*УДК 004.4’236*

**Графические приложения в c++**

**Автор Гегия Давит — студент.**

**Аннотация:** В статье предложены новые методы подхода к началу работы с графическими приложениями в целом и с фреймворками в частности

**Ключевые слова:** c++, графические приложения, фреймворк, графический интерфейс.

**Введение.**

**Актуальность темы.**

В последнее время использование графики для разнообразных задач становится всё более популярным. Графические приложения обеспечивают более простую и понятную связь с пользователем. С помощью графических приложений происходит визуализация разнообразных задач. Графические приложения также позволяют отслеживать любые действия пользователя по отношению к программе и к её графическим элементам. В то же время язык программирования С++ является крайне быстрым, высоко конфигурируемым и дает полноценный доступ к контролю памяти, что открывает возможности программирования в область микроконтроллеров и автоматизации базовых задач. Также благодаря поддержки фреймворков» и графических инструментариев, сильно уменьшается порог вхождения для решения тривиальных задач.

**Цель и задачи.**

Основная цель доклада – показать возможности графических инструментариев и фреймворков, работающих на языке программирования С++, разобрать подходы и методику разработки. Обрисовать основные принципы создания графических приложений, предоставить простые и понятные шаги, чтобы начать работу в разработке приложений. Объяснить сходства и различия, а также преимущества и недостатки двух фреймворков/графических инструментариев: wxWidgets и CLR (.Net framework).

**Основная идея.**

Основная идея работы заключается в том, чтобы на конкретных практических примерах показать сферы потенциального использования графических инструментариев, работающих на языке С++, описать вхождение в процесс разработки с самого начала, предложить уникальный подход к обучению и реализации всех предложенных возможностей на простых примерах и личном опыте.

**Краткое содержание.**

В докладе будут кратко рассмотрены три полноценные программы: две на инструментарии wxWidgets и одна на фреймворке и инструментарии .Net framework.

I. Калькулятор определителей матриц. Эта простая задача рассмотрена для того, чтобы показать базовые принципы и познакомить интересующихся с разработкой графических приложений в целом и с особенностями wxWidgets в частности.

Суть задачи: реализовать графический калькулятор определителей матриц размерности 2х2 и 3х3, начальное «меню» с выбором пользователем варианта, блоки для ввода пользователем с клавиатуры значений, кнопку, отвечающую за запуск решения, и блок для вывода результата.

II. «Скример». Цель – показать возможности инструментария wxWidgets в сфере обработки аудио- и визуального содержания. Раскрыть шире приёмы действий с пользователем.

Суть: создаётся простое окно стандартного приложения с белым квадратом по центру. Приложение реагирует на движение курсора мыши пользователя и при входе в зону квадрата или при выходе из окна приложения курсором запускает основную логику программы.

Основная логика программы состоит в том, что приложение разворачивается в полный экран, отключается возможность выхода из него пользователем, на экран проецируется соответствующая картинка, а на фоне постоянно воспроизводится звук. Выход из приложения возможен комбинацией клавиш Alt+F4.

III. Оскорбляющий калькулятор. Цель – показать возможности фреймворка .Net, описать его основные отличия от wxWidgets, выделить его положительные и отрицательные стороны. Организовать полноценное взаимодействие между графическими элементами и доступ к ним из логики программы, описать суть принципа «рычага» в графическом элементе «кнопка».

Суть: создать калькулятор, максимально похожий на встроенный системный калькулятор операционной системы Windows 10. Реализовать все соответствующие функции. Реализовать историю выполненных пользователем операций и взаимодействие с ними. Вкладка историй должна показываться и скрываться по нажатию кнопки. У калькулятора есть своя особенность: при нарушении арифметических правил, он начинает выводить оскорбления.

**Основная часть.**

**Графические приложения.**

На С++ с помощью специальных инструментариев и наборов библиотек можно создавать графические приложения, которые поддерживают взаимодействия с пользователем, формы, графические элементы, цвета и форматирование шрифтов.

