**Программирование в САПР**

Лабораторная работа №13

***Тема занятия:***

*Создание проекта MFC SDI Application. Использование дополнительного вида для документа*

***Аннотация:***

Получение навыков создания MFC SDI Application на архитектуре «документ-вид» в интегрированной среде разработки Microsoft Visual Studio.

***План занятия:***

1. Подготовка к выполнению работы, в том числе:
   1. изучение структуры SDI MFC приложения с использованием архитектуры Document/View
   2. повторение ранее изученного материала по архитектуре MFC DialogBased Application.
2. Создание SDI MFC приложения на архитектуре «документ-вид» на языке C++ с несколькими видами одного документа
3. Изучение классов CView, СDocument, CFrameWnd, СTreeView
4. Защита лабораторной работы.

В предыдущей лабораторной работе мы научились создавать SDI приложение MFC и освоили основные принципы его внутренней организации и программные взаимосвязи его элементов. Мы узнали, что за графическое отображение данных документа отвечает класс «Вид». Однако, становится очевидным, что для полноценного графического отображения данных документа, в подавляющем большинстве случаев использование одного вида катастрофически недостаточно. Характерным примером может служить интерфейс практически любой САПР, где кроме отображения модели или чертежа всегда присутствует так называемое «дерево построения», отображающее внутреннюю структуру проектируемого изделия. Кроме этого также очевидно, что различные виды одного и того же документа должны иметь динамически изменяемые размеры области вывода, для обеспечения приоритетов в конфигурировании приложения под решение текущей задачи. Например, в программе почтовом клиенте часто можно балансировать рабочую область между отображением тела письма и списком полученных или отправленных писем, тем самым достигая концентрации внимания пользователя на отдельно взятом письме, либо на списке писем.

Итак приступим к добалению еще одного вида в наше SDI приложение. Как мы помним контейнером, содержащим вид документа является класс окна-рамки. Логично предположить, что это же окно должно поддерживать разделение на несколько областей для видов документа и динамическое изменение их размеров пользователем.

Давайте обратимся к заголовочному файлу MainFrm.h и разместим в области protected класса CMainFrame следующее объявление:

CSplitterWnd m\_wndSplitter;

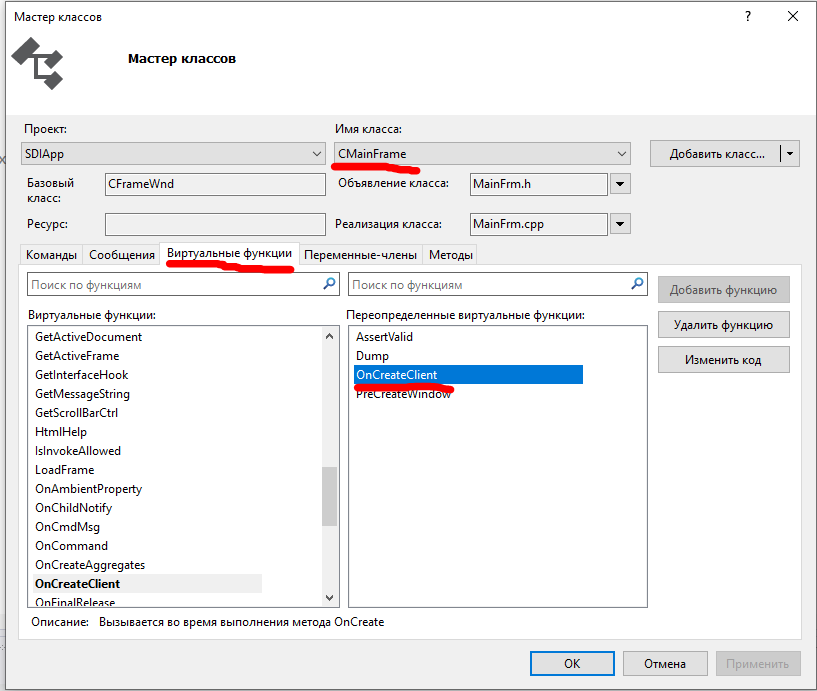


Рисунок 1. Мастер классов

Это экземпляр MFC класса «разделяемое окно», который будет отвечать деление клиентской области экрана. Сам факт разделения клиентской области экрана должен произойти в момент ее создания. Чтобы включиться в этот процесс в нужный момент мы должны реализовать для класса CMainFrame функцию OnCreateClient. Для этого воспользуемся мастером классов, выбрав в нем класс CMainFrame, раздел «Виртуальные функции». Добавим в созданную мастером приложений функцию OnCreateClient код создания сплиттера.

