NULL, 0, NULL);

if (Delete != NULL)

{

CloseHandle(Delete);

}

}

break;

}

case WM\_DELETE\_REG:

{

std::wstring message = L"Deleted ";

message += std::to\_wstring(deleteValues);

message += L" values out of ";

message += std::to\_wstring(emptyValues.size());

MessageBox(NULL, message.c\_str(), L"", MB\_ICONINFORMATION | MB\_OK);

break;

}

case WM\_CREATE:

{

#pragma region Elements

hwndComboBox = CreateWindow(L"COMBOBOX", NULL, WS\_CHILD | WS\_VISIBLE | CBS\_DROPDOWNLIST | WS\_VSCROLL, 0, 00, PW.x1, 200, hwnd, NULL, NULL, NULL);

SendMessage(hwndComboBox, CB\_ADDSTRING, 0, (LPARAM)L"Editing (E)");

SendMessage(hwndComboBox, CB\_ADDSTRING, 0, (LPARAM)L"Circle (A)");

SendMessage(hwndComboBox, CB\_ADDSTRING, 0, (LPARAM)L"Rectangle (B)");

SendMessage(hwndComboBox, CB\_ADDSTRING, 0, (LPARAM)L"Ellipse (C)");

SendMessage(hwndComboBox, CB\_ADDSTRING, 0, (LPARAM)L"Fill (D)");

SendMessage(hwndComboBox, CB\_SETCURSEL, 1, 0);

#pragma endregion

#pragma region SliderN

hSlider = CreateWindowEx(0, TRACKBAR\_CLASS, NULL, TBS\_AUTOTICKS | TBS\_ENABLESELRANGE | WS\_CHILD | WS\_VISIBLE, 0, 40, PW.x1, 40, hwnd, NULL, hInst, NULL);

SendMessage(hSlider, TBM\_SETRANGE, TRUE, MAKELPARAM(3, 20));

SendMessage(hSlider, WM\_SETREDRAW, FALSE, 0);

ShowWindow(hSlider, SW\_HIDE);

#pragma endregion

#pragma region Slider Thickness

hSliderThickness = CreateWindowEx(0, TRACKBAR\_CLASS, NULL, TBS\_AUTOTICKS | TBS\_ENABLESELRANGE | WS\_CHILD | WS\_VISIBLE, 0, 80, PW.x1, 40, hwnd, NULL, hInst, NULL);

SendMessage(hSliderThickness, TBM\_SETRANGE, TRUE, MAKELPARAM(0, 100));

#pragma endregion

#pragma region Color Choose Thickness

HWND hButton1 = CreateWindow(L"BUTTON", L"Color of Thickness", WS\_CHILD | WS\_VISIBLE, 0, 120, PW.x1, 40, hwnd, (HMENU)1001, GetModuleHandle(NULL), NULL);

ZeroMemory(&ccThickness, sizeof(ccThickness));

ccThickness.lStructSize = sizeof(ccThickness);

ccThickness.hwndOwner = hwndMain; // Parent window handle

ccThickness.lpCustColors = (LPDWORD)customColorsThickness;

ccThickness.rgbResult = RGB(255, 0, 0);

ccThickness.Flags = CC\_FULLOPEN | CC\_RGBINIT; // Color dialog options (full open and initial color)

#pragma endregion

#pragma region Color Choose Brush

HWND hButton2 = CreateWindow(L"BUTTON", L"Color of Fill", WS\_CHILD | WS\_VISIBLE, 0, 160, PW.x1, 40, hwnd, (HMENU)1002, GetModuleHandle(NULL), NULL);

ZeroMemory(&ccBrush, sizeof(ccBrush));

ccBrush.lStructSize = sizeof(ccBrush);

ccBrush.hwndOwner = hwndMain;

ccBrush.lpCustColors = (LPDWORD)customColorsThickness;

ccBrush.rgbResult = RGB(255, 0, 0);

ccBrush.Flags = CC\_FULLOPEN | CC\_RGBINIT;

#pragma endregion

#pragma region Slider Scale

hSliderScale = CreateWindowEx(0, TRACKBAR\_CLASS, NULL, TBS\_AUTOTICKS | TBS\_ENABLESELRANGE | WS\_CHILD | WS\_VISIBLE, 0, 200, PW.x1, 40, hwnd, NULL, hInst, NULL);

SendMessage(hSliderScale, TBM\_SETRANGE, TRUE, MAKELPARAM(1, 500));

#pragma endregion

#pragma region Slider Rotation

hSliderRotation = CreateWindowEx(0, TRACKBAR\_CLASS, NULL, TBS\_AUTOTICKS | TBS\_ENABLESELRANGE | WS\_CHILD | WS\_VISIBLE, 0, 240, PW.x1, 40, hwnd, NULL, hInst, NULL);

SendMessage(hSliderRotation, TBM\_SETRANGE, TRUE, MAKELPARAM(0, 360));

#pragma endregion

#pragma region Color Choose Brush

hwndList = CreateWindowEx(0, L"LISTBOX", NULL, WS\_CHILD | WS\_VISIBLE | WS\_BORDER | LBS\_NOTIFY | WS\_VSCROLL, 0, 280, PW.x1, 250, hwnd, (HMENU)100, GetModuleHandle(NULL), NULL);

hwndDeleteItem = CreateWindow(L"BUTTON", L"Delete", WS\_CHILD | WS\_VISIBLE, 0, 520, PW.x1 / 2, 40, hwnd, (HMENU)99, GetModuleHandle(NULL), NULL);

hwndDeleteItem = CreateWindow(L"BUTTON", L"Erase", WS\_CHILD | WS\_VISIBLE, PW.x1 / 2, 520, PW.x1 / 2, 40, hwnd, (HMENU)101, GetModuleHandle(NULL), NULL);

hwndUpItem = CreateWindow(L"BUTTON", L"Move Up", WS\_CHILD | WS\_VISIBLE, 0, 560, PW.x1 / 2, 40, hwnd, (HMENU)102, GetModuleHandle(NULL), NULL);

hwndDownItem = CreateWindow(L"BUTTON", L"Move Down", WS\_CHILD | WS\_VISIBLE, PW.x1 / 2, 560, PW.x1 / 2, 40, hwnd, (HMENU)103, GetModuleHandle(NULL), NULL);