Многие инструментарии позволяют сделать такие приложения ещё и кроссплатформенными – чтобы они запускались и нормально работали на самых разнообразных платформах.

Графические приложения – это одна из тех областей программирования, в которой можно спокойно применять фреймворки для создания необходимых элементов. Основной проблемой является придание свойств графическим элементам. Ведь нажатие кнопки должно к чему-то приводить, а текст на полотно должен каким-то образом выводится. И если с помощью любого фреймворка возможно с лёгкостью построить простой интерфейс, то вопрос о дальнейших шагах остаётся открытым.

Фреймворк — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Сразу же после установки всех файлов и создания первого проекта на wxWidgets мы обнаруживаем внутри простой фреймворк и пять различных файлов, в каждом из которых - более 50 строк. Причём при компиляции, на выходе мы получим простое окно (Рисунок. 1), в котором ничего нет.



Рисунок 1 Вид базового окна, сгенерированного фреймворком

**Калькулятор определителей матриц**

Как же не запутаться в это всём? Создаём пустой проект, внутри него два файла cApp.h и cApp.cpp, записываем внутрь несколько строчек и получаем практически такой же результат без всякого ненужного мусора.

Отдельное внимание хотелось бы выделить пустому деструктору. Дело в том, что все графические элементы wxWidgets удаляет самостоятельно, так что записывать по отдельности их в деструктор нет необходимости.

#ifndef CMAIN\_H

#define CMAIN\_H

#include "wx/wx.h"

#include "wx/listctrl.h"

#include <sstream>

#include <vector>

using namespace std;

class cMain : public wxFrame

{

public:

cMain();

~cMain();

};

#endif

ГРАФИЧЕСКИЕ ЭЛЕМЕНТЫ

WXWidgets включает в себя довольно обширный инструментарий: от банальных кнопок, блоков ввода и вывода текста, до различных графиков, таймеров, диодов, изображений, часов с календарём! Он умеет работать со звуком и экспортируемыми картинками. Из графических элементов в проекте используются лишь блоки ввода/вывода текста и кнопки. Для обширного спектра задач этого обычно бывает вполне достаточно.

Чем являются графические элементы по сути? Они есть ни что иное, как объекты собственного класса.

Что же делать с этими элементами дальше? Меню в программе создаётся просто: при запуске программы у нас появляется две кнопки «2х2» и «3х3», дающие пользователю выбрать размер матрицы. В зависимости от выбора создаём необходимые элементы, а эти кнопки просто скрываем.

cMain::cMain() : wxFrame(nullptr, 12, "Matrix", wxPoint(0,0), wxSize(800,600))

{

button1 = new wxButton(this, 1000, "3x3", wxPoint(100,100), wxSize(600,200));

button2 = new wxButton(this, 1001, "2x2", wxPoint(100,300), wxSize(600,200));

button3 = new wxButton(this, 1002, "SOLVE!", wxPoint(100,401), wxSize(300,100));

button3 -> Hide();

}

Всё довольно стандартно: инициализируем уже объявленные кнопки, в параметрах отправляя указатель, номер id (скоро пригодится), отображаемый текст, начальную позицию и размер.

Обращаем внимание на то, что кнопку button3 мы сразу же скрываем, она будет использоваться при любом выборе, но в меню, она нам совершенно точно не нужна!

Получаем следующий результат (Рисунок.2)



Рисунок 2 Стартовое меню программы

Поставили мы кнопки. При нажатии на них ничего не происходит, а должно!

Существует несколько способов передать действия пользователя (а это может быть не только нажатие кнопки, программа может реагировать на перемещение курсора, нажатие всяких кнопок на клавиатуре, двойное нажатие мышью, комбинации нажатия клавиш и многое другое), мы будем пользоваться очень простым способом – это так называемые «ивенты» (от англ. Events, события)

Для того, чтобы начать пользоваться событиями, мы для начала в заголовочном файле создаём функции типа void, в параметрах которых вписываем «command event». Существует несколько типов событий, command event сообщает о взаимодействии пользователя с внутренним графическим элементом, в данном случае с кнопкой. Ну и конечно же в самом конце мы сообщаем программе о том, что мы будем пользоваться ивентами.

