Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Оренбургский государственный университет»

А.И. Сергеев

РАЗРАБОТКА ПРИКЛАДНЫХ МОДУЛЕЙ ДЛЯ СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ «SIEMENS NX»

Учебное пособие

Рекомендовано ученым советом федерального государственного бюджетного образовательного учреждения высшего образования «Оренбургский государственный университет» для обучающихся по образовательным программам высшего образования по направлению подготовки 09.04.01 Информатика и вычислительная техника.

Оренбург
2021

УДК 658.512.26:004.925.8
ББК 30.2-5-05
С 32

Рецензент – профессор, доктор технических наук А.Н. Поляков

Сергеев, А.И.
С 32 Разработка прикладных модулей для системы автоматизированного проектирования «Siemens NX» : учебное пособие / А.И. Сергеев ; Оренбургский гос. ун-т. – Оренбург : ОГУ. – 2021. – 137 с.
ISBN

Учебное пособие содержит теоретические сведения о разработке прикладных программных модулей для системы автоматизированного проектирования «Siemens NX». Представлен материал о настройке среды программирования, об основных функциях Open NX. В пособие включены задания для выполнения практических работ. Для самоподготовки по каждой теме приводятся контрольные вопросы.

Учебное пособие предназначено для обучающихся по образовательным программам высшего образования по направлению подготовки 09.04.01 Информатика и вычислительная техника. Издание может быть полезно и для обучающихся других направлений подготовки при изучении дисциплин, связанных с разработкой подсистем автоматизированного проектирования.

УДК 658.512.26:004.925.8
ББК 30.2-5-05

ISBN

Содержание

Введение

Каждое предприятие сталкивается со специфическими задачами при проектировании изделий, поэтому почти во всех САПР присутствует механизм, с помощью которого пользователь может разрабатывать собственные встраиваемые в систему прикладные программные модули, решающие специализированные отраслевые задачи. К таким механизмам в NX относятся программирование с применением Open API NX.

NX Open API – это набор инструментов и технологий, посредством которых внешнее приложение может получить доступ к возможностям NX. NX Open API позволяет программным способом на основании рассчитанных параметров проектировать детали и сборки, а также выпускать документацию. Очень часто требуется интегрировать NX с различными специализированными приложениями, предназначенными для проведения расчетов. При этом геометрия, сформированная в NX, может выступать в качестве исходной информации для выполнения анализа, или, наоборот, NX возьмет на себя функции отображения сгенерированной приложением геометрии в модельном пространстве. Практически все возможности NX доступны с помощью NX Open API, однако имеется целый класс объектов, создание которых возможно только программным способом.

Использование функциональности Open API открывает перед пользователем расширенные возможности автоматизации моделирования и обработки 2D и 3D объектов благодаря набору библиотек и подпрограмм, позволяющих внешнему (независимому) или внутреннему (интегрированному) приложению создавать модели в NX и получать доступ к их объектам для совершения различных операций.

Материал пособия основан на издании Тихомирова В.А. «Разработка приложений для NX на языке C» [1].

1 Подготовка среды разработки программных модулей NX на базе библиотек Open API

1.1 Установка и настройка среды компиляции проектов NX

Для доступа к трехмерной модели в среде NX и построения различных автоматизированных систем, управляющих, как самими моделями, так и операциями их изменения, предусмотрен механизм разработки прикладных программных модулей на языке Си с использованием библиотек встроенных функций NX Open API.

Прикладной модуль может быть внешнего типа и выполнен, как отдельное приложение в формате EXE файла, или внутреннего типа - DLL файл, который запускается из запущенной CAD-системы NX. Инструментарием для создания таких модулей рекомендуется использовать MS Visual Studio.

Возможны три пути установки и настройки среды Microsoft Visual Studio для компиляции проектов внутренних и внешних модулей для NX:

1 настройка, когда CAD система NX устанавливается после того, как установлена система MS Visual Studio;

2 настройка, когда MS Visual Studio ставится после установки CAD системы NX;

3 ручная настройка проекта, в случае если ни первый, ни второй путь не позволил получить требуемого результата.

Штатным (соответствующим рекомендациям разработчиков NX) следует считать первый вариант. При нем вся установка и настройка выполняется в автоматическом режиме.

При первом способе во время установки происходит автоматическая интеграция пакета NX в среду разработки MS VS и, если после завершения установки NX открыть Visual Studio (File ->New-> Project), можно увидеть в окне шаблонов проектов вновь добавленные специальные шаблоны для создания приложений NX (рисунок 1.1). Новые шаблоны доступны вместе со стандартными - на вкладках для различных языков программирования: Visual C++, Visual C#, Visual Basic.

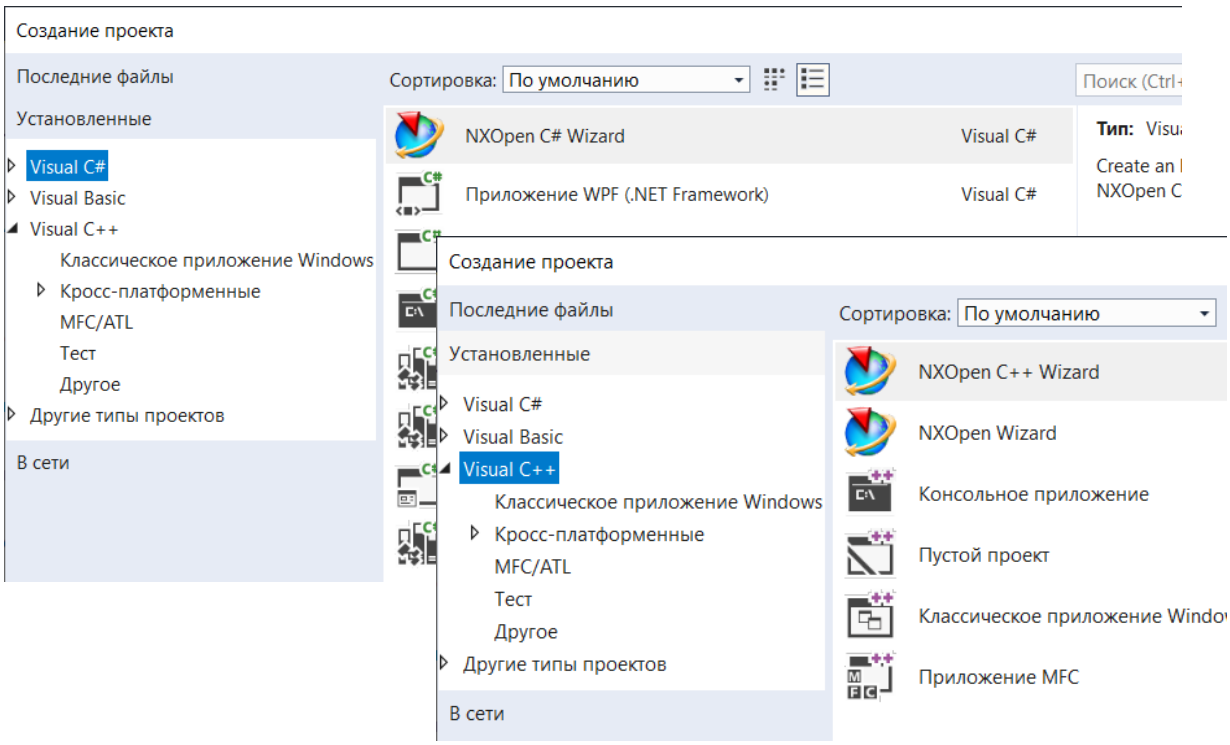


Рисунок 1.1 – Специальные шаблоны для создания приложений NX

1.2 Подготовка среды разработки

Если по каким-то причинам правильный вариант (вариант - 1) установки пакетов провести не удалось и пакеты установлены в обратном порядке, то автоматической интеграции NX в Visual Studio не происходит и шаблоны проектов NX в соответствующих окнах не появляются.

Чтобы вручную установить шаблоны проектов NX в Visual Studio необходимо скопировать содержимое папки X:\Program Files\Siemens\NXver\UGOPEN\vs_files\в папку X:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\, где X: – буква диска с установленным пакетом NX, ver – версия NX.

После копирования указанных папок в окне шаблонов проектов Visual Studio появятся шаблоны для создания приложений NX.

1.2.1 Настройка проекта Visual Studio для разработки прикладного модуля NX

Возможна ситуация, когда необходимо создать проект NX в среде, не воспринимающей готовые шаблоны проектов, находящиеся в каталоге ...\UGOPEN\vs_files. Например, Visual Studio 2008 шаблоны версий NX 4.xx, 5.xx, 6.xx «не понимает».

В таком случае требуется ручная настройка проекта разрабатываемого модуля NX. Для ручной настройки необходимо выполнить следующие шаги.

Шаг 1. Открыть Visual Studio и выбрать «Создать ->Проект».

Шаг 2. Выбрать язык программирования (в нашем случае C++), для чего в левом окне «Создание проекта» выбрать Visual C++. Затем следует выбрать тип проекта - в нашем случае «Консольное приложение (Win32 Console Application)» и дать название проекту (в примере на рисунке 1.2 - Project_UG).

Замечание. Ни в путях, ни в названиях файлов не рекомендуется использоваться символы кириллицы.