#pragma endregion

#pragma region Open/Save

hwndSave = CreateWindow(L"BUTTON", L"Open File", WS\_CHILD | WS\_VISIBLE, 0, PW.y2 - 160, PW.x1, 40, hwnd, (HMENU)104, GetModuleHandle(NULL), NULL);

hwndSave = CreateWindow(L"BUTTON", L"Save File", WS\_CHILD | WS\_VISIBLE, 0, PW.y2 - 120, PW.x1, 40, hwnd, (HMENU)105, GetModuleHandle(NULL), NULL);

hwndSave = CreateWindow(L"BUTTON", L"Export to PNG", WS\_CHILD | WS\_VISIBLE, 0, PW.y2 - 80, PW.x1, 40, hwnd, (HMENU)106, GetModuleHandle(NULL), NULL);

#pragma endregion

#pragma region Select CP

hwndComboBoxCP = CreateWindow(L"COMBOBOX", NULL, WS\_CHILD | WS\_VISIBLE | CBS\_DROPDOWNLIST | WS\_VSCROLL, PW.x2, 0, 180, 200, hwnd, NULL, NULL, NULL);

hwndListCP = CreateWindowEx(0, L"LISTBOX", NULL, WS\_CHILD | WS\_VISIBLE | WS\_BORDER | LBS\_NOTIFY | WS\_VSCROLL, PW.x2, 500, 180, 220, hwnd, NULL, GetModuleHandle(NULL), NULL);

hSliderCP = CreateWindowEx(0, TRACKBAR\_CLASS, NULL, TBS\_AUTOTICKS | TBS\_ENABLESELRANGE | WS\_CHILD | WS\_VISIBLE, PW.x2, 50, 180, 40, hwnd, NULL, hInst, NULL);

SendMessage(hSliderCP, TBM\_SETRANGE, TRUE, MAKELPARAM(50, 2000));

SendMessage(hSliderCP, TBM\_SETPOS, TRUE, 1000);

SYSTEM\_INFO sysInfo;

GetSystemInfo(&sysInfo);

int numCores = sysInfo.dwNumberOfProcessors;

SendMessage(hwndComboBoxCP, CB\_ADDSTRING, 0, (LPARAM)L"\_Total");

for (int a = 0; a < sysInfo.dwNumberOfProcessors; a++)

{

SendMessage(hwndComboBoxCP, CB\_ADDSTRING, 0, (LPARAM)(L"0," + std::to\_wstring(a)).c\_str());

}

SendMessage(hwndComboBoxCP, CB\_SETCURSEL, 0, 0);

#pragma endregion

hButton6 = CreateWindow(L"BUTTON", L"Search", WS\_CHILD | WS\_VISIBLE, PW.x2, 720, 90, 40, hwnd, (HMENU)141, GetModuleHandle(NULL), NULL);

HANDLE hButton7 = CreateWindow(L"BUTTON", L"Semaphore", WS\_CHILD | WS\_VISIBLE, PW.x2 + 90, 720, 90, 40, hwnd, (HMENU)142, GetModuleHandle(NULL), NULL);

#pragma region Server

hwndListServer = CreateWindowEx(0, L"LISTBOX", NULL, WS\_CHILD | WS\_VISIBLE | WS\_BORDER | LBS\_NOTIFY | WS\_VSCROLL, PW.x2 + 200, 100, 180, 400, hwnd, NULL, GetModuleHandle(NULL), NULL);

hwndEditServer = CreateWindowEx(0, L"EDIT", NULL, WS\_CHILD | WS\_VISIBLE | ES\_AUTOHSCROLL | WS\_BORDER | LBS\_NOTIFY, PW.x2 + 200, 510, 180, 20, hwnd, NULL, GetModuleHandle(NULL), NULL);

SendMessage(hwndEditServer, EM\_SETLIMITTEXT, 200, 0);

hButtonSend = CreateWindow(L"BUTTON", L"Send", WS\_CHILD | WS\_VISIBLE, PW.x2 + 200, 540, 180, 40, hwnd, (HMENU)6601, GetModuleHandle(NULL), NULL);

hwndEditIP = CreateWindowEx(0, L"EDIT", NULL, WS\_CHILD | WS\_VISIBLE | ES\_AUTOHSCROLL | WS\_BORDER | LBS\_NOTIFY, PW.x2 + 200, 10, 180, 20, hwnd, NULL, GetModuleHandle(NULL), NULL);

hwndEditPort = CreateWindowEx(0, L"EDIT", NULL, WS\_CHILD | WS\_VISIBLE | ES\_AUTOHSCROLL | WS\_BORDER | LBS\_NOTIFY, PW.x2 + 200, 30, 180, 20, hwnd, NULL, GetModuleHandle(NULL), NULL);

SendMessage(hwndEditIP, EM\_SETLIMITTEXT, 20, 0);