Далее переходим к файлу cMain.cpp, в нём, мы ещё даже до инициализации конструктора сообщаем о начале таблицы событий, внутрь передаём класс, после этого описываем все необходимые нам события, в программке их получилось всего три, и сообщаем о конце таблицы событий (всё ещё до конструктора!)

Файл cMain.cpp

wxBEGIN\_EVENT\_TABLE(cMain, wxFrame)

EVT\_BUTTON(1000, OnButton1Clicked)

wxEND\_EVENT\_TABLE()

Сообщаем что это событие нажатия кнопки, и обращаем внимание на то, что в параметрах мы указываем то самое id кнопки, которое мы ей дали при инициализации, ну и конечно же название объявленной нами функции.

void cMain::OnButton2Clicked(wxCommandEvent &evt)

{

flag = 2;

text1 = new wxTextCtrl(this, 100, "", wxPoint(100,100), wxSize(100,100));

text2 = new wxTextCtrl(this, 101, "", wxPoint(100,200), wxSize(100,100));

text3 = new wxTextCtrl(this, 102, "", wxPoint(200,100), wxSize(100,100));

text4 = new wxTextCtrl(this, 103, "", wxPoint(200,200), wxSize(100,100));

text\_ctrl1 = new wxTextCtrl(this, 2000, wxEmptyString, wxPoint(500,150), wxSize(200,200),wxTE\_READONLY);

button1 -> Hide();

button2 -> Hide();

button3 -> Show();

}

Внутри функции мы задаём флаг, и только теперь инициализируем наши текстовые элементы – 4 или 9 штук соответственно для матрицы 2х2 или 3х3. sizer(). Не забываем, что wxWidgets вообще-то кроссплатформенный!

Ну и под конец реализуем нашу задумку, скрывая первые две кнопки и показывая кнопку “solve” общую для обоих выборов.

Логики самой программы в докладе нет, можно лишь сказать, что хранить значения в графических элементах и считывать их в дальнейшем довольно удобно.

Просто для понимания стоит отметить, что сам процесс вычисления реализован в пяти строках, и если бы писалось консольное приложение, то на тех же пяти строках и была бы финальная программа, однако со всеми графическими элементами и взаимодействием с пользователем ушло чуть более трёхсот строк.

**«Скример».**

Скример (от англ. «scream» — крик, он же «кричалка», «пугач», пугалка) — разновидность шок-контента, основанная на элементе неожиданности, призванная запугать зрителя до смерти. Представляет собой жестокий розыгрыш в Интернете, где ты сначала смотришь видео (или играешь в игру), а потом выскакивает ужастик.

Переходим к реализации самой программы, на самом деле от предыдущей она отличается не сильно: всё также вручную создаём окно, вручную указываем размеры и всё остальное.

public:

wxPanel \*pan = nullptr;

wxPanel \*pan1 = nullptr;

void OnWindowEnter(wxMouseEvent &evt);

void OnWindowLeave(wxMouseEvent &evt);

wxDECLARE\_EVENT\_TABLE();

Создаём две панели, и две функции, которые в параметрах имеют уже wxMouseEvent, потому что реагировать они будут уже не на действия с графическими элементами, а на движение курсора мыши. Ну и конечно же не забываем объявить о начале таблицы событий.

cMain::cMain() : wxFrame(nullptr, 12, "screamer", wxPoint(0,0), wxSize(600,600),wxSYSTEM\_MENU | wxCAPTION | wxCLIP\_CHILDREN)

{

pan = new wxPanel(this, 250,250,100,100,wxBORDER\_NONE);

pan1 = new wxPanel(this,0,0,1920,1080);

pan1->Hide();

}

Сверху всё та же структура наследования, только теперь во флагах спецификации окна не указываем кнопки закрытия и свёртывания, соответственно их у нас и не будет, а кнопка закрытия станет неактивной.