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

// TODO: добавьте специализированный код или вызов базового класса

m\_wndSplitter.CreateStatic(this, 1, 2);

return CFrameWnd::OnCreateClient(lpcs, pContext);

}

Метод *CreateStatic* принимает как первый параметр ссылку на родительское окно для сплиттера, которым в данном случае является текущий экземпляр класса CMainFrame, поэтому используется ключевое слово *this,* как ссылка на самого себя. Вторым и третьм параметрами метода *CreateStatic* являются количество колонок и рядов в создаваемом сплиттере. В нашем случае мы предполагаем разделить окно приложения по вертикали на две области: в правой области оставить исходный вид документа, а в левой создать древовидный список для отображения и настройки компонентов документа. Продолжим формировать сплиттер и добавим следующий код.

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

// TODO: добавьте специализированный код или вызов базового класса

m\_wndSplitter.CreateStatic(this, 1, 2);

m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CSDIAppView), CSize(400, 0), pContext);

m\_wndSplitter.CreateView(0, 1, RUNTIME\_CLASS(CSDIAppView), CSize(0, 0), pContext);

return CFrameWnd::OnCreateClient(lpcs, pContext);

}

Метод *CreateView* размещает в определенной области сплиттера вид документа. Область задается индексами ряда и колонки в первых двух параметрах метода. Индексация колонок и рядов начинается от нуля. Для того что бы указать какой именно вид мы хотим разместить в конкретной области сплиттера в третьем параметре метода *CreateView* указывается обрамленное в макрос RUNTIME\_CLASS имя класса размещаегомого вида. Поскольку на данный момент у нас отсутствует класс для второго вида, мы временно разместим *CSDIAppView* в обе области сплиттера. Четвертым параметром метода *CreateView* является начальный размер создавемого вида, причем в нашем случае значимым является только горизонтальный размер левого вида, поскольку правый вид займет все оставшееся от левого пространство по горизонтали и оба они займут все доступное место по вертикали.

Следуещее, что мы должны сделать – убрать вызов метода OnCreateClient родительского класса, заменив его оператором *return*.

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

// TODO: добавьте специализированный код или вызов базового класса

m\_wndSplitter.CreateStatic(this, 1, 2);

m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CSDIAppView), CSize(400, 0), pContext);

m\_wndSplitter.CreateView(0, 1, RUNTIME\_CLASS(CSDIAppView), CSize(0, 0), pContext);

return TRUE;

}

После чего мы наконец-то можем запустить приложение и визуально наблюдать созданный сплиттер. Также мы наблюдаем одинаковые виды в двух клиентских областях сплиттера, что естественно, поскольку мы использовали один класс для их заполнения.