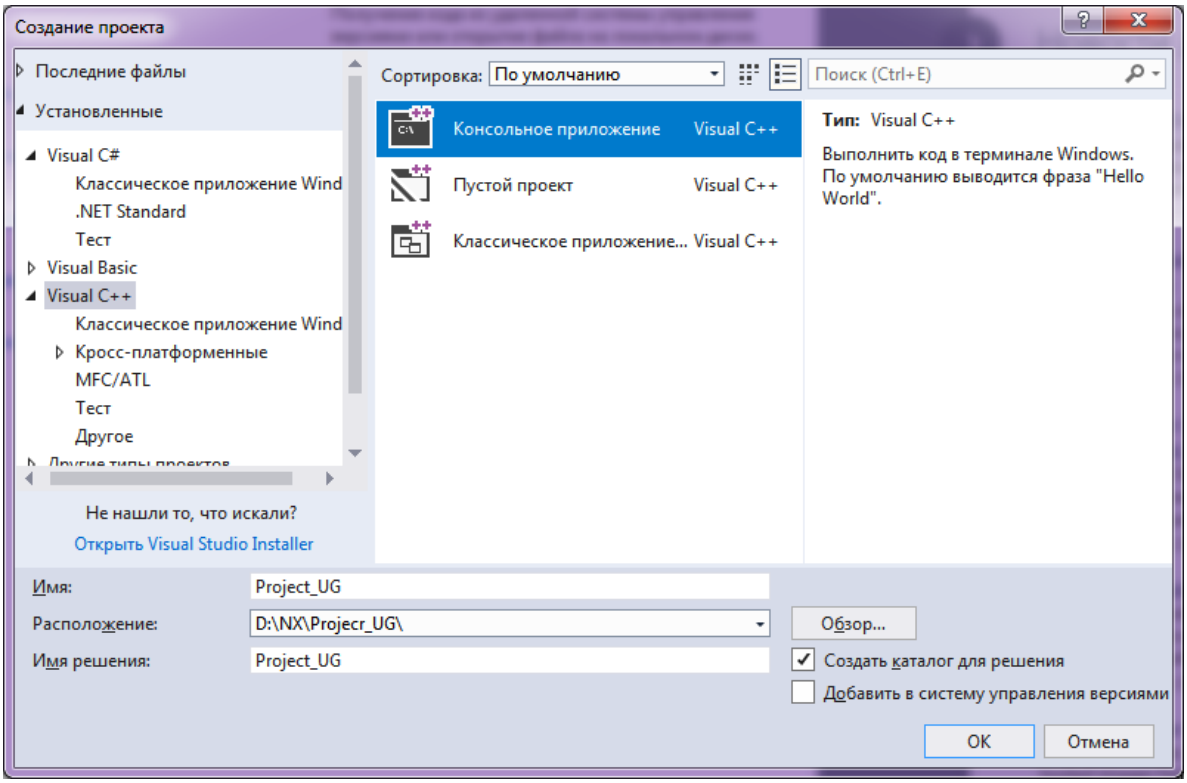


Рисунок 1.2 – Выбор пункта создания консольного приложения

Шаг 3. В опциях создания проекта «Страницы свойств Project_UG» в разделе «Свойства конфигурации→Общие» в пункте «Тип конфигурации» следует выбрать тип «Динамическая библиотека (*.dll)» (рисунок 1.3), если планируется создание внутреннего модуля NX) или «Приложение (*.exe)», если планируется создание внешнего модуля NX.

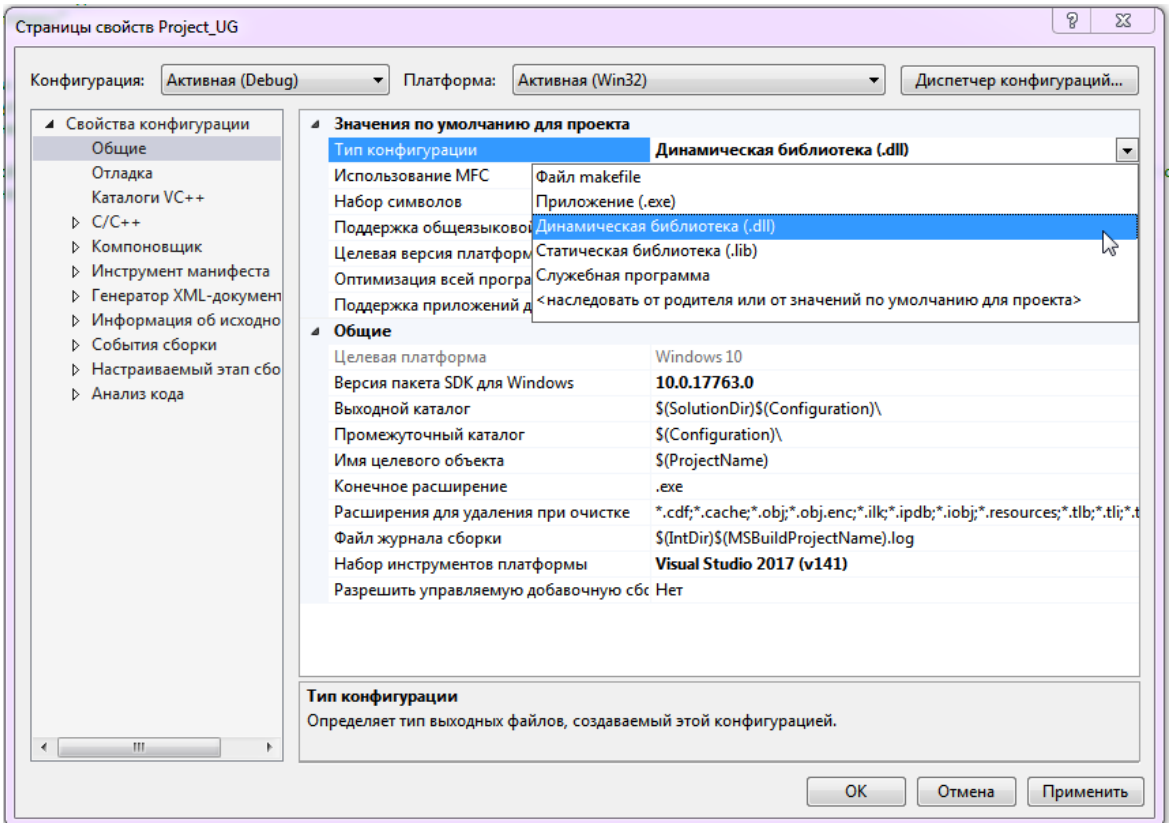


Рисунок 1.3 – Выбор пункта создания динамической библиотеки

Шаг 4. Настройка свойств проекта. Чтобы проект правильно скомпилировался в исполняемый файл NX, необходимо прописать место нахождения подключаемых библиотек и некоторые зависимости компоновщика.

Из вкладки «Свойства конфигурации → C/C++ → Общие» перейти к строке «Дополнительные каталоги включаемых файлов» (Additional Include Directories) добавить местонахождение папок UGOPEN и UGOPENPP (рисунок 1.4), которые

находятся в каталоге NX (в примере обе папки находятся в каталоге c:\Program Files\Siemens\NXver\).

Данная опция подключает UGOPEN и UGOPENPP к проекту NX (для программирования только на языке C достаточно пути к одной папке UGOPEN).

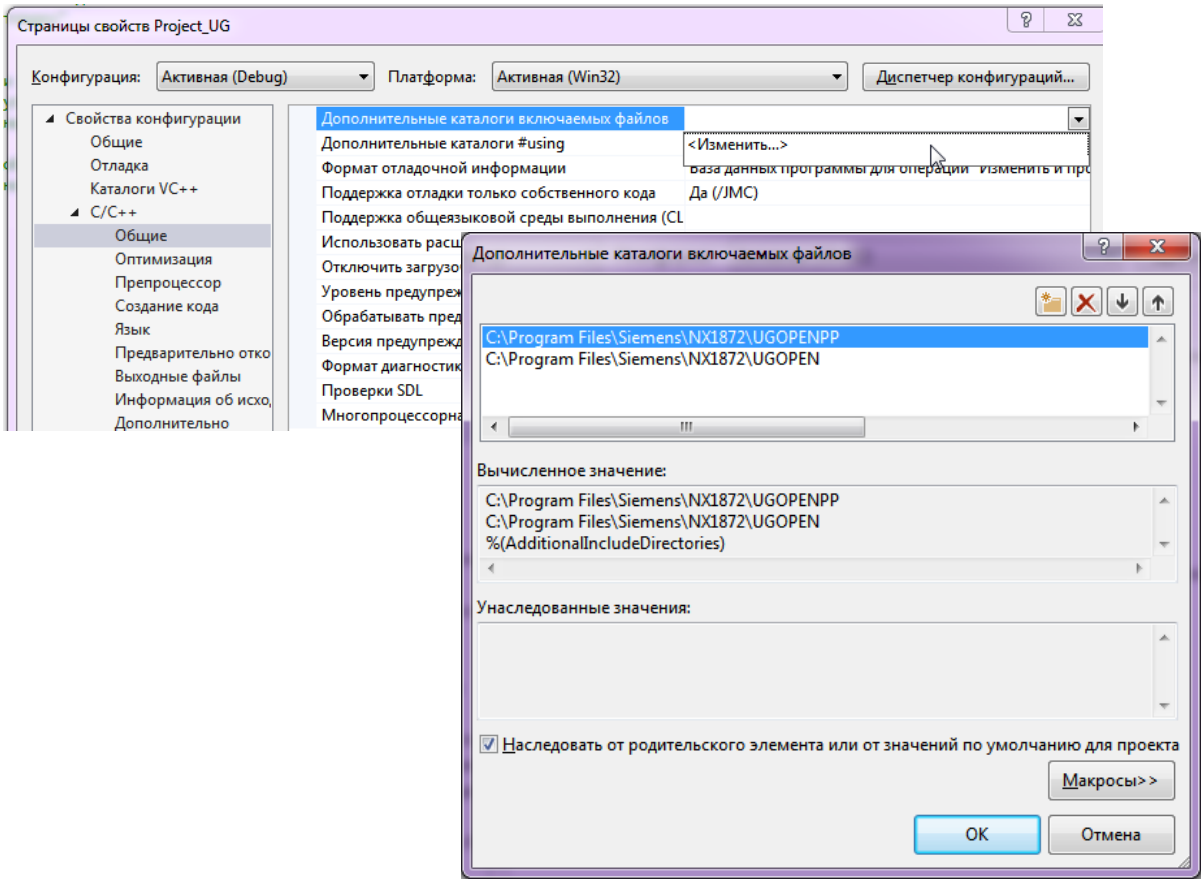


Рисунок 1.4 – Указание местонахождения папок UGOPEN и UGOPENPP

Шаг 5. Из вкладки «Свойства конфигурации→Компоновщик→Общие» выбрать параметр «Дополнительные каталоги библиотек» (Additional Library Directories) и добавить путь к папке UGOPEN (рисунок 1.5). Это необходимо для подключения к проекту основной папки с библиотеками UGOPEN.