Характеризуем первую панель, которая и будет тем самым тёмно-серым квадратом, только указываем флаг «без границ». И вторая панель – наша «тёмная лошадка», которая будет скрываться до поры, ставим ей указатель в нулевые координаты и задаём размер полного экрана. (само окно у нас сразу открывается в размер 600х600).

Файл cMain.cpp

void cMain::OnWindowLeave(wxMouseEvent &evt)

{

wxSound("Untitled.wav").Play(wxSOUND\_ASYNC|wxSOUND\_LOOP);

wxClientDC dc(this->pan1);

pan1->Show();

int w,h;

this->SetSize(1920,1080);

wxMemoryDC mdc;

dc.GetSize(&w, &h);

wxImage img(wxT("scr.jpg"), wxBITMAP\_TYPE\_JPEG);

wxBitmap logo(img.Scale(w,h,wxIMAGE\_QUALITY\_HIGH));

mdc.SelectObject(logo);

dc.Blit(0,0, logo.GetWidth(), logo.GetHeight(), &mdc, 0, 0,wxCOPY, 0);

mdc.SelectObject(wxNullBitmap);

}

Первой же строкой начинаем проигрывать звук, исходник которого находится в папке с проектом, также в параметрах проигрывания указываем, что звук будет производится до тех пор, пока мы его не остановим системно, или не запустим другой звук.

Далее мы говорим программе, что сейчас будем на ней показывать изображение, а конкретно- на второй панели, которая на данный момент скрыта. В следующей строке показываем её и функцией SetSize() изменяем размер основного окна до стандартного разрешения. Выделяем память под изображение, говорим объекту dc, что отображать будем под размер окна.

**Калькулятор.**

Конечный вид программы показан на Рисунке 3.

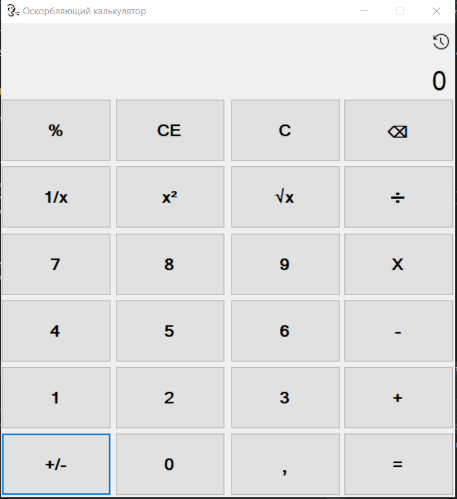


Рисунок 3 Конечный вид программы «Оскорбляющий калькулятор»

На Рисунке. 4 показан интерфейс .Net фреймворка.

Слева представлены все графические элементы, справа большой выбор настроек и свойств, чтобы полностью сконфигурировать вид элементов и даже некоторые элементы управления.

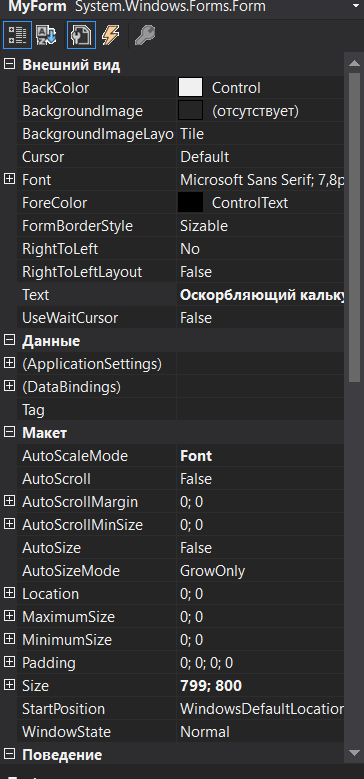
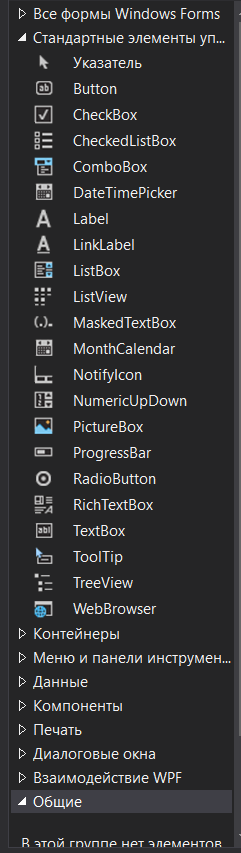