SendMessage(hwndEditPort, EM\_SETLIMITTEXT, 10, 0);

hButtonPort = CreateWindow(L"BUTTON", L"Connect", WS\_CHILD | WS\_VISIBLE, PW.x2 + 200, 50, 180, 40, hwnd, (HMENU)6602, GetModuleHandle(NULL), NULL);

#pragma endregion

break;

}

case WM\_PAINT:

{

if (isDrawing)

{

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hwnd, &ps);

Gdiplus::Graphics graphics(hdc);

Gdiplus::Pen pen(Gdiplus::Color({ GetRValue(selectedColorThickness),GetGValue(selectedColorThickness),GetBValue(selectedColorThickness) }));

pen.SetWidth(Thickness);

Gdiplus::SolidBrush brush(Gdiplus::Color({ GetRValue(selectedColorBrush),GetGValue(selectedColorBrush),GetBValue(selectedColorBrush) }));

BitBlt(hdc, PW.x1, PW.y1, PW.Width, PW.Height, hdcBuffer, 0, 0, SRCCOPY);

DrawShape(graphics, &pen, &brush, GetKeyState(VK\_SHIFT) & 0x8000);

EndPaint(hwnd, &ps);

}

if (isTaskManager) {

isTaskManager = false;

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hwnd, &ps);

if (cpuLoadHistory.size() == 1)

{

RECT rectSlider2{ PW.x2,0,PW.x2 + 200,PW.y2 };

RECT rectSlider3{ PW.x2,100,PW.x2 + 200,300 };

RECT rectSlider4{ PW.x2,300,PW.x2 + 200,500 };

FillRect(hdc, &rectSlider2, (HBRUSH)(COLOR\_WINDOW));

FillRect(hdc, &rectSlider3, (HBRUSH)(COLOR\_WINDOW + 2));

FillRect(hdc, &rectSlider4, (HBRUSH)(COLOR\_WINDOW + 4));

}

SendMessage(hwndListCP, LB\_RESETCONTENT, 0, 0);

for (const std::wstring& item : StarusCP)

{

// Add an item to the list

SendMessage(hwndListCP, LB\_ADDSTRING, 0, reinterpret\_cast<LPARAM>(item.c\_str()));

}

DrawGraph(hdc, cpuLoadHistory, 300, RGB(255, 0, 0));

DrawGraph(hdc, memoryLoadHistory, 500, RGB(0, 0, 255));

EndPaint(hwnd, &ps);

}

break;

}

case WM\_LBUTTONDOWN:

{

if (LOWORD(lParam) >= PW.x1 && LOWORD(lParam) <= PW.x2 + 200 && selectedShape != 0) {

isDrawing = true;

currentShape.left = LOWORD(lParam);

currentShape.top = HIWORD(lParam);

currentShape.right = LOWORD(lParam);

currentShape.bottom = HIWORD(lParam);

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hwnd, &ps);

BitBlt(hdcBuffer, 0, 0, PW.Width, PW.Height, hdc, PW.x1, PW.y1, SRCCOPY);

EndPaint(hwnd, &ps);

if (selectedShape == 4) {

pen.push\_back({ LOWORD(lParam) ,HIWORD(lParam) });

}

InvalidateRect(hwnd, NULL, TRUE);

}

if (selectedShape == 0)

{

isMove = true;

startPos = { LOWORD(lParam), HIWORD(lParam) };

InvalidateRect(hwnd, NULL, TRUE);

}

break;

}

case WM\_MOUSEMOVE:

{

if (isDrawing)

{

currentShape.right = LOWORD(lParam);

currentShape.bottom = HIWORD(lParam);

if (selectedShape == 4) {

pen.push\_back({ LOWORD(lParam) ,HIWORD(lParam) });

}

InvalidateRect(hwnd, NULL, TRUE);

}

if (isMove)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

selectedIndex = shapes.size() - selectedIndex - 1;

int x = LOWORD(lParam) - startPos.X;

int y = HIWORD(lParam) - startPos.Y;

if (shapes[selectedIndex].selectedShape == 4)

{

std::vector<Gdiplus::Point> bufpoints = shapes[selectedIndex].pen;

for (long a = 0; a < shapes[selectedIndex].pen.size(); a++)

{

shapes[selectedIndex].pen[a].X += x;

shapes[selectedIndex].pen[a].Y += y;

}

RePaint(false, false);

shapes[selectedIndex].pen = bufpoints;

bufpoints.clear();

}

else

{

RECT bufrect = shapes[selectedIndex].rect;

shapes[selectedIndex].rect.left += x;

shapes[selectedIndex].rect.right += x;

shapes[selectedIndex].rect.top += y;

shapes[selectedIndex].rect.bottom += y;

RePaint(false, false);

shapes[selectedIndex].rect = bufrect;

}

}

}

break;

}

case WM\_LBUTTONUP:

{

if (isDrawing) {

isDrawing = false;

InvalidateRect(hwnd, NULL, TRUE);

if (selectedShape == 4) {

pen.push\_back({ LOWORD(lParam) ,HIWORD(lParam) });

}

shapes.push\_back({ currentShape, bool(GetKeyState(VK\_SHIFT) & 0x8000),n,selectedShape, Thickness,selectedColorThickness,selectedColorBrush,pen });

pen.clear();

wchar\_t buffer[30];

swprintf(buffer, 30, L"(%d, %d), (%d, %d)", currentShape.left, currentShape.top, currentShape.right, currentShape.bottom);

SendMessage(hwndList, LB\_INSERTSTRING, (WPARAM)0, (LPARAM)buffer);

currentShape = { 0, 0, 0, 0 };

}

if (isMove) {

isMove = false;