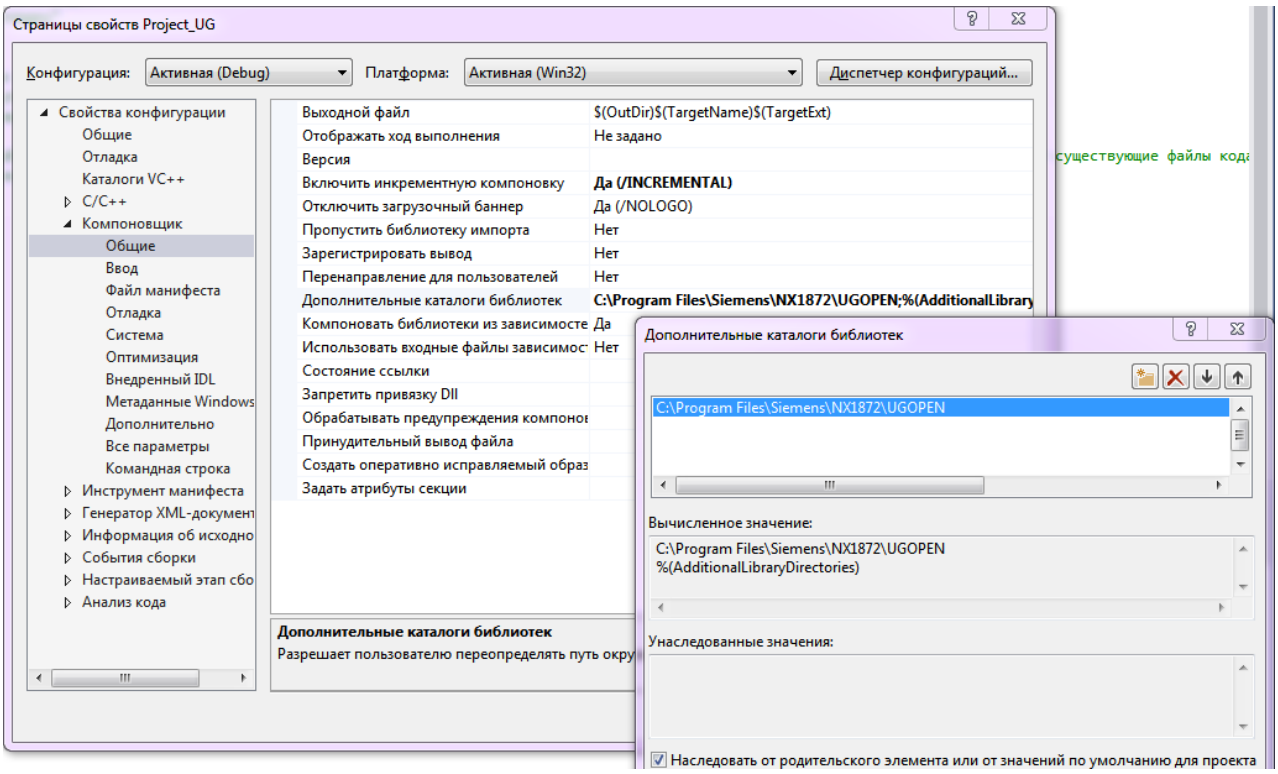


Рисунок 1.5 – Подключение к проекту основной папки с библиотеками UGOPEN

Шаг 6. Из вкладки «Свойства конфигурации→Компоновщик→Ввод» выбрать параметр «Дополнительные зависимости» (Additional Dependencies) и прописать, следующее библиотеки: libufun.lib; libnxopencpp.lib; libugopenint.lib; libnxopenuicpp.lib; libopenpp.lib; libopenintpp.lib; libvmathpp.lib.

Каждую в отдельной строке (рисунок 1.6).

Шаг 7. Из вкладки «Свойства конфигурации→Компоновщик→Система» для параметра «Подсистема» (Subsystem) выбрать Консоль (/SUBSYSTEM:CONSOLE) как показано на рисунке 1.7.

После выполнения описанных шагов система готова для разработки прикладных модулей, в нашем случае в виде прикладной библиотеки, для САПР Siemens NX.

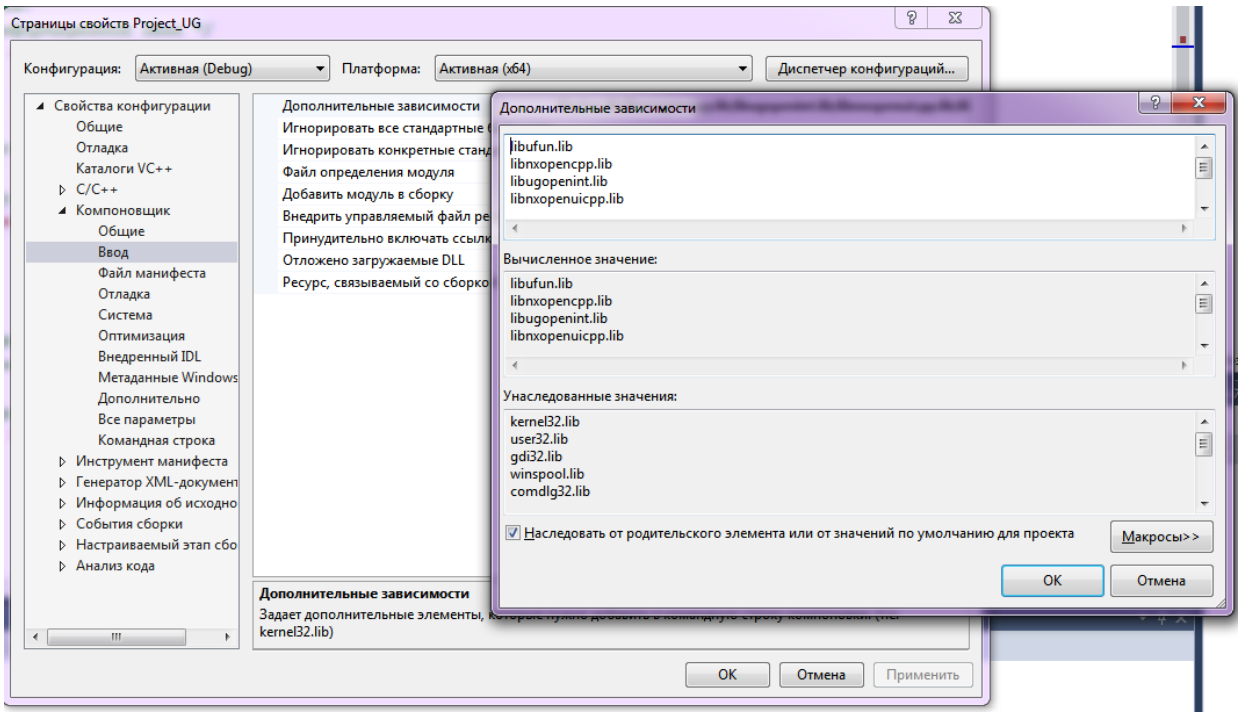


Рисунок 1.6 – Настройка параметра компоновщика «Дополнительные зависимости»

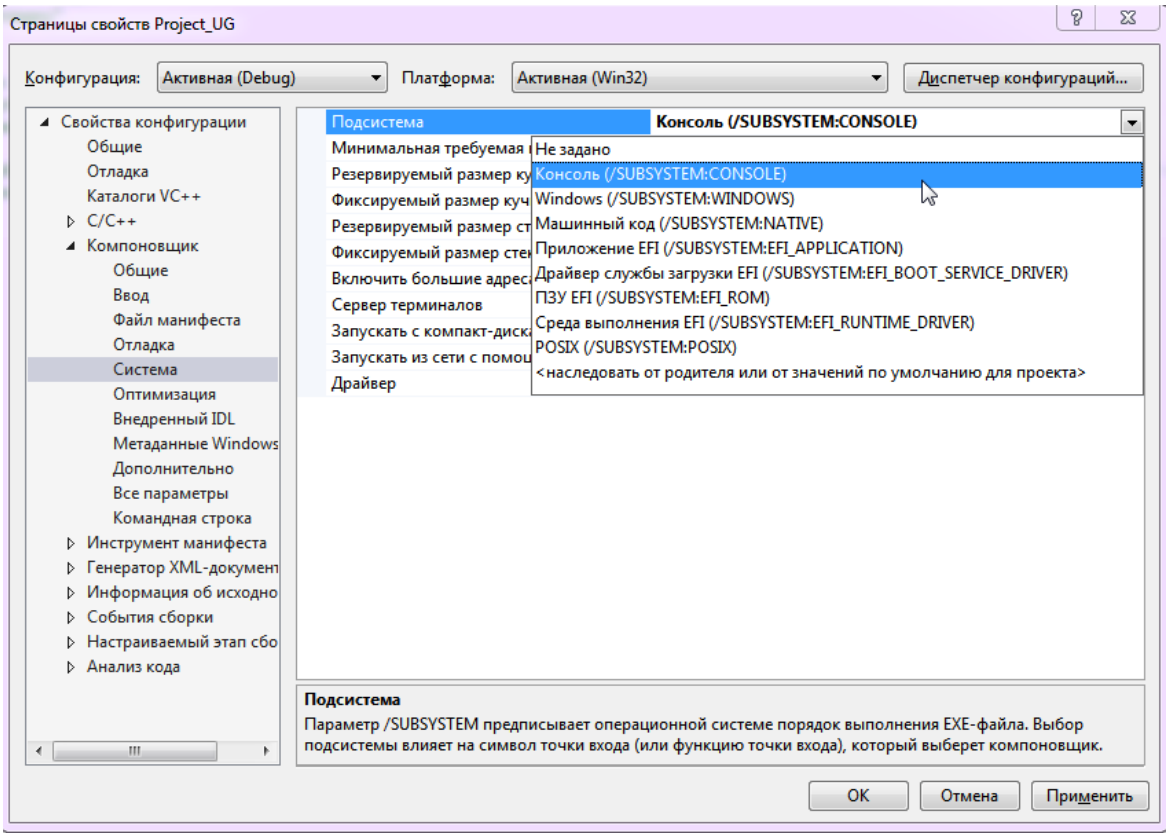


Рисунок 1.7 – Настройка параметра компоновщика «Подсистема»

1.3 Выполнение модулей DLL в среде NX. Шаблон внутренних прикладных модулей NX

Внутренние (Internal) прикладные модули системы NX, разрабатываемые на языке C, создаются в формате DLL библиотек и запускаются из работающей NX системы. Таким образом, внутренние прикладные модули выполняются в адресном пространстве процесса системы NX, что облегчает доступ из модуля к объектам NX, ускоряет операции их создания и редактирования.

Стандартной точкой входа в прикладную DLL NX является функция ufusr (). Рабочий шаблон этой функции представлен в листинге 1.1

Листинг 1.1

```
/* Минимальный шаблон внутреннего модуля NX на базе Open C
входные параметры:
param - если эта функция будет вызвана из menuscript, то сюда передастся
название меню, в других случаях этот параметр не используется;
parm_len - длина строки параметров
выходные параметры:
retcod - возвращаемый код прикладного модуля   */

#include   <uf.h> // Используемые файлы описаний
void  ufusr(char   *param, int   *retcod, int  parm_len)
{
     UF_initialize(); //Инициализация библиотечных функций Open API
     ... // здесь должно быть тело прикладного модуля

UF_terminate(); //Закрытие библиотек функций Open API

}
```

В шаблоне представлена одна, главная, точка входа в прикладной модуль - ufusr (). Всего же в NX предусмотрено 39 различных точек входа в процедуры исполняемого модуля, которые система NX вызывает в различные моменты своей работы. Информация о трех, наиболее часто используемых точках входа представлена в таблице 1.1.

Вторая точка входа имеет сигнатуру:

```
extern   int ufusr_ask_unload(void)
{
return   (<код  выхода  из  модуля>);
}
```

и обычно содержит только код возврата, определяющий правила завершения работы прикладного модуля. Возможные коды возврата представлены в таблице 1.2.