Рисунок 4 Интерфейс .Net framework

Именно тут мы начинаем знакомство с фреймворками, разобравшись во внутреннем устройстве графических приложений до того уровня, чтобы понимать принципы создания и взаимодействия между графическими элементами. Можно облегчить себе жизнь, не вписываю вручную положение, размер и свойства всех графических элементов.

Нужно отметить, что предварительно записанного «ужасного» кода .Net делает не так уж и много, по сравнению со своим коллегой wxWidgets. Вот например, что было сгенерировано фреймворком в файле MyForm.cpp

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

Application::EnableVisualStyles();

Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MyForm);

return 0;

Далее происходит «раздача» свойств этим графическим элементам, и, за что отдельная благодарность разработчикам, распределение свойств каждому элементу чётко разграничено и отдельно в комментариях упомянут сам элемент, с которым проходит работа.

//

// button1

//

this->button1->AutoSize = true;

this->button1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.15F, System::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point, static\_cast<System::Byte>(204)));

this->button1->Location = System::Drawing::Point(0, 651);

this->button1->Name = L"button1";

this->button1->Size = System::Drawing::Size(190, 100);

this->button1->TabIndex = 0;

this->button1->Text = L"+/-";

this->button1->UseVisualStyleBackColor = true;

this->button1->Click += gcnew System::EventHandler(this, &MyForm::button1\_Click);

Последним условием задачи была вкладка истории операций, которая так и называется (да-да можно создавать объекты на русском!)

По своей сути она является listbox-ом, у этого лист бокса автоматически создаётся свой вертикальный скроллбар, и лёгким кликом мыши в свойствах можно сделать аналогичный горизонтальный. Также он даёт пользователю возможность выделять отдельный элемент, либо несколько (если включить эту функцию в свойствах). Однако, к сожалению, он не умеет автоматически создавать события, поэтому, чтобы сделать вызов значения из истории операций, приходится вручную писать последнюю строчку, которая в отличии от кнопки будет реагировать только на двойной клик.

А ещё ниже мы работаем уже с самим окном приложения, устанавливаем размер, поочерёдно добавляем графические элементы, устанавливаем иконку приложения, отключаем свойства развёртывания и даём название.

На этом заканчивается практическая работа фреймворка. Ещё ниже идёт объявление событийных функций, хотя на самом деле система нам предлагает там же их и инициализировать, генерируя следующий код:

System::Void button1\_Click(System::Object^ sender, System::EventArgs^ e){}

Однако лёгким движением руки мы её превращаем в:

System::Void button1\_Click(System::Object^ sender, System::EventArgs^ e);

Ибо инициализации функций в заголовочном файле не место!

В заголовочном файле место объявлению переменных и функций, чем мы и пользуемся, объявляя в нём все необходимые элементы.

public: bool check = false;

bool check1 = false;

bool check2 = false;

char sym\_gl;

String^ last;

…

На этом и заканчивается наш заголовочный файл MyForm.h. И переходим мы уже к основному файлу Events.cpp. Замечу кстати, что файл Events.h у нас так и остался до конца пустым.

#include "Events.h"

#include "MyForm.h"

#include <iostream>

#include <cstring>

#include <msclr\marshal\_cppstd.h>

#include <fstream>

#include <math.h>

#include <stdlib.h>

#include <cstdlib>

#include <ctime>

using namespace Project1;

std::ofstream outfile("D://vs//projects//Project1//logs.txt");

Подключаем все необходимые библиотеки, не забывая конечно же подключить заголовочный файл, и создаём небольшой файлик в проекте, куда будем выводить логи.