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

selectedIndex = shapes.size() - selectedIndex - 1;

int x = LOWORD(lParam) - startPos.X;

int y = HIWORD(lParam) - startPos.Y;

if (shapes[selectedIndex].selectedShape == 4)

{

for (long a = 0; a < shapes[selectedIndex].pen.size(); a++)

{

shapes[selectedIndex].pen[a].X += x;

shapes[selectedIndex].pen[a].Y += y;

}

RePaint(false, false);

}

else

{

shapes[selectedIndex].rect.left += x;

shapes[selectedIndex].rect.right += x;

shapes[selectedIndex].rect.top += y;

shapes[selectedIndex].rect.bottom += y;

RePaint(false, false);

}

}

SetFocus(hwndMain);

}

break;

}

case WM\_KEYDOWN:

{

int lower = tolower((unsigned char)wParam);

if (wParam == 'Z' && GetKeyState(VK\_CONTROL) < 0) {

RePaint(true, true);

SetFocus(hwnd);

}

else

{

switch (lower)

{

case L'a':

case L'c':

case L'd':

selectedShape = lower - L'a' + 1;

SendMessage(hwndComboBox, CB\_SETCURSEL, selectedShape, 0);

ShowWindow(hSlider, SW\_HIDE);

FillRectWindow();

break;

case L'b':

selectedShape = lower - L'a' + 1;

SendMessage(hwndComboBox, CB\_SETCURSEL, selectedShape, 0);

ShowWindow(hSlider, SW\_SHOW);

break;

case L'e':

selectedShape = 0;

SendMessage(hwndComboBox, CB\_SETCURSEL, selectedShape, 0);

ShowWindow(hSlider, SW\_HIDE);

FillRectWindow();

break;

}

}

break;

}

case WM\_COMMAND:

{

if (LOWORD(wParam) == 0 && HIWORD(wParam) == CBN\_SELCHANGE) {

int selectedIndex = SendMessage(hwndComboBox, CB\_GETCURSEL, 0, 0);

if (selectedIndex != CB\_ERR) {

if (selectedIndex == 2)

{

ShowWindow(hSlider, SW\_SHOW);

}

else

{

ShowWindow(hSlider, SW\_HIDE);

FillRectWindow();

}

selectedShape = selectedIndex;

}

SetFocus(hwnd);

}

if (LOWORD(wParam) == 1001)

{

if (ChooseColor(&ccThickness))

{

selectedColorThickness = ccThickness.rgbResult;

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].selectedColorThickness = ccThickness.rgbResult;

RePaint(false, false);

}

}

}

if (LOWORD(wParam) == 1002)

{

if (ChooseColor(&ccBrush))

{

selectedColorBrush = ccBrush.rgbResult;

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].selectedColorBrush = ccBrush.rgbResult;

RePaint(false, false);

}

}

}

if (LOWORD(wParam) == 100) {

if (HIWORD(wParam) == LBN\_SELCHANGE)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

SetFocus(hwndMain);

SendMessage(hwndList, LB\_SETSEL, selectedIndex, 0);

SendMessage(hSliderScale, TBM\_SETPOS, TRUE, shapes[shapes.size() - selectedIndex - 1].scale);

SendMessage(hSliderRotation, TBM\_SETPOS, TRUE, shapes[shapes.size() - selectedIndex - 1].rotation);

SendMessage(hSliderThickness, TBM\_SETPOS, TRUE, shapes[shapes.size() - selectedIndex - 1].Thickness);

SendMessage(hSlider, TBM\_SETPOS, TRUE, shapes[shapes.size() - selectedIndex - 1].n);

}

}

if (LOWORD(wParam) == 99)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

SetFocus(hwndMain);

SendMessage(hwndList, LB\_SETCURSEL, -1, 0);

}

else

{

MessageBox(NULL, L"The item was not added to the list!", L"Error", MB\_ICONERROR | MB\_OK);

}

}

if (LOWORD(wParam) == 101)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

RePaint(false, true);

}

else

{

MessageBox(NULL, L"The item was not added to the list!", L"Error", MB\_ICONERROR | MB\_OK);

}

SetFocus(hwndMain);

}

if (LOWORD(wParam) == 102)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

if (selectedIndex > 0)

{

Shape buf = shapes[shapes.size() - selectedIndex];

shapes[shapes.size() - selectedIndex] = shapes[shapes.size() - selectedIndex - 1];

shapes[shapes.size() - selectedIndex - 1] = buf;

wchar\_t buffer1[30];

wchar\_t buffer2[30];

SendMessage(hwndList, LB\_GETTEXT, selectedIndex - 1, (LPARAM)buffer1);

SendMessage(hwndList, LB\_GETTEXT, selectedIndex, (LPARAM)buffer2);

SendMessage(hwndList, LB\_DELETESTRING, selectedIndex - 1, 0);

SendMessage(hwndList, LB\_DELETESTRING, selectedIndex - 1, 0);

SendMessage(hwndList, LB\_INSERTSTRING, selectedIndex - 1, (LPARAM)buffer1);

SendMessage(hwndList, LB\_INSERTSTRING, selectedIndex - 1, (LPARAM)buffer2);

SendMessage(hwndList, LB\_SETCURSEL, selectedIndex - 1, 0);

RePaint(false, false);

}

}

else

{

MessageBox(NULL, L"The item was not added to the list!", L"Error", MB\_ICONERROR | MB\_OK);

}

SetFocus(hwndMain);

}

if (LOWORD(wParam) == 103)

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

if (selectedIndex < shapes.size() - 1)

{

Shape buf = shapes[shapes.size() - selectedIndex - 1];

shapes[shapes.size() - selectedIndex - 1] = shapes[shapes.size() - selectedIndex - 2];

shapes[shapes.size() - selectedIndex - 2] = buf;

wchar\_t buffer1[30];

wchar\_t buffer2[30];