Таблица 1.1 – Основные точки входа во внутренних приложениях системы NX

Имя точки входа	Момент вызова	Назначение точки входа
ufusr	Запуск модуля	Запустить процесс
ufusr_ask_unload	После выгрузки модуля	Принять код завершения процесса
ufusr_cleanup	При завершении работы модуля	Выполнить операции по очистке и освобождению динамических переменных модуля

Таблица 1.2 – Коды возврата из внутренних приложений NX

Код возврата	Описание
UF_UNLOAD_IMMEDIATELY	Обеспечивает выгрузку DLL библиотеки модуля из памяти сразу после завершения работы модуля.
UF_UNLOAD_SEL_DIALOG	Обеспечивает возврат управления окну диалога, вызвавшему данный модуль.
UF_UNLOAD_UG_TERMINATE	Обеспечивает выгрузку DLL модуля вместе с завершением работы системы NX.

Третья точка входа имеет сигнатуру:

```
extern void ufusr_cleanup(void) { return; }.
```

В теле этой функции разработчик должен выполнить очистку и уничтожение всех динамических объектов, созданных им в ходе работы приложения (если таковые остались). В результате - полный типовой шаблон стандартного внутреннего модуля системы NX должен состоять из трех вышеперечисленных функций. Проведем проверочную компоновку и компиляцию такого шаблона на примере.

Пусть необходимо разработать прикладную библиотеку, которая в результате работы каждой точки входа выводит в информационное окно соответствующее текстовое сообщение. Программный код, реализующий поставленную задачу приведен в листинге 1.2.

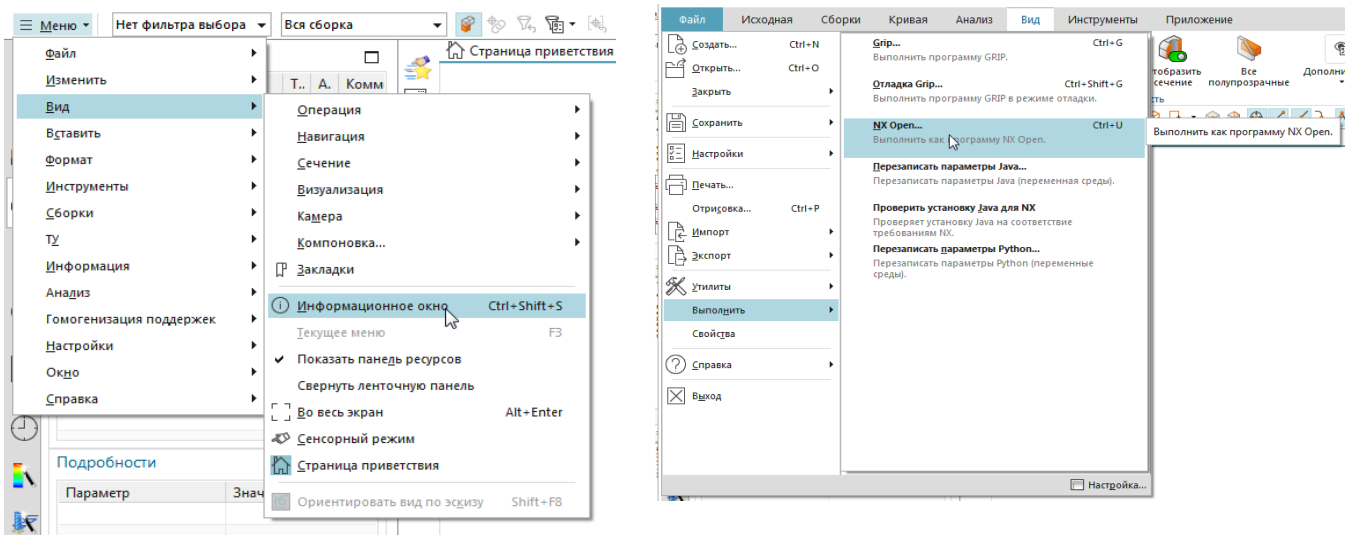
Листинг 1.2

```
#include   <stdio.h>
#include   <uf.h>
#include   <uf_ui.h>
char  msg[120]; //строка для формирования текстового сообщения
// Процедура главной точки входа в прикладной модуль NX
extern   DllExport  void  ufusr(char   *param,  int *retcode, int
parm_len)
{
     if (UF_initialize())      return;
     /* Место для собственного кода разработчика:
     готовим текстовую строку и выводим ее в информационное окно */
     sprintf_s(msg,    "Вход  в   тело   DLL   выполнен \n");
     UF_UI_write_listing_window(msg);
     UF_terminate();
}
     //   Функция, вызываемая системой после выгрузки модуля
     extern   int   ufusr_ask_unload (   void   )
     {   //вывод информации о выходе из процедуры
     sprintf_s   (msg,    "Выход  из   DLL   выполнен \n\n");
     UF_UI_write_listing_window   (msg);
     // выгрузить модуль немедленно
     return    (UF_UNLOAD_IMMEDIATELY);
     }
```

```
//    Функция, вызываемая системой при завершении приложения для
//      выполнения операций очистки динамических переменных модуля
extern      void  ufusr_cleanup(void)
{      //вывод информации о входе в процедуру
       sprintf_s(msg, "Выполнение   функции  ОЧИСТКА \n\n");
       UF_UI_write_listing_window(msg);
       return;
       }
```

Перед запуском модуля сначала нужно запустить NX, создать в нем новый документ, через меню Меню→Вид открыть информационное окно (Information Window), или с помощью клавиш Ctrl+Shift+S (рисунок 1.8, а).

Открыть окно запуска модуля можно через меню Файл →Выполнить→NX Open или сочетанием клавиш Ctrl+U (рисунок 1.8, б). В окне выбирать созданную ранее DLL (в нашем примере Project_UG), произойдет загрузка и выполнение модуля.



а б

Рисунок 1.8 – Выбор отображения информационного окна и запуска прикладной библиотеки

Информационное окно заполняется строками из разработанной программы. Как видно из рисунка 1.9, сначала был осуществлен вход в тело функции `ufsr (...)`, что подтверждает вывод строки 1, затем произошел запуск функции `ufusr_ask_unload (...)` (строка 2) и последней (при выгрузке модуля) была запущена функция `ufusr_cleanup (…)` (строка 3).

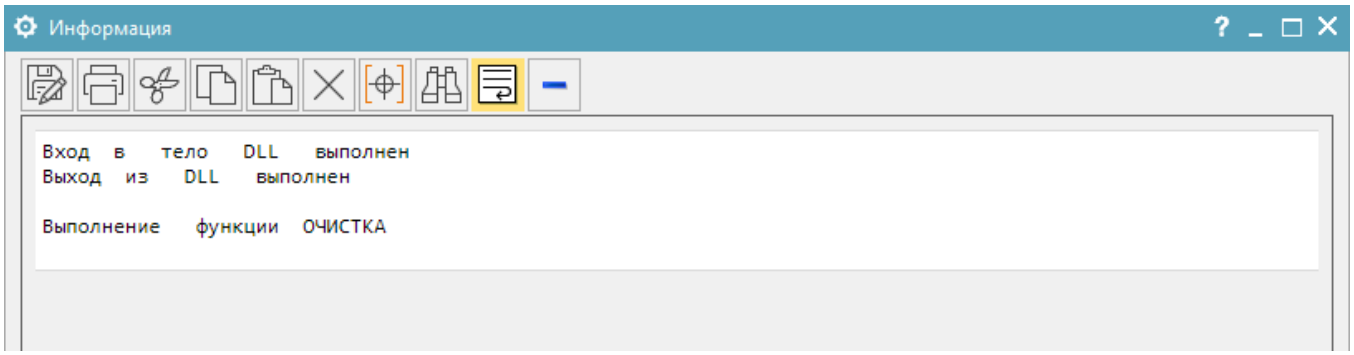
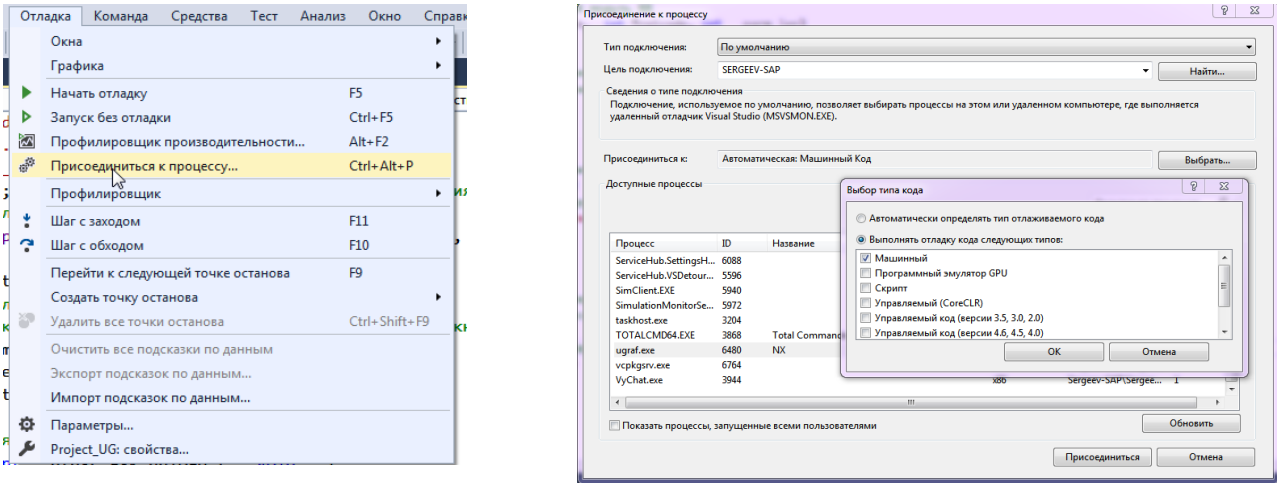


Рисунок 1.9 – Информационное окно с выведенными сообщениями

1.4 Отладка внутренних прикладных модулей NX

Если возникла необходимость отладки (трассировки) программного кода внутреннего прикладного модуля NX, следует выполнить следующие действия. Сначала, в среде MS Visual Studio, применяемой для разработки приложения, следует присоединиться к главному процессу системы NX. Для этого в меню «Отладка» надо выбрать режим «Присоединиться к процессу...» (Ctrl+Alt+P, рисунок 1.10, а) для получения соответствующего окна со списком всех доступных процессов, имеющихся в операционной системе. В этом окне (рисунок 1.10, б) следует нажать кнопку «Выбрать», чтобы открыть окно «Выбор типа кода» для выбора типа кода, подлежащего отладке. Надо включить радиокнопку «Выполнять отладку кода следующих типов» и выбрать тип кода «Машинный».