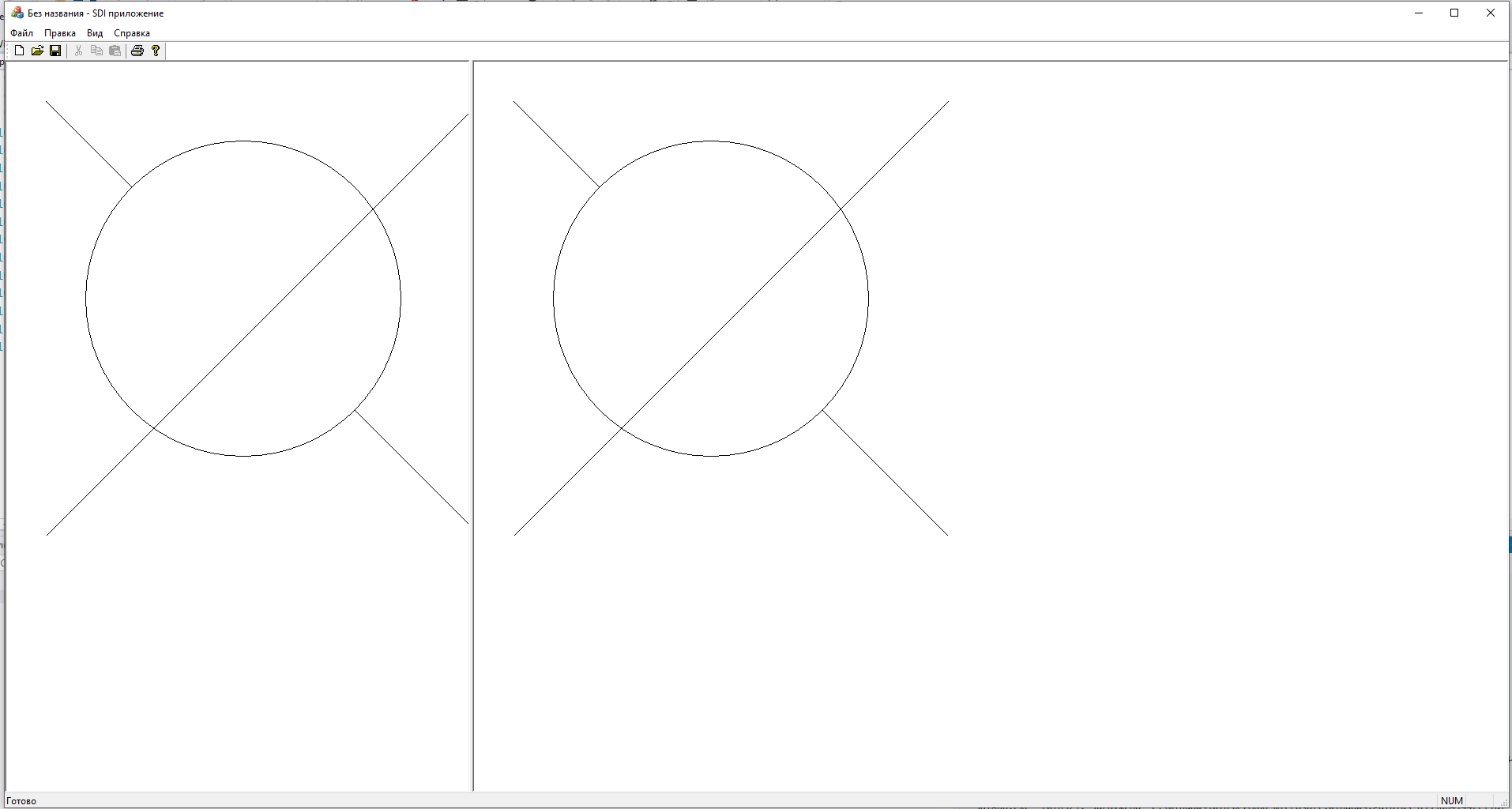
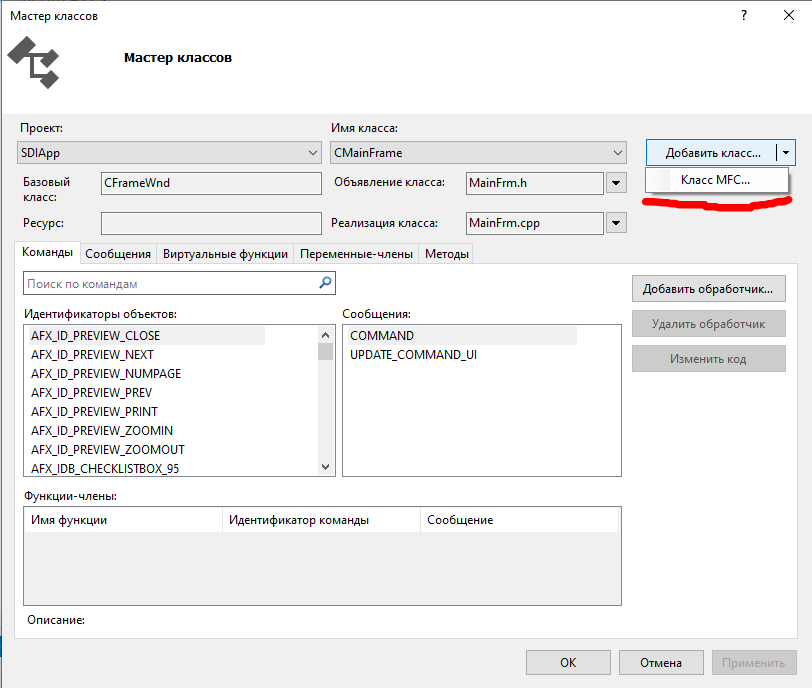