SendMessage(hwndList, LB\_GETTEXT, selectedIndex + 1, (LPARAM)buffer1);

SendMessage(hwndList, LB\_GETTEXT, selectedIndex, (LPARAM)buffer2);

SendMessage(hwndList, LB\_DELETESTRING, selectedIndex, 0);

SendMessage(hwndList, LB\_DELETESTRING, selectedIndex, 0);

SendMessage(hwndList, LB\_INSERTSTRING, selectedIndex, (LPARAM)buffer2);

SendMessage(hwndList, LB\_INSERTSTRING, selectedIndex, (LPARAM)buffer1);

SendMessage(hwndList, LB\_SETCURSEL, selectedIndex + 1, 0);

RePaint(false, false);

}

}

else

{

MessageBox(NULL, L"The item was not added to the list!", L"Error", MB\_ICONERROR | MB\_OK);

}

SetFocus(hwndMain);

}

if (LOWORD(wParam) == 104)

{

// Saving the file for further processing in a separate thread

HANDLE fileHandle = CreateFile(L"shapes.dat", GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (fileHandle == INVALID\_HANDLE\_VALUE) {

MessageBox(NULL, L"Error 1: Unable to save the file for processing!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

// Get the file size

DWORD fileSize = GetFileSize(fileHandle, NULL);

// Create memory-mapped file for reading

HANDLE mapHandle = CreateFileMapping(fileHandle, NULL, PAGE\_READONLY, 0, fileSize, NULL);

if (mapHandle == NULL) {

CloseHandle(fileHandle);

MessageBox(NULL, L"Error 2: Failed to create memory-mapped file!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

CloseHandle(fileHandle);

// Map the file into memory for reading

LPVOID mapView = MapViewOfFile(mapHandle, FILE\_MAP\_READ, 0, 0, 0);

if (mapView == NULL) {

CloseHandle(mapHandle);

MessageBox(NULL, L"Error 3: Failed to map view of file!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

CloseHandle(mapHandle);

// Determine the number of shapes in the file

size\_t numShapes = fileSize / sizeof(Shape);

// Clear the vector if it contains any data

shapes.clear();

// Copy the data from the memory-mapped file to the vector

for (size\_t i = 0; i < numShapes; ++i) {

shapes.push\_back(\*(reinterpret\_cast<const Shape\*>(static\_cast<const char\*>(mapView) + i \* sizeof(Shape))));

}

// Unmap the memory-mapped file and trigger repaint

UnmapViewOfFile(mapView);

RePaint(false, false, true);

MessageBox(NULL, L"File loaded successfully!", L"Success", MB\_ICONINFORMATION | MB\_OK);

}

if (LOWORD(wParam) == 105)

{// Create memory-mapped file

HANDLE mapHandle = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, sizeof(Shape) \* shapes.size(), L"MyMappedFile");

if (mapHandle == NULL) {

MessageBox(NULL, L"Error 1: Failed to create memory-mapped file!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

// Map the file into memory for writing

LPVOID mapView = MapViewOfFile(mapHandle, FILE\_MAP\_WRITE, 0, 0, 0);

if (mapView == NULL) {

CloseHandle(mapHandle);

MessageBox(NULL, L"Error 2: Failed to map view of file!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

// Copy data from the vector to the memory-mapped file

memcpy(mapView, &shapes[0], sizeof(Shape) \* shapes.size());

// Unmap the memory-mapped file

UnmapViewOfFile(mapView);

CloseHandle(mapHandle);

// Create a file for writing the mapped data

HANDLE fileHandle = CreateFile(L"shapes.dat", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (fileHandle == INVALID\_HANDLE\_VALUE) {

MessageBox(NULL, L"Error 3: Failed to create file for writing!", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

// Write the data from memory-mapped file to the file

OVERLAPPED overlapped;

memset(&overlapped, 0, sizeof(OVERLAPPED));

DWORD bytesWritten;

WriteFile(fileHandle, mapView, sizeof(Shape) \* shapes.size(), NULL, &overlapped);

if (!GetOverlappedResult(fileHandle, &overlapped, &bytesWritten, TRUE)) {

MessageBox(NULL, L"Error 4: Failed to write data to file!", L"Error", MB\_ICONERROR | MB\_OK);

CloseHandle(fileHandle);

return 1;

}

MessageBox(NULL, L"Data successfully saved!", L"Success", MB\_ICONINFORMATION | MB\_OK);

CloseHandle(fileHandle);

}

if (LOWORD(wParam) == 106)

{

HDC screenDC = GetDC(hwndMain);

RECT windowRect;

GetWindowRect(hwndMain, &windowRect);

HBITMAP hBitmap = CreateCompatibleBitmap(screenDC, PW.Width - 16, PW.Height - 35);

HDC memDC = CreateCompatibleDC(screenDC);

SelectObject(memDC, hBitmap);

BitBlt(memDC, 0, 0, windowRect.right - windowRect.left + PW.x1, windowRect.bottom - windowRect.top, screenDC, PW.x1, PW.y1, SRCCOPY);

Gdiplus::Bitmap bitmap(hBitmap, NULL);

CLSID pngClsid;

GetEncoderClsid(L"image/png", &pngClsid);

bitmap.Save(L"paint.png", &pngClsid, NULL);

MessageBox(NULL, L"PNG saved!", L"Success", MB\_ICONINFORMATION | MB\_OK);

DeleteObject(hBitmap);

DeleteDC(memDC);