а б

Рисунок 1.10 – Настройка параметров отладки в MS Visual Studio

После чего окно «Выбор типа кода» закрыть, в списке «Доступные процессы» в колонке «Процесс» выбрать процесс с именем ugraf.exe (основной исполняемый модуль NX) и нажать кнопку «Присоединиться». Среда разработки соединится с основным процессом системы NX. В исходном тексте модуля следует поставить точку останова (F9) на интересующее место программы, перейти в окно системы NX и запустить отлаживаемую DLL.

Как только исполнение кода дойдет до точки останова, будет вызвана среда разработки и программист сможет вести отладку разрабатываемого приложения в стандартном режиме (рисунок 1.11).

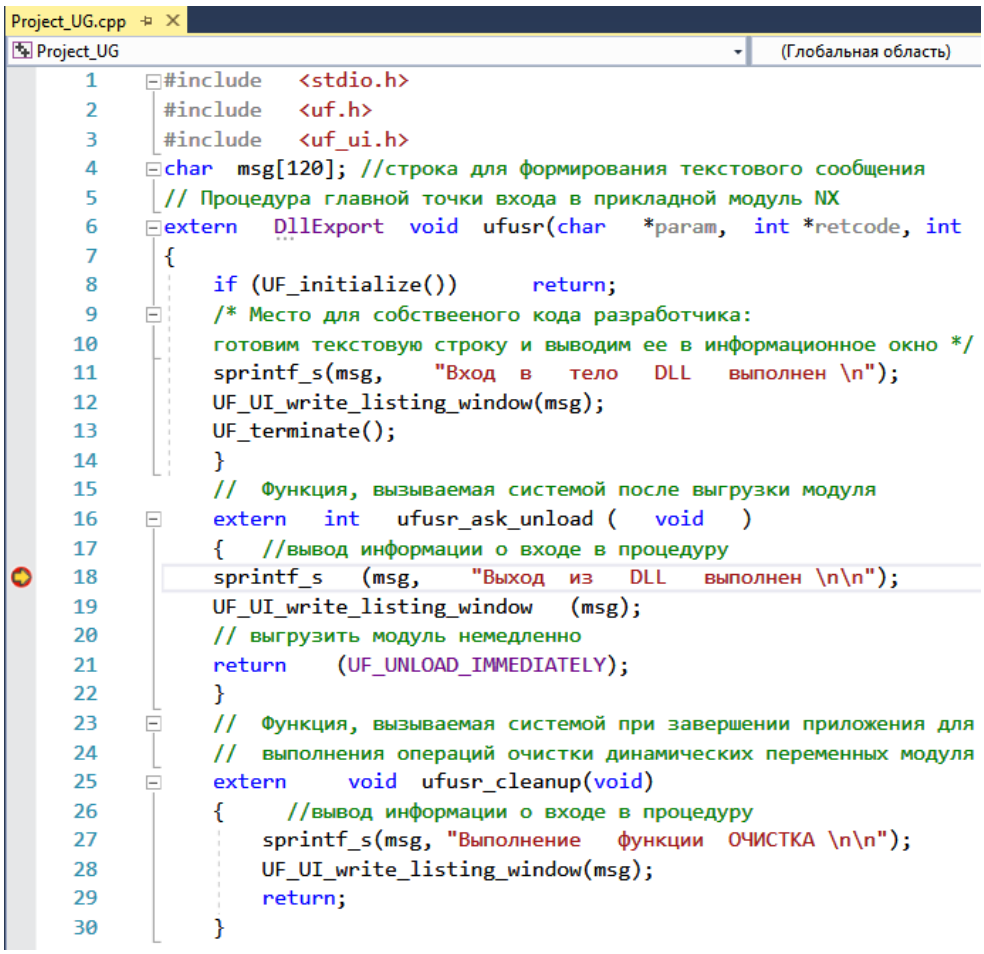


Рисунок 1.11 – Выполнение библиотеки в режиме отладки

1.5 Описание функций Open API NX языка C

Для разработки прикладных модулей NX на языке C, создатели CAD (CAM, CAE) системы разместили все необходимые компоненты (файлы определений функций, констант и структур, библиотеки функций и примеры их использования) в каталоге установленной системы NX ... \UGOPEN. Перечень некоторых из этих файлов описаний с общими пояснениями содержащихся в них функций и структур представлен в таблице 1.3.

Файлы описаний содержат множество комментариев, обеспечивающих разработчика полезной информацией и их изучение, является одним из часто используемых методов освоения программирования в NX.

При изучении функций Open API NX следует учитывать следующие соглашения. Имена функций (в большинстве случаев) составляются по правилам в соответствии с рисунком 1.12.

Таблица 1.3 – Перечень некоторых файлов описаний функций Open API NX языкаC

Имя файла описаний	Назначение функций в файле описаний
uf.h	Общий интерфейс NX/Open - функции общего назначения, использующиеся большинством программ, а также специализированные функции, не входящие в другие интерфейсы.
uf_abort.h	Функции, позволяющие перехватывать нажатие пользователем кнопки ОК на окошке отмены для отмены операции.
uf_modl.h	Modeling Create - функции, для моделирования - создания примитивов и проведения операций над ними.
uf_modl_curves.h	Моделирование кривых.
uf_ugopenint.h	Общий интерфейс NX/Open - функции общего назначения, использующиеся большинством внутренних модулей.
uf_ui.h	User Interface - функции для работы с пользовательским интерфейсом NX.

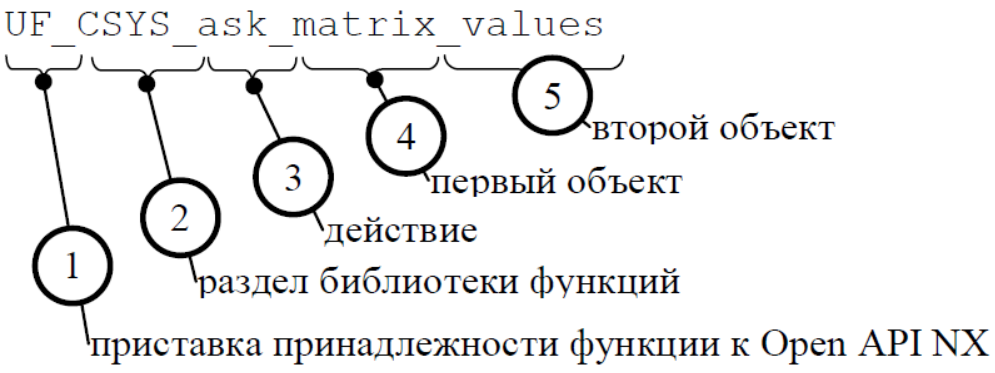


Рисунок 1.12 – Иллюстрация правил составления имен функций Open API NX

Рассмотрим позиции, отмеченные на рисунке 1.12.

Позиция 1. Имя функции Open API начинается с приставки UF_, что определяет принадлежность функции к рассматриваемой технологии программирования.

Позиция 2. Далее идет префикс, обозначающий раздел библиотек, к которому относится функция. В таблице 1.4 приведены префиксы наиболее часто применяемых файлов описаний библиотек.

Таблица 1.4 – Префиксы файлов описаний библиотек Open API NX

Префикс	Наименование раздела библиотек
CSYS	Функции работы с координатами
DISP	Функции рисования на дисплее
DRAW	Функции черчения
CURVE	Функции работы с кривыми
FACET	Функции работы с фасетами
MODL	Функции моделирования объектов
OBJ	Функции работы с объектами модели
UI	Функции графического интерфейса пользователя

Позиция 3. далее идет префикс, обозначающий действие, выполняемое функцией:

- `ask` - получить (спросить) данные;
- `set` - установить параметры;
- `create` - создать объект;
- `delete` - удаление объекта;
- `move` - перемещение объекта и т.д.

Иногда на месте этого префикса может стоять имя объекта, например в функции: `UF_MODL_feature_can_be_copied (...),` устанавливающей факт «может ли элемент быть скопированным».

Позиция 4. Далее следует имя первого объекта, на который распространяется объявленное действие.

Позиция 5. Далее - имя второго объекта (может отсутствовать), с которым взаимодействует первый.

Описанные соглашения в именах функций могут слегка варьироваться, но, в любом случае, внимательное прочтение имени функции может дать общее представление об её назначении.

Почти все функции Open API NX возвращают целое число. Оно равно нулю, если функция выполнилась успешно. В противном случае число представляет из себя код ошибки функции. Код ошибки всегда можно перевести в текст с использованием функции:

`UF_get_fail_message(…).`

При описании параметров функций используются следующие обозначения:

- `<I>` - входной параметр функции;
- `<O>` - выходной параметр функции;
- `<EC>` - в параметре возвращается код ошибки формата NX;
- `<OF>` - выходной параметр, получающий адрес созданной в функции динамической переменной, которую впоследствии надо освободить с помощью специальной функции. Имя функции для освобождения переменной обычно указывается тут же в описании параметра.

Описания применяемых переменных тоже придерживаются определенных соглашений, их имена имеют следующие предопределенные окончания:

- `_t` - переменная, хранящая данные;
- `_p_t` - переменная - указатель на данные;
- `_s` - переменная типа структуры;
- `_u_t` - переменная типа объединения;
- `_u_p_t` - переменная - указатель на объединение;
- `_f_t` - переменная - указатель на функцию.

Часто встречающимся типом переменной в Open API NX является тег (`tag_t`) или указатель на тег (`tag_p_t`). Понятие тега в Open API NX в чем-то аналогично

понятию хендела (handle) в ОС Windows. Это некоторый указатель (не всегда в прямом смысле) на объект системы, который используется во многих функциях и через который обеспечивается доступ к объекту, на который он указывает. Следует понимать, что `tag_t` объекта выделяется динамически и не стоит рассчитывать на то, что при следующем открытии файла он сохранит свое значение. Некоторые начинающие программисты пытаются сохранять тег объекта и передавать его из сессии в сессию для обращения к объекту.

В файле описаний uf_curve.h содержатся функции построения точек и кривых разного вида в системе NX при программировании на языке C.

1.5.1 Построение окружности

Создадим пример приложения, строящего полную окружность в 3D пространстве изучаемой системы. Для построения воспользуемся функцией, описанной в uf_curve.h как:

```
extern UFUNEXPORT int UF_CURVE_create_arc(
                UF_CURVE_arc_p_t arc_coords,
                tag_t* arc);
```

Эта функция, практически как и все функции Open API NX, в результате своего выполнения возвращают число 0, как признак успешного выполнения функции. Не нулевое значение - будет означать ошибку выполнения. У функции два параметра. Первый - структура:

`UF_CURVE_arc_p_t  arc_coords` - описана в файле описаний, как:

```
struct   UF_CURVE_arc_s
{/* тег матрицы, определяющей положение объекта в системе координат */
    tag_t matrixtag;
    double  start_angle;    //начальный угол создаваемой дуги в радианах
    double  end_angle;      //конечный угол создаваемой дуги в радианах
    double  arc_center[3] ; //координаты центра дуги
    double  radius;         //радиус дуги
}
```

По структуре видно, что функция может создавать дуги с произвольным началом и концом в пространстве. Окружность - частный случай для этой функции.