Рисунок 2. Динамически разделяемые области в окне-рамке

Теперь самое время заняться созданием класса для второго вида нашего документа. Для этого опять воспользуемся мастером классов и выберем позицию «Добавить класс MFC»



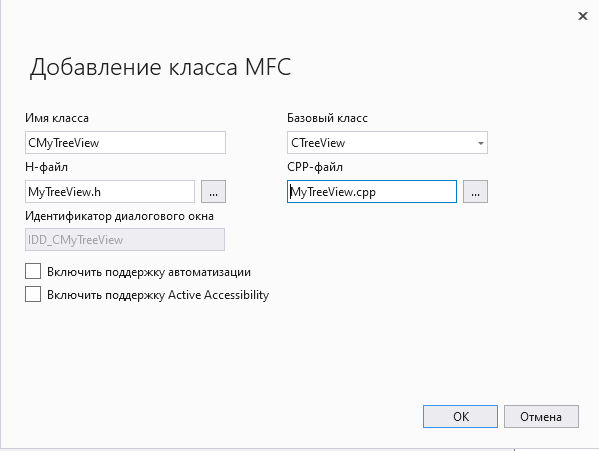


Рисунок 3. Создание класса

В последующем после выбора позиции «Добавить класс MFC» окне следует указать имя создаваемого класса (в нашем случае это *CMyTreeView*) и базовый класс. Базовый класс – это класс наследником которого станет создаваемый класс вида, а следовательно приобретет некоторую функциональность как говорят «из коробки».

После создания класса требуется включить в его заголовочный файл заголовочный файл класса-родителя *CTreeView* afxcview.h.

Теперь давайте заменим поставленную нами в начале «заглушку» для класса второго вида в методе *OnCreateClient* класса *CMainFrame.* После чего код метода будет выглядеть следующим образом:

Естественно предварительно нужно включить заголовочный файл класса *CMyTreeView* в файл реализации класса *CMainFrame.*

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

// TODO: добавьте специализированный код или вызов базового класса

m\_wndSplitter.CreateStatic(this, 1, 2);

m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CMyTreeView), CSize(400, 0), pContext);

m\_wndSplitter.CreateView(0, 1, RUNTIME\_CLASS(CSDIAppView), CSize(0, 0), pContext);

return TRUE;

}

На данном этапе в наш древовидный список уже существует но в нем пока нет ни одного элемента. Давайте приступим к его наполнению. Обычно в такого рода классах-списках организуют некоторый метод, который отвечает за поддержание актуального состояния списка. Давайте добавим такой метод с названием *FillTree* в наш класс. Задачей этого метода будет добавление в древовидный список корневого элемента «Линии» и дочерних от него элементов «Прямая1», «Окружность» и «Прямая2». Тем самым мы создадим интерфейс для управления отображением элементов в главном виде.

Каждый элемент деревовидного списка описывается типом *HTREEITEM,* поэтому чтобы иметь возможность постоянного доступа к содержимому дерева нам необходимо завести в классе *CMyTreeView* три свойства соответствующего типа для хранения ссылок на его элементы. В заголовочном файле класса *CMyTreeView* добавим следующее:

HTREEITEM m\_hLinesRoot, m\_hLine1, m\_hCircle, m\_hLine2;

void FillTree();

В файле реализации класса *CMyTreeView* разместим код метода *FillTree(*пока пустой*):*

void CMyTreeView::FillTree()

{

}

Теперь начнем писать содержательную часть метота *FillTree.* Первым делом мы должны получить доступ к контролу, который «зашит» в класс вида. В данном случае это объект типа *CTreeCtrl*. Получить ссылку на этот объект пожно воспользовавшись соответствующим методом класса *СTreeView*

void CMyTreeView::FillTree()

{

CTreeCtrl &tree = GetTreeCtrl();

}

Теперь с помощью переменной *tree* мы можем управлять содержимым нашего древовидного списка. Давайте добавим в него корневой элемент «Линии».

void CMyTreeView::FillTree()

{

CTreeCtrl &tree = GetTreeCtrl();

tree.DeleteAllItems();

m\_hLinesRoot = tree.InsertItem(L"Линии", -1, -1, NULL,TVI\_FIRST);