Полностью логику разбирать мы, конечно же, не будем, можно лишь пробежаться по некоторым ключевым моментам.

Во-первых, главная идея реализуется через оператор множественного выбора switch и функции rand() и srand().

После указания типа и перед названием функций обязательно нужно указывать принадлежность к классу MyForm, т.к. инициализацию мы проводим в совершенно другом, не связанном с ним файле.

Вместо того, чтобы пытаться динамически читать ввод с кнопок или записывать их куда-то в файл, мы решаем эту задачу крайне просто. Все значения всё равно записываются в блоки (текст боксы), и для логики нам абсолютно не будет важно, стирал ли пользователь что-то в процессе ввода, изменял ли что-то или добавлял. Всё, что нам интересно, будет лежать в конечном варианте уже в том самом текстовом блоке. Таким образом мы наблюдаем довольно уникальную ситуацию, когда значения передаются уже из графического интерфейса прямо в функции, не находясь при этом в коде.

Почти все событийные функции занимают 5-6 строчек кода, исключение составляет лишь 0, так как для неё пришлось писать ограничения на месте.

Общий вид цифровых кнопок:

System::Void MyForm::button8\_Click(System::Object^ sender, System::EventArgs^ e)

{

label1->Text = ret(conv(System::Convert::ToString(label1->Text)), '1');

}

По сути, мы просто копируем ту самую функцию из MyForm.h, не забывая конечно же про MyForm::

Далее отправляем внутрь текст, для чего пользуемся функциями, которые сначала берут уже имеющийся текст и добавляют в него нужную нам цифру, иначе сделать не получится – метод Text просто заменяет текущий текст новым.

System::Void MyForm::button5\_Click(System::Object^ sender, System::EventArgs^ e)

{

label2->Text = ret(conv(System::Convert::ToString(label1->Text)), '+');

}

Далее идут 4 важнейших функции – арифметические операции.

Если вы обратили внимание, то номер лейбла изменился: теперь после определения действия текущее число переносится в верхний блок, туда же переносится и знак.

На Рисунке. 5 показан верхний левый угол программы с двумя блоками ввода и кнопкой истории операций.

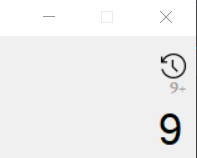


Рисунок 5 Верхний интерфейс программы: два блока и кнопка истории

Также исключением является кнопка ‘=‘, но она, как и остальные, вызывалась парой строк, ровно до тех пор, пока не была начата реализация истории операций.

System::Void MyForm::button3\_Click(System::Object^ sender, System::EventArgs^ e)

{

temp1 = System::Convert::ToString(label2->Text);

temp2 = list2 = System::Convert::ToString(label1->Text);

temp3 = fin;

fin = equals(conv(System::Convert::ToString(label2->Text)), conv(System::Convert::ToString(label1->Text)));

if (check2 == false)

{

list1 = temp1;

list2 = temp2;

}

else

{

list1 = temp3;

std::stringstream sss;

sss << conv(temp3) << sym\_gl;

std::string s1;

sss >> s1;

list1 = gcnew System::String(s1.c\_str());

}

label1->Text = fin;

label2->Text = "";

list(list1+list2+"="+fin);

}

Такой большой кусок кода нужен для того, чтобы отслеживать другую функцию.

И вот мы добрались до самого интересного. Во-первых, это реакция на двойное нажатие, само событие мы прописали ещё в файле MyForm.h. теперь осталось лишь указать, что конкретно оно будет делать. Сама реализация событийной функции выглядит следующим образом:

System::Void MyForm::lboxclicked(System::Object^ sender, System::EventArgs^ e)

{

label1->Text = ret2(System::Convert::ToString(История->SelectedItem));

}

Для начала мы вызываем метод у Истории selecteditem, который нам возвращает текст из конкретного элемента листбокса, и применяем его в качестве параметра в функции ret2.