Что касается матрицы преобразований координат (`matrix_tag`), то это знакомый (по графическим функциям ядра операционной системы Windows) объект - массив, состоящий из девяти элементов. Применяемые в компьютерной графике преобразования координат делятся на следующие категории: трансляция, масштабирование, вращение, сдвиг и отражение.

Трансляция означает добавление констант к горизонтальным и вертикальным координатам объекта.

Масштабирование – растягивание или сжатие горизонтального и вертикального экстента объекта.

Вращение – точки объекта вращаются вокруг начала отсчета.

Сдвиг — это преобразование прямоугольников в параллелограммы. Сдвиг добавляет смещение к горизонтальной координате точки, пропорциональное вертикальной координате, и наоборот.

Отражение отображает объект относительно горизонтальной или вертикальной оси.

Все эти преобразования можно выразить в матричной форме, используя матрицы размером 3×3. Хотя любое линейное преобразование можно выразить в форме последовательности пяти перечисленных основных преобразований, обобщенное линейное преобразование может не быть просто трансляцией, масштабированием, вращением, сдвигом или отражением. Обобщенное линейное преобразование (для графических функций API Windows) выражается следующей матричной формулой

$$\begin{matrix} x^1 & y^1 & z^1 \end{matrix} = \begin{matrix} x & y & z \end{matrix} \cdot \begin{matrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ D_X & D_Y & D_Z \end{matrix}. \qquad (1.1)$$

Примерно такой же тип матриц передается функции создания дуги в первом поле структуры `UF_CURVE_arc_p_t  arc_coords`.

Матричный тэг определяет ориентацию дуги. И если мы задаем дугу с центром в точке `arc_center [3],` то итоговые координаты всех точек дуги, передаваемые к построению, определяются матричным умножением:

$$R = X \cdot T . \qquad (1.2)$$

где R - вектор отображаемых координат объекта $x^1 \quad y^1 \quad z^1$,

X - вектор с исходными координатами объекта $x \quad y \quad z$,

T - матрица ориентации, представленная в `matrix _tag`.

В нашем проекте для задания матрицы преобразования воспользуемся функцией:

```
int UF_CSYS_ask_wcs(tag_t* wcs_id),
```

описанной в файле элементов координатных систем uf_csys.h.

Эта функция получает от CAD-оболочки объектный тег её рабочей системы координат. Тег матрицы вращения можно извлечь из объектного тега с помощью функции:

```
extern UFUNEXPORT int UF_CSYS_ask_matrix_of_object(
     tag_t object_id,      //тег объекта системы координат
     tag_t* matrix_id)     //тег матрицы вращения системы координат.
```

Реализуя все описанные действия и функции, получим листинг, на примере построения окружности в 3D пространстве сцены NX.

Листинг 1.3.

```
#include   <uf.h>          //файл описаний общих функций
#include   <uf_curve.h>    //файл описаний функций кривых
#include   <uf_csys.h>     //файл описаний функций работы с координатами
void  ufusr(char   *param, int   *retcode, int  paramLen)
{
     tag_t     arc_id, wcs_tag;         //тэги окружности
                                        //и мировой  системы координат
     UF_CURVE_arc_t  arc_coords; //структура свойств дуги
     //если активизация функций Open API не прошла - прервать программу
     if (UF_initialize())  return;
```

```
    arc_coords.start_angle = 0.0; //начальный угол окружности
    arc_coords.end_angle = 360.0   *   DEGRA;   //конечный угол
    arc_coords.arc_center[0] = 0.0;    //координата центра X
    arc_coords.arc_center[1] = 0.0; //координата центра Y
    arc_coords.arc_center[2] = 20.0; //координата центра Z
    arc_coords.radius = 50.0;   //радиус окружности
    UF_CSYS_ask_wcs(&wcs_tag); //получение абсолютных координат
    //"перенос" абсолютных координат на создаваемую окружность
    UF_CSYS_ask_matrix_of_object(wcs_tag, &arc_coords.matrix_tag);
    //построение окружности
    UF_CURVE_create_arc(&arc_coords, &arc_id);
    UF_terminate();
}
int   ufusr_ask_unload(void) //функция выхода из приложения
{
    return    (UF_UNLOAD_IMMEDIATELY);
}
```

Константа *DEGRA*, примененная в программе, задана в файле описаний, как значение одного градуса в радианах и поэтому применяется для перевода значений из градусов в радианы. Вместо выражения (360.0 * *DEGRA*) можно было использовать константу *TWOPI*, которая определена как 2π радиан и таким образом задает полную окружность.

В результате выполнения представленного кода получается окружность, диаметром 100 мм, сдвинутая по оси *Z* от начала координат на расстояние 50 мм, представленная на рисунке 1.13.

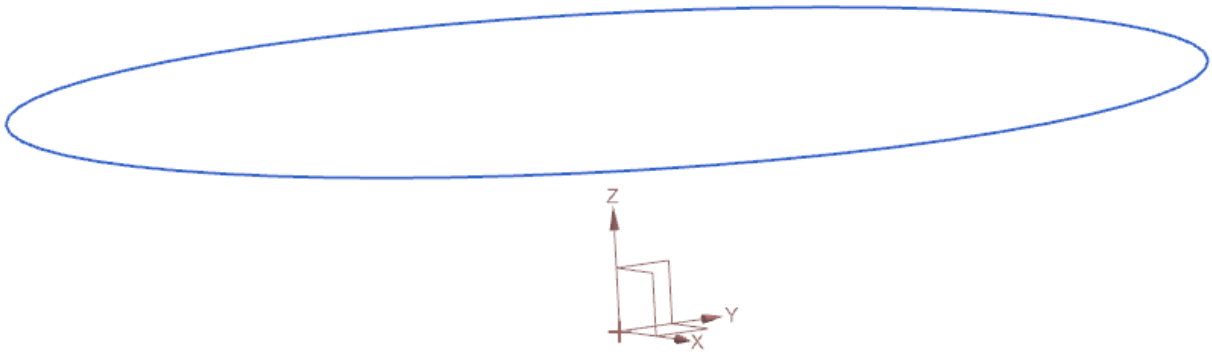


Рисунок 1.13 – Окружность, полученная в результате выполнения кода, представленного в Листинге 1.3

Возможен вариант построения окружности по трем точкам. Для этого используется функция:

```
int   UF_CURVE_create_arc_thru_3pts(
    int   create_flag, /*  флаг, если 1 - то строится дуга;
                                  2 - строится окружность */
    double   first_point[3],      // координаты первой точки
    double   second_point [ 3 ],  // координаты второй точки
    double   third_point [ 3 ],   // координаты третьей точки
    tag_t*   arc_tag              //на выходе - тэг созданного объекта
);
```

Разбирая параметры функции, можно увидеть, что для построения окружности необходимо задать координаты трех точек в пространстве и установить переменную `create_flag` в 2. Следует отметить, что для построения окружности по трем точкам не требуется задавать тег матрицы, определяющей ее положение в системе координат. Пример построения окружности по трем точкам с радиусом 10 мм приведен в листинге 1.4.

Листинг 1.4.

```
#include   <uf.h>          //файл описаний общих функций
#include   <uf_curve.h>    //файл описаний функций кривых
void  ufusr(char   *param, int   *retcode, int  paramLen)
{
     int   create_flag=2; /*  выбираем окружность */
```

```
    double   first_point[3] = { 10,0,0 };      // координаты первой точки
    double   second_point[3] = { 0,10,0 };     // координаты второй точки
    double   third_point[3] = { -10,0,0 };     // координаты третьей точки
    tag_t    arc_tag;                  //на выходе - тэг созданного объекта
    //если активизация функций Open API не прошла - прервать программу
    if (UF_initialize())  return;
    //построение окружности
    UF_CURVE_create_arc_thru_3pts(create_flag, first_point, second_point,
third_point, &arc_tag);
    UF_terminate();
}
int   ufusr_ask_unload(void) { //функция выхода из приложения
    return    (UF_UNLOAD_IMMEDIATELY);
}
```

1.5.2 Построение отрезка

Для построения прямой достаточно иметь информацию о координатах двух точек пространства.

Функция для построения прямой линии в NX для языка C имеет вид:

```
int UF_CURVE_create_line(
     UF_CURVE_line_p_t line_coords, tag_t *line)
```

Первый параметр функции - структура, содержащая координаты точек концов прямой. Структура состоит из двух массивов по три элемента двойной точности (по одному массиву на каждую точку):

```
struct UF_CURVE_line_s {
double start_point[3];  //координаты начальной точки
double end_point[3];    //координаты конечной точки
};
```

Пример модуля для создания прямой линии с помощью вышеуказанной функции представлен в листинге 1.5.

Листинг 1.5.

```
#include   <uf.h>
#include   <uf_curve.h>
void  ufusr(char   *param, int   *retcode, int  paramLen)
{
      tag_t         entid = 0;       // Идентификатор объекта «линия»
      UF_CURVE_line_t  line_coords;   /* структура концевых точек линии */
      if (UF_initialize())  return;
      //заполнение координат концевых точек линии
      line_coords.start_point[0] = 0.;// X1
      line_coords.start_point[1] = 0.;// Y1
      line_coords.start_point[2] = 0.;// Z1
      line_coords.end_point[0] = 100.;// X2
      line_coords.end_point[1] = 100.;// Y2
      line_coords.end_point[2] = 100.;// Z2
      // построение линии
      UF_CURVE_create_line(&line_coords, &entid);
      UF_terminate();
}
//процедура выхода из приложения
int  ufusr_ask_unload(void) {
      return   (UF_UNLOAD_IMMEDIATELY);
};
```

В результате будет построен отрезок в соответствии с рисунком 1.14.

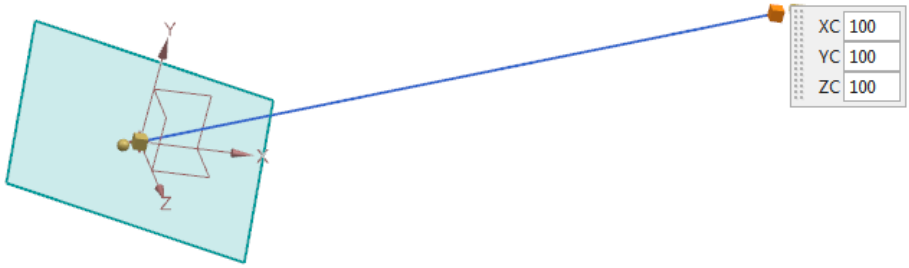


Рисунок 1.14 – Отрезок, полученный в результате выполнения кода, представленного в Листинге 1.5

1.6 Задание для лабораторной работы

1 Изучить теоретический материал.