}

Мы использовали метод *InsertItem,* в параметры которого передается имя элемента списка, индекс картинки элемента в обычном и выделенном состоянии из набора битовых изображений, котрый может быть ассоциирован со списком путем вызова метода *SetImageList*, родительский элемент дерева и режим вставки в ветку дерева. Следует отметить, что метод *InsertItem* имеет и другие реализации. Результатирующий элемент дерева мы запоминаем в переменную *m\_hLinesRoot,* дабы в дальнейшем иметь возможность просто к нему обращаться. Добавим далее дочерние элементы «Прямая1», «Окружность» и «Прямая2» для только что добавленного корневого.

void CMyTreeView::FillTree()

{

CTreeCtrl &tree = GetTreeCtrl();

tree.DeleteAllItems();

m\_hLinesRoot = tree.InsertItem(L"Линии", -1, -1, NULL,TVI\_FIRST);

m\_hLine1 = tree.InsertItem(L"Прямая1", -1, -1, m\_hLinesRoot, TVI\_FIRST);

m\_hLine2 = tree.InsertItem(L"Прямая2", -1, -1, m\_hLinesRoot, TVI\_FIRST);

m\_hCircle = tree.InsertItem(L"Окружность", -1, -1, m\_hLinesRoot, TVI\_FIRST);

}

Процесс добавления дочерних элементов в целом аналогичен, с той лишь разницей, что в параметрах метода *InsertItem* мы указываем *m\_hLinesRoot* в качестве родительского элемента.

Теперь нам конечно хотелось бы посмотреть на результат своего труда и увидеть созданный древовоидный список на экране в окне нашего приложения, однако, запустив программу на данном этапе мы ничего не увидим. Происходит это потому что метод *FillTree* хоть нами и реалирован, но пока не вызывается нигде в нашем коде.

Давайте посмотрим немного шире на эту проблему. На данный момент ни документ ни тем более основной его вид понятия не имеют, что у нас в программе появился дополнительный вид документа. Очевидно, что нужно установить связи между этими объектами, дабы они могли управлять друг другом и получать нужную информацию друг от друга. Наиболее удобным методом «установления связей» является передача указателей на соответсятвующие объекты. То есть в классе документа нам нужно разместить указатель на дерево, в дереве указатель на документ, и потом проделать тоже самое в паре видов нашего документа или передавать данные опосредованно через класс документ.

Итак, добавим в заголовочный файл класса документа указатель на дерево:

CMyTreeView \*m\_pTreeView;

Это потребует включения заголоваочного файла класса дерева или использования приема *forward declaration*. Пойдем вторым путем и в заголовочном файле документа применим *forward declaration* к классу *CMyTreeView*

class CMyTreeView;

Аналогично сделаем указатель на документ и *forward declaration* его класса для класса дерева.

class CSDIAppDoc;

CSDIAppDoc\* m\_pDoc;

Осталось инициализировать заведенные указатели. Давайте вернемся в метод *OnCreateClient* класса *CMainFrame* и попробуем инициализировать указатели с использованием информации, хранящейся в сплиттере.

Для этого вызовем метод GetPane, который возвращает CWnd указатель на область сплиттера в которой расположен вид.

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)

{

// TODO: добавьте специализированный код или вызов базового класса

m\_wndSplitter.CreateStatic(this, 1, 2);

m\_wndSplitter.CreateView(0, 0, RUNTIME\_CLASS(CMyTreeView), CSize(400, 0), pContext);

m\_wndSplitter.CreateView(0, 1, RUNTIME\_CLASS(CSDIAppView), CSize(0, 0), pContext);

SetActiveView((CView\*)m\_wndSplitter.GetPane(0, 1));

CSDIAppDoc \*pDoc = (CSDIAppDoc\*)GetActiveDocument();

pDoc->m\_pTreeView = (CMyTreeView\*)m\_wndSplitter.GetPane(0, 0);

pDoc->m\_pView = (CSDIAppView\*)m\_wndSplitter.GetPane(0, 1);

pDoc->m\_pTreeView->m\_pDoc = pDoc;

return TRUE;

}

Вызов метода *SetActiveView* обусловлен тем, что мы должны быть уверены в том, что на момент вызозова метода *GetActiveDocument* активным будет именно основной вид. Также следует обратить внимание на то, что в данном коде несколько раз используется механизм привидения типа указателя к необходимому. Например, указатель типа *CDocument* который возвращает метод *GetActiveDocument* может быть безболезненно преобразован к типу CSDIAppDoc так как экземпляром именно этого класса описывается документ в нашем приложении и ничего другого за этим указателем скрываться не может.