}

if (LOWORD(wParam) == 141) {

Search = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)StartSearch, NULL, 0, NULL);

}

if (LOWORD(wParam) == 142) {

HANDLE aThread[10];

DWORD ThreadID;

for (int i = 0; i < 10; i++)

{

aThread[i] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)SemaforTask, NULL, 0, &ThreadID);

}

}

if (LOWORD(wParam) == 6602) {

TCHAR buffer[15];

TCHAR buffer2[25];

GetWindowText(hwndEditIP, buffer2, sizeof(buffer2) / sizeof(TCHAR));

GetWindowText(hwndEditPort, buffer, sizeof(buffer) / sizeof(TCHAR));

std::wstring text(buffer);

int charSize = WideCharToMultiByte(CP\_ACP, 0, buffer2, -1, NULL, 0, NULL, NULL);

char\* charString = new char[charSize];

WideCharToMultiByte(CP\_ACP, 0, buffer2, -1, charString, charSize, NULL, NULL);

try {

closesocket(clientSocket);

int value = std::stoi(text);

clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (clientSocket == INVALID\_SOCKET) {

MessageBox(NULL, L"Failed to create socket", L"Error", MB\_OK | MB\_ICONERROR);

break;

}

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(value);

inet\_pton(AF\_INET, charString, &serverAddr.sin\_addr);

if (connect(clientSocket, reinterpret\_cast<sockaddr\*>(&serverAddr), sizeof(serverAddr)) == SOCKET\_ERROR) {

MessageBox(NULL, L"Failed to connect to server", L"Error", MB\_OK | MB\_ICONERROR);

closesocket(clientSocket);

break;

}

SendMessage(hwndListServer, LB\_RESETCONTENT, 0, 0);

MessageBox(NULL, L"Connected", L"", MB\_ICONINFORMATION | MB\_OK);

}

catch (std::exception const& e)

{

MessageBox(NULL, L"Incorrect port", L"Error", MB\_OK | MB\_ICONERROR);

}

}

if (LOWORD(wParam) == 6601) {

TCHAR buffer[256];

GetWindowText(hwndEditServer, buffer, sizeof(buffer) / sizeof(TCHAR));

int bufferSize = WideCharToMultiByte(CP\_UTF8, 0, buffer, -1, NULL, 0, NULL, NULL);

char\* utf8Buffer = new char[bufferSize];

WideCharToMultiByte(CP\_UTF8, 0, buffer, -1, utf8Buffer, bufferSize, NULL, NULL);

int bytesSent = send(clientSocket, utf8Buffer, bufferSize - 1, 0);

delete[] utf8Buffer;

if (bytesSent == SOCKET\_ERROR) {

MessageBox(NULL, L"Failed to send data to server", L"Error", MB\_OK | MB\_ICONERROR);

closesocket(clientSocket);

break;

}

}

break;

}

case WM\_HSCROLL:

{

int selectedIndex = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

if (lParam == (LPARAM)hSlider)

{

n = SendMessage(hSlider, TBM\_GETPOS, 0, 0); selectedColorThickness = ccThickness.rgbResult;

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].n = n;

RePaint(false, false);

}

}

if (lParam == (LPARAM)hSliderCP)

{

speed = SendMessage(hSliderCP, TBM\_GETPOS, 0, 0);

}

if (lParam == (LPARAM)hSliderThickness)

{

Thickness = SendMessage(hSliderThickness, TBM\_GETPOS, 0, 0);

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].Thickness = Thickness;

RePaint(false, false);

}

}

if (lParam == (LPARAM)hSliderScale) {

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].scale = SendMessage(hSliderScale, TBM\_GETPOS, 0, 0);

RePaint(false, false);

}

}

if (lParam == (LPARAM)hSliderRotation) {

if (selectedIndex != LB\_ERR)

{

shapes[shapes.size() - selectedIndex - 1].rotation = SendMessage(hSliderRotation, TBM\_GETPOS, 0, 0);

RePaint(false, false);

}

}

SetFocus(hwnd);

break;

}

case WM\_DESTROY:

{

PostQuitMessage(0);

break;

}

}

return DefWindowProc(hwnd, msg, wParam, lParam);

}

void FillRectWindow() {

HDC hdchwndMain = GetDC(hwndMain);

RECT rect1{ 0,0,PW.x1,PW.y2 };

FillRect(hdchwndMain, &rect1, (HBRUSH)(COLOR\_WINDOW));

ReleaseDC(hwndMain, hdchwndMain);

}

void RePaint(bool ctrlZ, bool del, bool list)

{

if (list) {

SendMessage(hwndList, LB\_RESETCONTENT, 0, 0);

for (int a = 0; a < shapes.size(); a++)

{

wchar\_t buffer[30];

swprintf(buffer, 30, L"(%d, %d), (%d, %d)", shapes[a].rect.left, shapes[a].rect.top, shapes[a].rect.right, shapes[a].rect.bottom);

SendMessage(hwndList, LB\_INSERTSTRING, (WPARAM)0, (LPARAM)buffer);

}

}

if (shapes.size() > 0)

{

int indexItem;

if (del) {

if (ctrlZ)

{

shapes.pop\_back();

indexItem = 0;

}

else

{

indexItem = SendMessage(hwndList, LB\_GETCURSEL, 0, 0);

shapes.erase(shapes.begin() + shapes.size() - indexItem - 1);

}

SendMessage(hwndList, LB\_DELETESTRING, indexItem, 0);

}

COLORREF bufSCT = selectedColorThickness;

COLORREF bufSCB = selectedColorBrush;

RECT bufCurrentShape = currentShape;

int bufSelectedShape = selectedShape;

int bufn = n;

std::vector<Gdiplus::Point> bufpen = pen;