2 Настроить проект Visual Studio для корректной компиляции прикладной библиотеки Siemens NX.

3 Разработать прикладную библиотеку NX которая строит чертеж детали по заданному варианту, приведенному в таблице А.1, приложения А.

4 Подготовить ответы на контрольные вопросы.

1.7 Содержание отчета

В отчете по лабораторной работе согласно СТО 02069024.110-2008 [5] должны содержаться следующие пункты:

- название лабораторной работы;
- цель работы;
- задание на лабораторную работу;
- код разработанной программы с комментариями;
- экранные формы с отображением результата выполнения программы в системе NX;
- выводы;
- список использованных источников.

1.8 Контрольные вопросы

1) Способы разработки собственных инструментов и приложений в NX.
2) Последовательность среды разработки приложений Visual Studio и Siemens NX.
3) Настройка приложений при различных вариантах установки.
4) Ручная настройка проекта разрабатываемого модуля NX.
5) В каких каталогах находятся библиотеки функций для C и C++?

6) Каким образом в NX запускаются библиотеки разработанные *.dll?

7) Варианты точек входа во внутренних приложениях системы NX.

8) Создание типового шаблона стандартного внутреннего модуля системы NX.

9) Как обеспечить выгрузку разработанного модуля из памяти сразу по завершению его работы.

10) Подключение отладчика к приложению NX.

11) Точки останова.

2 Моделирование кривых

2.1 Построение сплайнов

Математической моделью трехмерного сплайна является матричное произведение матрицы весовой функции положения ключевых точек на сплайне [F] и матрицы геометрической информации [G] в этих точках.

$$P_k\ \tau\ =\ F\ \cdot\ G \tag{2.1}$$

Это произведение определяет вектор координат некоторой *k*-той точки сплайна, находящейся на относительном расстоянии τ (по дуге сплайна) от его начала.

Компоненты матрицы весовой функции определяются через параметр *t* сплайна, где *t* — это длина дуги сплайна от начала сплайна до некоторой его расчетной точки. Для сплайна третьей степени, например, матрица весовой функции состоит из следующих элементов:

$$F\ =\ F_1\ \tau\quad F_2\ \tau\quad F_3\ \tau\quad F_4\ \tau\ \ , \tag{2.2}$$

где

$$\tau =\ \frac{t}{t_{k+1}}\ , \tag{2.3}$$

$$F_{1k}\ \tau\ = 2\tau^3 - 3\tau^2 + 1, \tag{2.4}$$

$$F_{2k}\ \tau\ = -2\tau^3 + 3\tau^2, \tag{2.5}$$

$$F_{3k}\ \tau\ = \tau\ \tau^2 - 2\tau + 1\ t_{k+1}, \tag{2.6}$$

$$F_{4k}\ \tau\ = \tau\ \tau^2 - \tau\ t_{k+1}. \tag{2.7}$$

Индексом к в этих формулах обозначен номер некоторой промежуточной ключевой точки на сплайне.

$$1 \leq k\ \leq\ n - 1\ , \tag{2.8}$$

где *n* - количество ключевых точек, через которые строится сплайн.

Понятие параметра *t* сплайна для некоторой промежуточной расчетной точки, лежащей между ключевыми точками *k* и *k*+1, проиллюстрировано рисунком 2.1.

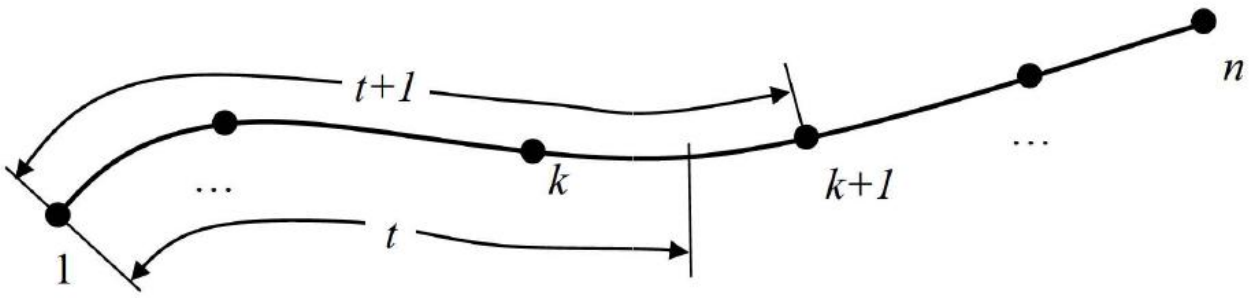


Рисунок 2.1 – Иллюстрация понятия параметра *t* сплайна

Элементы матрицы [*G*] (для того же кубического сплайна) определяются из выражения

$$G^{\ T} = \ P_k \quad P_{k+1} \quad P'_k \quad P'_{k+1}\ , \tag{2.9}$$

где P_k и P_{k+1} - вектора координат ключевых точек сплайна,

P'_k и P'_{k+1} - вектора первых производных (касательных) в этих точках.

Для построения сплайнов в Open API NX для языка **C** предусмотрено две функции:

`UF_CURVE_create_spline_thru_pts` - построение сплайна по точкам;

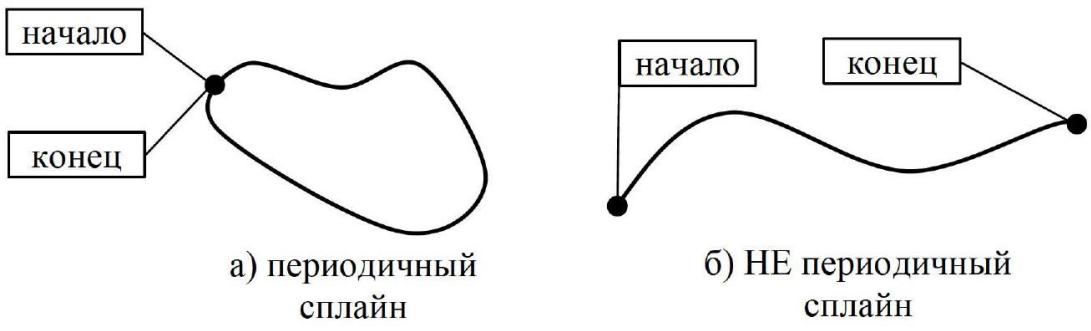
`UF_CURVE_create_spline` - построение произвольного сплайна.

Полный синтаксис первой функции выглядит следующим образом:

```
int  UF_CURVE_create_spline_thru_pts(
int  degree,            // степень сплайна
int  periodicity, /*периодичность сплайна:
0 - не периодичный;
1 - периодичный. */
int  num_points,     //количество заданных точек
UF_CURVE_pt_slope_crvatr_t  point_data[],/*массив
координат точек и направлений касательных в них*/
double  parameters[],   //параметры входных точек
int   save_def_data,   /*  1 - сохранять входные данные при создании кривой,
                          2 - не сохранять */
tag_t   *spline_tag         //тег создаваемого объекта
)
```

Первое поле в структуре отвечает за степень сплайна. Обычно она изменяется от 3 и выше.

Второе поле - периодичность сплайна. Замкнутые сплайны называются периодичными (рисунок 2.2): начальная и конечная точки кривой у периодичных сплайнов совпадают. Все остальные сплайны - не периодичные.



а) периодичный сплайн

б) НЕ периодичный сплайн

Рисунок 2.2 – Периодичный и не периодичный сплайны

Четвертое поле структуры - данные по точке, в свою очередь представляется структурой следующего вида:

```
struct UF_CURVE_pt_slope_crvatr_s
{
double point[3];//координаты точки
int slope_type; //метод задания наклона кривой
double slope[3];//  вектор, задающий направление касательной
int crvatr_type;//метод задания кривизны кривой
double crvatr[3]//вектор, направления кривизны
}
```

В первое поле этой структуры заносятся координаты точки (*x, y, z*).

Во второе поле - код метода задания касательной в точке. Варианты констант, определяющих метод задания касательной следующие:

UF_CURVE_SLOPE_NONE - касательная не задается;

UF_CURVE_SLOPE_AUTO - касательная определяется системой автоматически;

UF_CURVE_SLOPE_VEC - касательная задается вектором (используется величина и направление вектора);

UF_CURVE_SLOPE_DIR - вектором задается только направление касательной.

В третье поле заносится сам вектор касательной (его орты *I, J, K*).

Следующие два поля повторяют предыдущие два, но для вектора, задающего кривизну кривой в точке (величина, обратная радиусу кривой в точке). Константы для описания методов задания кривизны следующие:

UF_CURVE_CRVATR_NONE - кривизна не назначена;

UF_CURVE_CRVATR_AUTO_DIR - автоматическое определение направления кривизны, при этом тип наклона кривой должен стоять UF_CURVE SLOPE_AUTO;

UF_CURVE_CRVATR_VEC - кривизна задается размером вектора пользователя.

Вектор кривизны должен обязательно быть перпендикулярным к вектору касательной.

Когда касательная задана константами UF_CURVE_SLOPE_VEC или UF_CURVE_SLOPE_DIR, вектор кривизны определяет размер и направление кривизны.

Когда касательная задана константой UF_CURVE_SLOPE_AUTO, только величина вектора используется для задания величины кривизны, направление вычисляется автоматически.

В сплайнах со степенью менее 2 наклоны касательных в точках не назначаются.

В сплайнах со степенью менее 3 кривизна в точках не назначается.

После структуры данных в структуре UF_CURVE_create_spline thru_pts идет массив параметров точек.

Параметр t_i входных точек p_i - это относительная (или абсолютная) длина кривой (вдоль кривой) от начала до рассматриваемой точки (что полностью соответствует понятию матричного параметра t, рисунок 2.3).

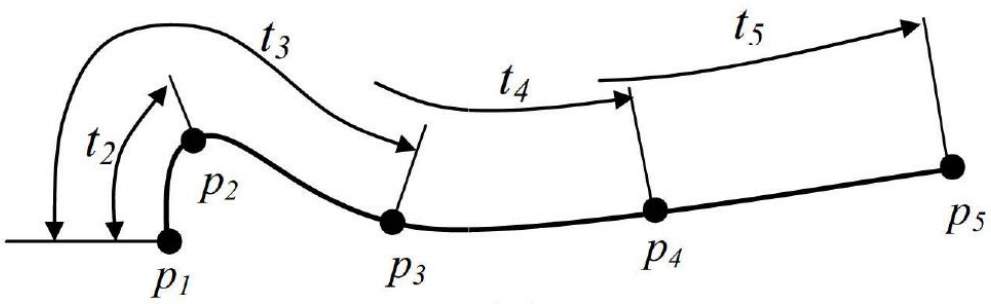


Рисунок 2.2 – Иллюстрация параметра t_i входных точек p_i сплайна

Массив параметров входных точек используются, если необходимо обеспечить монотонность кривой. Для этого следует задать parameters (i) < parameters (i + 1) для всех точек сплайна.