Данный код отлично сработает при запуске приложения, но что будет если в приложении произойдет смена документа? Очевидно, связи нарушатся. Будем выходить из положения путем редакции метода *OnNewDocument* класса *CSDIAppDoc:*

BOOL CSDIAppDoc::OnNewDocument()

{

if (!CDocument::OnNewDocument())

return FALSE;

CWnd\* MainWnd = AfxGetMainWnd();

if (MainWnd)

{

m\_pTreeView = (CMyTreeView\*)((CMainFrame\*)MainWnd)->m\_wndSplitter.GetPane(0, 0);

m\_pTreeView->m\_pDoc = this;

m\_pView = (CSDIAppView\*)((CMainFrame\*)MainWnd)->m\_wndSplitter.GetPane(0, 1);

}

m\_pTreeView->FillTree();

return TRUE;

}

В данном коде мы сначала проверяем на валидность указатель на главное окно приложения. Если указатель валиден, значит это не запуск приложения а открытие очередного документа и можно через указатель на главное получить доступ к сплиттеру и его содержимому, инициализировав все что необходимо. Если же указатель на главное окно приложения нулевой, то значит приложение только стартует и все первичные инициализации произойдут в *CMainFrame::OnCreateClient*. Метод *FillTree* класса древовидного списка вызывается в любом случае.

Теперь, запустив приложение, мы наконец-то можем увидеть наш древовидный список на экране, в отведенном ему месте.

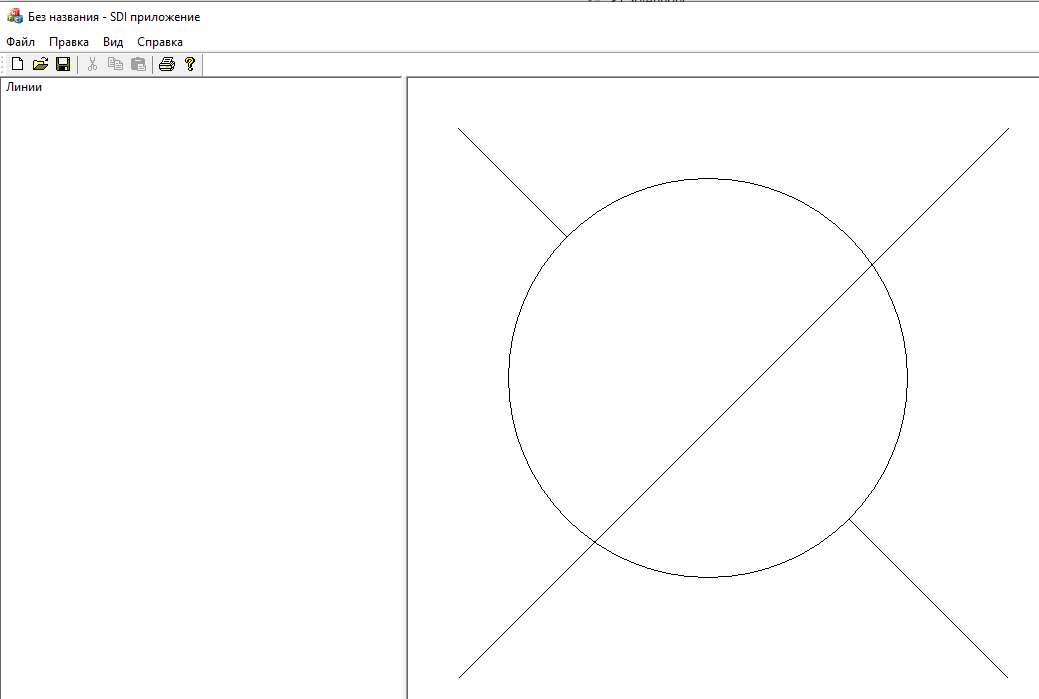


Рисунок 4. Окно с древовидным списком

Однако, мы видим только корневой элемент «Линии», поскольку он находится в свернутом состоянии и его дочерние элементы не отображаются. Давайте модифицируем код метода FillTree так, чтобы дерево появлялось сразу в развернутом состоянии.

void CMyTreeView::FillTree()

{

CTreeCtrl &tree = GetTreeCtrl();

tree.DeleteAllItems();

m\_hLinesRoot = tree.InsertItem(L"Линии", -1, -1, NULL,TVI\_FIRST);

m\_hLine1 = tree.InsertItem(L"Прямая1", -1, -1, m\_hLinesRoot, TVI\_FIRST);

m\_hLine2 = tree.InsertItem(L"Прямая2", -1, -1, m\_hLinesRoot, TVI\_FIRST);

m\_hCircle = tree.InsertItem(L"Окружность", -1, -1, m\_hLinesRoot, TVI\_FIRST);

tree.Expand(m\_hLinesRoot, TVE\_EXPAND);

}

Теперь все элементы дерева видны.