HDC hdc = GetDC(hwndMain);

RECT rect1{ PW.x1, PW.y1, PW.x2, PW.y2, };

FillRect(hdc, &rect1, (HBRUSH)(COLOR\_WINDOW + 1));

for (int a = 0; a < shapes.size(); a++) {

n = shapes[a].n;

selectedShape = shapes[a].selectedShape;

pen = shapes[a].pen;

selectedColorThickness = shapes[a].selectedColorThickness;

selectedColorBrush = shapes[a].selectedColorBrush;

currentShape = shapes[a].rect;

Gdiplus::Graphics graphics(hdc);

Gdiplus::Pen pen(Gdiplus::Color({ GetRValue(selectedColorThickness),GetGValue(selectedColorThickness),GetBValue(selectedColorThickness) }));

pen.SetWidth(shapes[a].Thickness);

Gdiplus::SolidBrush brush(Gdiplus::Color({ GetRValue(selectedColorBrush),GetGValue(selectedColorBrush),GetBValue(selectedColorBrush) }));

DrawShape(graphics, &pen, &brush, shapes[a].isCorrect, shapes[a].scale, shapes[a].rotation);

}

ReleaseDC(hwndMain, hdc);

selectedColorThickness = bufSCT;

selectedColorBrush = bufSCB;

currentShape = bufCurrentShape;

selectedShape = bufSelectedShape;

n = bufn;

pen = bufpen;

bufpen.clear();

}

}

void DrawShape(Gdiplus::Graphics& graphics, Gdiplus::Pen\* penPlus, Gdiplus::SolidBrush\* brush, bool isCorrect, int scale, int rotation)

{

long x1 = currentShape.left;

long y1 = currentShape.top;

long x2 = currentShape.right;

long y2 = currentShape.bottom;

float s = scale / 100.0;

graphics.TranslateTransform((x2 + x1) / 2, (y2 + y1) / 2);

graphics.ScaleTransform(s, s);

graphics.RotateTransform(rotation);

graphics.TranslateTransform(-(x2 + x1) / 2, -(y2 + y1) / 2);

switch (selectedShape) {

// Êðóã

case 1:

{

if (isCorrect) {

int centerX = (x1 + x2) / 2;

int centerY = (y1 + y2) / 2;

int radius = (x2 - x1) / 2;

x1 = centerX - radius;

x2 = centerX + radius;

y1 = centerY - radius;

y2 = centerY + radius;

Gdiplus::Rect ellipseRect(x1, y1, x2 - x1, y2 - y1);

graphics.FillEllipse(brush, ellipseRect);

graphics.DrawEllipse(penPlus, ellipseRect);

}

else {

Gdiplus::Rect ellipseRect(x1, y1, x2 - x1, y2 - y1);

graphics.FillEllipse(brush, ellipseRect);

graphics.DrawEllipse(penPlus, ellipseRect);

}

break;

}

// N-óãîëüíèê

case 2: {

if (isCorrect)

{

double angle = 2 \* M\_PI / n;

int radius, x;

int y = y1;

std::vector<Gdiplus::Point> vertices;

if (x2 >= x1)

{

if (y2 >= y1)

{

radius = (x2 - x1) / 2;

}

else

{

radius = (x1 - x2) / 2;

}

x = x1 + (x2 - x1) / 2 - radius \* tan(M\_PI / n);

}

else

{

if (y2 >= y1)

{

radius = (x1 - x2) / 2;

}

else

{

radius = (x2 - x1) / 2;

}

x = x1 - abs(x2 - x1) / 2 - radius \* tan(M\_PI / n);

}

for (int i = 0; i < n; i++) {

vertices.push\_back({ x,y });

x += static\_cast<int>(radius \* 2 \* cos(angle \* i));

y += static\_cast<int>(radius \* 2 \* sin(angle \* i));

}

graphics.FillPolygon(brush, &vertices[0], vertices.size());

graphics.DrawPolygon(penPlus, &vertices[0], vertices.size());

}

else

{

int width = x2 - x1;

int height = y2 - y1;

int centerX = (x1 + x2) / 2;

int centerY = (y1 + y2) / 2;

double angle = 2 \* M\_PI / n;

std::vector<Gdiplus::Point> vertices;

for (int i = 0; i < n; i++)

{

int x = static\_cast<int>(centerX + width / 2 \* cos(i \* angle));

int y = static\_cast<int>(centerY + height / 2 \* sin(i \* angle));

vertices.push\_back({ x, y });

}

graphics.FillPolygon(brush, &vertices[0], vertices.size());

graphics.DrawPolygon(penPlus, &vertices[0], vertices.size());

}

break;

}

// Ïðÿìàÿ

case 3: {

Gdiplus::Point startPoint(x1, y1);

Gdiplus::Point endPoint(x2, y2);

graphics.DrawLine(penPlus, startPoint, endPoint);

break;

}

// Êàðàíäàø

case 4:

{

graphics.DrawCurve(penPlus, &pen[0], pen.size());

break;

}

}

FillRectWindow();

}

int GetEncoderClsid(const WCHAR\* format, CLSID\* pClsid)

{

UINT num = 0; // number of image encoders

UINT size = 0; // size of the image encoder array in bytes

Gdiplus::ImageCodecInfo\* pImageCodecInfo = NULL;

Gdiplus::GetImageEncodersSize(&num, &size);

if (size == 0)

return -1; // Failure

pImageCodecInfo = (Gdiplus::ImageCodecInfo\*)(malloc(size));

if (pImageCodecInfo == NULL)

return -1; // Failure

GetImageEncoders(num, size, pImageCodecInfo);

for (UINT j = 0; j < num; ++j)