String^ MyForm::ret2(String^ str)

{

if (ret(conv(str), 'e')=="")

return System::Convert::ToString(label1->Text);

return ret(conv(str), 'e');

}

На самом деле, это функция-матрёшка, мы внутри неё вызываем уже знакомую нам функцию ret, которая используется для всего, и даём в качестве аргумента специальный знак.

Вначале мы пытаемся предотвратить двойной клик по самому названию.

Зато именно здесь обнаружился очевидный минус называния функции русскими символами.

Если мы в первой строчке пропишем в условиях …==“История”, то функция не сработает: видя русскую раскладку она банально возвращает пустую строку. Казалось бы, хоть историю не вписывает в блок - и то хорошо, но основной блок ни при каких условиях не должен опустошаться – это может привести к непредвиденным последствиям. Поэтому мы и ставим проверку на пустую строку, раз уж именно её он нам возвращает, и в итоге, если пользователь будет делать двойное нажатие по Истории, то ему будет казаться, что ничего не происходит, однако каждый раз значение в основном лейбле просто заменяет само себя!

И закончить рассмотрение этого проекта мне бы хотелось на кнопке Истории.

С самой Историей мы не будем сильно заморачиваться, поэтому если сравнить интерфейс встроенного калькулятора и этого, то можно увидеть существенные различия, но основную свою идею мы выполнили.

Итак, для начала, через свойства или прямо в коде у этого блока объявляем скрытность:

this->История->Visible = false;

Это не является критическим, но, чтобы превратить кнопку Истории в «рычаг», или простой переключатель, можно будет обойтись без добавления новой булевой переменной, просто отслеживая состояние блока и меняя его после операций.

Ещё на этапе конструирования не выделяем место под историю, а делаем размер, как будто его там нет, но перед этим замеряя, каким примерно должен быть размер вместе с ней.

System::Void MyForm::button25\_Click(System::Object^ sender, System::EventArgs^ e)

{

if (this->История->Visible == false) {

this->ClientSize = System::Drawing::Size(750, 610);

this->История->Visible = true;

}

else

{

this->ClientSize = System::Drawing::Size(585, 610);

this->История->Visible = false;

}

}

Именно здесь пользуемся в условном операторе тем, что история у нас скрыта, и на основе этого изменяем под неё размер и не забываем сделать её видимой! После чего при следующем вызове этой функции она уйдёт в оператор else, где мы сворачиваем окно до первоначальных размеров и снова скрываем нашу историю.

**Заключение.**

В результате выполнения этой работы мною были подробно рассмотрены различные сферы использования графических инструментариев и фреймворков, для создания соответствующих приложений. Были рассмотрены принципы работы с аудио- и с графикой, отслеживание действий пользователя, форматирование текста, взаимодействие с цветовым спектром.

Были описаны базовые способы взаимодействия между графическими элементами, которые были применены на конкретных практических примерах, основу которых можно взять для создания собственных графических приложений на языке программирования C++.

**Список использованных источников**

1. Джулиан Смарт, Вадим Зеитлин, Робин Дунн, Стефан Ксомор, Брайан Петти, Франческо Монторси, Роберт Роблинг. Документация WxWidgets // <https://docs.wxwidgets.org/3.0/>
2. Джеф Просиз. Программирование для Microsoft .NET. - М.: Русская редакция, 2003. — С. 704.
3. Давит Гегия. Матричный калькулятор // <https://github.com/DavidaaWoW/wxWidgets_PJ_Matrix_calculator>
4. Давит Гегия. «Скример» // <https://github.com/DavidaaWoW/wxWidgets_PJ_Screamer>
5. Давит Гегия. «Оскорбляющий калькулятор» // <https://github.com/DavidaaWoW/DavidaaWoW-CLI_PJ_OffendingCalculator>
6. Википедия – свободная энциклопедия. Определение «фреймворка» // <https://ru.wikipedia.org/wiki/Фреймворк>
7. Wikireality, ВР. Определение «скримера» // <http://wikireality.ru/wiki/Скример>