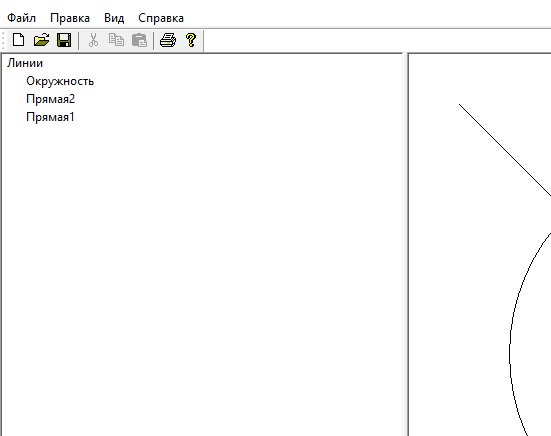


Рисунок 5. Корневой элемент в развернутом состоянии

Теперь обратим внимание на внешний вид нашего древовидного списка. Он далек от привычного не правда ли? Давайте исправим ситуацию. Для этого нам придется реализовать метод *OnCreate* для класса *CMyTreeView.* Делаем это опять же через мастер классов, добавляя обработчик на сообщение WM\_CREATE.

int CMyTreeView::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

if (CTreeView::OnCreate(lpCreateStruct) == -1)

return -1;

return 0;

}

Изменяя параметр *style* структуры *lpCreateStruct* мы можем модифицировать стиль отображения древовидного списка. Допустим нам необходимы линии между элементами, чек-бокс у каждого элемента и знаки «плюс –минус» для разворачивания и сворачивания корневого элемента дерева. Изменим параметры следующим образом:

int CMyTreeView::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

lpCreateStruct->style |= TVS\_HASLINES | TVS\_HASBUTTONS | TVS\_LINESATROOT | TVS\_SHOWSELALWAYS | TVS\_CHECKBOXES;

if (CTreeView::OnCreate(lpCreateStruct) == -1)

return -1;

return 0;

}

Итеперь дерево выглядит более презентабельно.

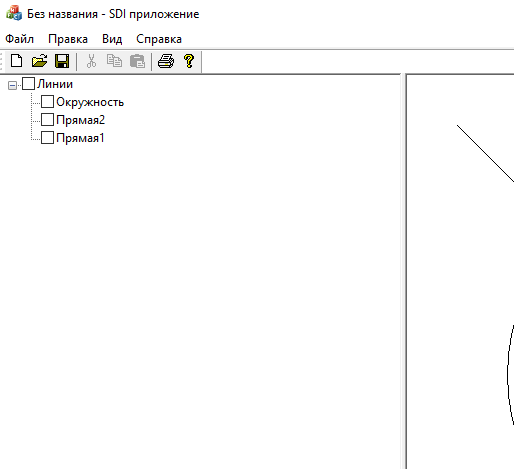


Рисунок 5. Модификация внешнего вида списка

Теперь у нас есть полный инструментарий для управления отображением графических примитивов в основном виде. Давайте разместим в классе документа три логические переменные, отвечающие за вывод каждого из примитивов на экран.

bool m\_bLine1, m\_bLine2, m\_bCircle;

А также модифицируем метод OnNewDocument установив начальные значения данных переменных.