{

if (wcscmp(pImageCodecInfo[j].MimeType, format) == 0)

{

\*pClsid = pImageCodecInfo[j].Clsid;

free(pImageCodecInfo);

return j; // Success

}

}

free(pImageCodecInfo);

return -1; // Failure

}

void UpdateData()

{

while (true)

{

int selectedIndex = SendMessage(hwndComboBoxCP, CB\_GETCURSEL, 0, 0);

int textLength = SendMessage(hwndComboBoxCP, CB\_GETLBTEXTLEN, selectedIndex, 0);

std::wstring buffer(textLength, L'2');

SendMessage(hwndComboBoxCP, CB\_GETLBTEXT, selectedIndex, (LPARAM)buffer.c\_str());

PDH\_STATUS status;

PDH\_HCOUNTER hCounter[10];

PDH\_HQUERY hQuery;

const LPCWSTR counterPaths[] = {

L"\\System\\Processes",

L"\\System\\Threads",

L"\\Process(\_Total)\\Handle Count",

L"\\System\\System Up Time",

L"\\Memory\\Available Bytes",

L"\\Memory\\Committed Bytes",

L"\\Memory\\Pool Nonpaged Bytes",

L"\\Memory\\Pool Paged Bytes",

};

status = PdhOpenQuery(NULL, 0, &hQuery);

status = PdhAddCounter(hQuery, (L"\\Processor Information(" + buffer + L")\\% Processor Utility").c\_str(), 0, &hCounter[0]);

for (int i = 0; i < 8; ++i)

{

status = PdhAddCounter(hQuery, counterPaths[i], NULL, &hCounter[i + 1]);

}

status = PdhCollectQueryData(hQuery);

std::this\_thread::sleep\_for(std::chrono::milliseconds(speed));

status = PdhCollectQueryData(hQuery);

PDH\_FMT\_COUNTERVALUE counterValue;

status = PdhGetFormattedCounterValue(hCounter[0], PDH\_FMT\_DOUBLE, NULL, &counterValue);

if (cpuLoadHistory.size() == 25)

{

cpuLoadHistory.clear();

memoryLoadHistory.clear();

}

cpuLoadHistory.push\_back(counterValue.doubleValue);

StarusCP.clear();

StarusCP.push\_back(L"CPU: " + std::to\_wstring(counterValue.doubleValue) + L"%");

for (int i = 1; i < 5; ++i)

{

status = PdhGetFormattedCounterValue(hCounter[i], PDH\_FMT\_LONG, NULL, &counterValue);

if (status == ERROR\_SUCCESS)

{

if (i == 1)

StarusCP.push\_back(L"Processes: " + std::to\_wstring(counterValue.longValue));

else if (i == 2)

StarusCP.push\_back(L"Threads: " + std::to\_wstring(counterValue.longValue));

else if (i == 3)

StarusCP.push\_back(L"Handles: " + std::to\_wstring(counterValue.longValue));

else if (i == 4)

StarusCP.push\_back(L"Up time: " + std::to\_wstring(counterValue.longValue));

}

}

StarusCP.push\_back(L"");

MEMORYSTATUSEX memInfo;

memInfo.dwLength = sizeof(MEMORYSTATUSEX);

GlobalMemoryStatusEx(&memInfo);

float memoryLoad = (float)(memInfo.ullTotalPhys - memInfo.ullAvailPhys) / (float)memInfo.ullTotalPhys \* 100.0f;

StarusCP.push\_back(L"Memory: " + std::to\_wstring(memoryLoad) + L"%");

StarusCP.push\_back(L"Total: " + std::to\_wstring(memInfo.ullTotalPhys / MYBYTES) + L" MB");

StarusCP.push\_back(L"In Use: " + std::to\_wstring((memInfo.ullTotalPhys - memInfo.ullAvailPhys) / MYBYTES) + L" MB");

for (int i = 5; i < 9; ++i)

{

status = PdhGetFormattedCounterValue(hCounter[i], PDH\_FMT\_DOUBLE, NULL, &counterValue);

if (status == ERROR\_SUCCESS)

{

if (i == 5)

StarusCP.push\_back(L"Available: " + std::to\_wstring(long long(counterValue.doubleValue / MYBYTES)));

else if (i == 6)

StarusCP.push\_back(L"Committed: " + std::to\_wstring(long long(counterValue.doubleValue / MYBYTES)));

else if (i == 7)

StarusCP.push\_back(L"Pool Nonpaged: " + std::to\_wstring(long long(counterValue.doubleValue / MYBYTES)));

else if (i == 8)

StarusCP.push\_back(L"Pool Paged: " + std::to\_wstring(long long(counterValue.doubleValue / MYBYTES)));

}

}

memoryLoadHistory.push\_back(memoryLoad);

PdhCloseQuery(hQuery);

isTaskManager = true;

InvalidateRect(hwndMain, NULL, TRUE);

}

}

void DrawGraph(HDC hdc, const std::vector<float>& data, int y, COLORREF color)

{

int width = 800;

int height = 200;

HBRUSH hBrush = CreateSolidBrush(color);

SelectObject(hdc, hBrush);

MoveToEx(hdc, 0, y, NULL);

int dataSize = data.size();

int step;

if (dataSize)

{

step = max(5, width / dataSize \* 5);

}

step = 5;

int x = (dataSize - 1) \* step + PW.x2;

int value = data[dataSize - 1];

int barHeight = min((int)((float)value / 100.0 \* height), height);

Rectangle(hdc, x, y, x + step, y - barHeight);

DeleteObject(hBrush);

}