BOOL CSDIAppDoc::OnNewDocument()

{

if (!CDocument::OnNewDocument())

return FALSE;

CWnd\* MainWnd = AfxGetMainWnd();

if (MainWnd)

{

m\_pTreeView = (CMyTreeView\*)((CMainFrame\*)MainWnd)->m\_wndSplitter.GetPane(0, 0);

m\_pView = (CSDIAppView\*)((CMainFrame\*)MainWnd)->m\_wndSplitter.GetPane(0, 1);

m\_pTreeView->m\_pDoc = this;

}

m\_pTreeView->FillTree();

m\_bLine1 = m\_bLine2 = m\_bCircle = false;

return TRUE;

}

Так же измерим код метода *OnDraw* в классе основного вида и привяжем вывод линий к переменным в классе документа

void CSDIAppView::OnDraw(CDC\* pDC)

{

CSDIAppDoc\* pDoc = GetDocument();

ASSERT\_VALID(pDoc);

if (!pDoc)

return;

if (pDoc->m\_bLine1)

{

pDC->MoveTo(50, 50);

pDC->LineTo(600, 600);

}

if (pDoc->m\_bCircle)

{

pDC->Ellipse(100, 100, 500, 500);

}

if (pDoc->m\_bLine2)

{

pDC->MoveTo(600, 50);

pDC->LineTo(50, 600);

}

}

Теперь привяжем изменение состояния переменных документта, отвечающих за вывод линий к действиям пользователя в древовидном списке. Если галочка напротив элемента установлена – он выводится на экран, в противном случае – нет. Галочки на всех элементах разом можно снять или установить кликая на корневом элементе дерева. Для этого реализуем в классе *CMyTreeView* метод *OnLButtonDown* опять же воспользовавшись мастером классов и создав обработчик сообщения WM\_LBTNDOWN

После получения ссылки на контрол, ассоциированный с видом мы вызываем метод базового класса для того чтобы он выполнил все необходимые действия относящиеся к базовому функционалу, например нарисовал галочку у элемента по которому кликнули. Далее мы отсекаем реакцию на клики по пустому полю дерева через проверку попадания точки клика *point* в прямоугольник, ограничивающий элемент дерева и в случае если клик пришелся на элемент, выделяем его методом *SelectItem.* Потом проверям не является ли выделенный элемент корневым и если является приводим состояние дочерних элементов в соответствие с корневым, в противном случае принудительно снимаем с него галочку. Далее устанавливаем значения переменных в документе в соответствии с полодением дел в дереве и вызываем принудительную перерисовку основного вида.

void CMyTreeView::OnLButtonDown(UINT nFlags, CPoint point)

{

// TODO: добавьте свой код обработчика сообщений или вызов стандартного

CTreeCtrl& tree = GetTreeCtrl();

CTreeView::OnLButtonDown(nFlags, point);

CRect rc;

tree.GetItemRect(m\_hLinesRoot, &rc,false);

if (rc.PtInRect(point))

tree.SelectItem(m\_hLinesRoot);

tree.GetItemRect(m\_hLine1, &rc,false);

if (rc.PtInRect(point))

tree.SelectItem(m\_hLine1);

tree.GetItemRect(m\_hLine2, &rc, false);

if (rc.PtInRect(point))

tree.SelectItem(m\_hLine2);

tree.GetItemRect(m\_hCircle, &rc, false);

if (rc.PtInRect(point))

tree.SelectItem(m\_hCircle);

if (tree.GetSelectedItem() == m\_hLinesRoot)

{

bool check = tree.GetCheck(m\_hLinesRoot);

tree.SetCheck(m\_hLine1, check);

tree.SetCheck(m\_hLine2, check);

tree.SetCheck(m\_hCircle, check);

}

else

tree.SetCheck(m\_hLinesRoot, false);

m\_pDoc->m\_bLine1 = tree.GetCheck(m\_hLine1);

m\_pDoc->m\_bLine2 = tree.GetCheck(m\_hLine2);

m\_pDoc->m\_bCircle = tree.GetCheck(m\_hCircle);

m\_pDoc->m\_pView->Invalidate();

}

На этом мы наконец достигли своей цели – графические элементы в главном виде подчиняются выбору пользователя в дополнительном виде древовидного списка, а взаимодействие видов осуществляется посредством изменения данных хранящихся в классе документа.

***Задание для самостоятельной работы***

Создать проект MFC SDI-приложения в Microsoft Visual Studio в соответствии с настройками рассмотренными в данной работе. Модифицировать класс документа и класс основного вида для отрисовки десяти прямых линий, десяти окружностей. Древовидный список сделать многоуровневым с корневым элементом «Линии», дочерними к нему элементами «Прямые» и «Окружности», в которых содержатся списки соответствующих графических примитивов отрисовкой которых можно управлять по аналогии с тем как было показано в данной работе.

***Форма представления выполненного задания***

На проверку преподавателю требуется представить архив (RAR, ZIP) с проектом MS Visual Studio, реализующим задание для самостоятельной работы.