Если определение и задание массива параметров точек вызывает проблему, то указатель этого поля в структуре можно установить в NULL, тогда система NX будет сама подбирать этот массив.

Пример построения сплайна в 3D пространстве сцены NX приведен в листинге 2.1.

Листинг 2.1.

```
#include <stdio.h>
#include <uf_curve.h>
#include <uf.h>
#define NUMBER_POINTS 5 //количество точек в сплайне зададим пять
void ufusr(char *param, int *retcode, int paramLen)
{
	int degree = 3; // строить будем сплайн третьей степени
	int periodicity = 0; // не периодичный сплайн
	int num_points = NUMBER_POINTS;
	// объявление массива структур с данными по точкам
	UF_CURVE_pt_slope_crvatr_t point_data[NUMBER_POINTS];
	// зададим массив параметров точек
	double parameters[NUMBER_POINTS] = { 0.00, 0.17, 0.32, 0.45, 1.29 };
	// зададим массив координат точек
	double points[3 * NUMBER_POINTS] = {
	1.1000, 0.5320, 2.0000,
	1.5240, 0.6789, 2.3000,
	2.0000, 0.9000, 3.5956,
	2.3456, 1.3456, 3.7890,
	3.1000, 2.4567, 3.3214 };
	// зададим массив методов задания касательных
	int slopeTypes[NUMBER_POINTS] = {
	UF_CURVE_SLOPE_DIR,
	UF_CURVE_SLOPE_AUTO,
	UF_CURVE_SLOPE_NONE,
	UF_CURVE_SLOPE_DIR,
	UF_CURVE_SLOPE_VEC };
	// зададим массив векторов касательных в точках
	double slopeVecs[3 * NUMBER_POINTS] = {
	1.2300, 5.0506, 4.0360,
	0.0000, 0.0000, 0.0000,
```

```
0.0000, 0.0000, 0.0000,
0.5000, 1.0000, 0.5000,
1.0000,-2.0000, 1.0000 };
// зададим массив методов задания кривизн в точках
int crvatrTypes[NUMBER_POINTS] = {
UF_CURVE_CRVATR_NONE,
UF_CURVE_CRVATR_AUTO_DIR,
UF_CURVE_CRVATR_NONE,
UF_CURVE_CRVATR_VEC,
UF_CURVE_CRVATR_VEC };
// зададим массив векторов кривизн в точках
double crvatrVecs[3 * NUMBER_POINTS] = {
0.0000, 0.0000, 0.0000,
1.0000, 2.5780, 5.6700,
0.0000, 0.0000, 0.0000,
1.0000,-1.0000, 1.0000,
-1.0000,-1.0000,-1.0000 };
int i, save_def_data = 1;
tag_t spline_tag;
if (!UF_initialize())
{ /* выполним цикл переноса данных из заданных массивов в
соответствующие поля структуры данных сплайна */
      for (i = 0; i < NUMBER_POINTS; i++)
      {
            point_data[i].point[0] = points[3 * i];
            point_data[i].point[1] = points[3 * i + 1];
            point_data[i].point[2] = points[3 * i + 2];
            point_data[i].slope_type = slopeTypes[i];
            point_data[i].slope[0] = slopeVecs[3 * i];
            point_data[i].slope[1] = slopeVecs[3 * i + 1];
            point_data[i].slope[2] = slopeVecs[3 * i + 2];
            point_data[i].crvatr_type = crvatrTypes[i];
            point_data[i].crvatr[0] = crvatrVecs[3 * i];
            point_data[i].crvatr[1] = crvatrVecs[3 * i + 1];
```

```
                point_data[i].crvatr[2] = crvatrVecs[3 * i + 2];
            }
            // создание сплайна
            UF_CURVE_create_spline_thru_pts(degree,
                periodicity,
                num_points,
                point_data,
                parameters,
                save_def_data,
                &spline_tag);
            UF_terminate();
        }
}
int ufusr_ask_unload(void)
{
    return (UF_UNLOAD_IMMEDIATELY);
}
```

Результат выполнения Листинга 2.1 приведен на рисунке 2.3.

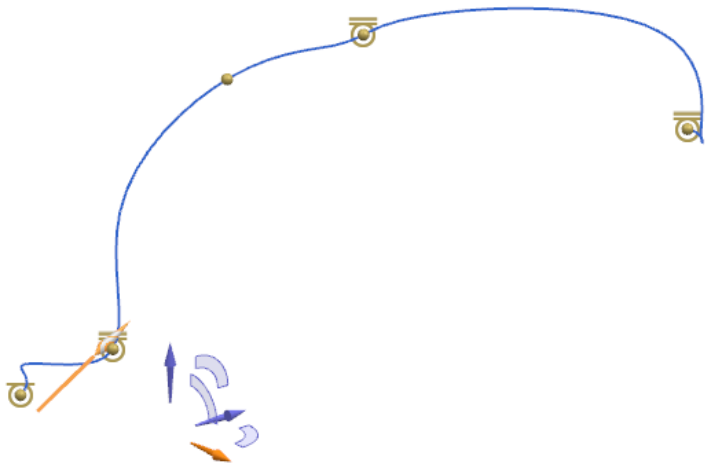


Рисунок 2.3 – Сплайн, полученный в результате выполнения кода, представленного в Листинге 2.1

Рассмотрим еще один пример программного построения сплайна.

Пусть задано пять точек p_1 ... p_5 и требуется программным путем построить через эти точки сплайн. Заданы направления касательных v_1 и v_2 в начальной и конечной точках сплайна.

Согласно рисунку 2.4 координаты точек в формате { *x y z* } будут:

p_1{0,0 0,0 0,0}; p_2{30,0 40,0 0,0}; p_3{200,0 72,0 0,0}; p_4{600,0 55,0 0,0}; p_5{1000,0 0,0 0,0};

Векторы касательных в краевых точках в формате {*I J K*} будут:

v_1={0,0 1,0 0,0}; v_2={1,0 -0,15 0,0};

где вектор *J* = -0.15 получен как тангенс угла наклона касательной в точке p_5:

J= -tg(8.531).

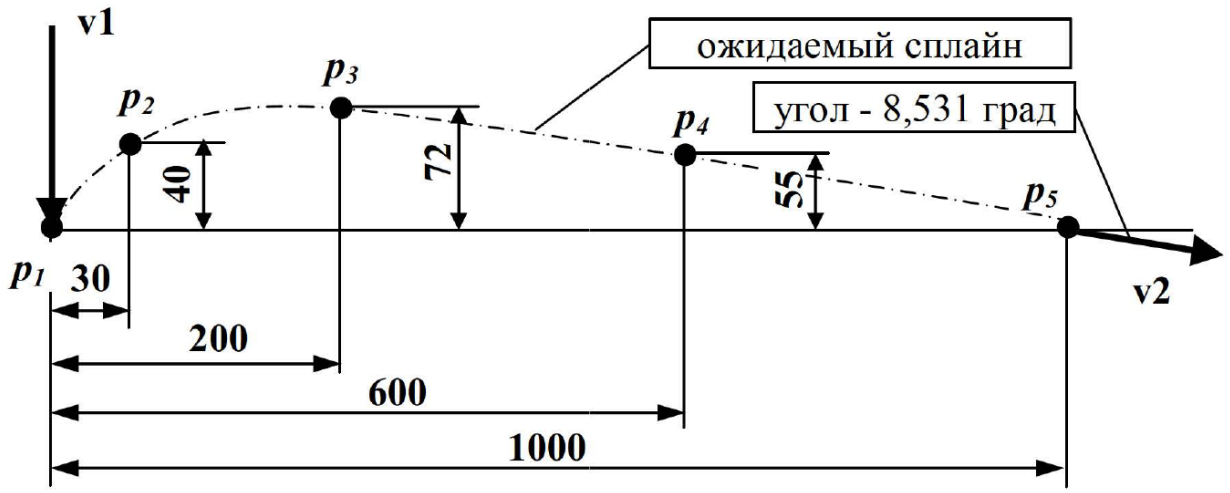


Рисунок 2.4 – Исходные данные для построения сплайна через пять заданных точек

Текст модуля, строящего сплайн через пять заданных точек представлен в листинге 2.2.

Листинг 2.2

```
#include <stdio.h>
#include <uf_curve.h>
#include <uf.h>
#define NUMBER_POINTS 5 // количество точек в сплайне
void ufusr(char *param, int *retcode, int paramLen)
{
```

```
      int degree = 3; // степень сплайна
      int periodicity = 0; // сплайн не периодичный
      int num_points = NUMBER_POINTS;
      int save_def_data = 1;
      tag_t spline_tag;
      // заполнение структуры параметров сплайна
      UF_CURVE_pt_slope_crvatr_t point_data[NUMBER_POINTS]
      {
      { {0.0, 0.0, 0.0}, // координаты первой точки
      UF_CURVE_SLOPE_DIR, /*в ней будет задано направление
      касательной */
      { 0.0, 1.0, 0.0000 }, // вектор касательной
      UF_CURVE_CRVATR_NONE, // кривизна задаваться не будет
      { 0.0000, 0.0000, 0.0000 }
      },
            // в следующих точках задаются только координаты
            { {30, 40, 0.0000},
            UF_CURVE_SLOPE_NONE, {0.0000, 0.0000, 0.0000},
            UF_CURVE_CRVATR_NONE, {0.0000, 0.0000, 0.0000}
            },
            { {200, 72, 0.0000},
            UF_CURVE_SLOPE_NONE, { 0.0000, 0.0000, 0.0000 },
            UF_CURVE_CRVATR_NONE, { 0.0000, 0.0000, 0.0000 }
            },
{ {600, 55, 0.0},
UF_CURVE_CRVATR_NONE, { 0.0000, 0.0000, 0.0000 },
UF_CURVE_CRVATR_NONE, { 0.0000, 0.0000, 0.0000 }
},
// в замыкающей точке, как и в первой, назначается вектор касательной
{ {1000, 0.0, 0.0},
UF_CURVE_SLOPE_DIR,{1.0, -0.15, 0.0},
UF_CURVE_CRVATR_NONE,{0.0, 0.0, 0.0}
}
      };
```

```
    if (!UF_initialize())
    {
        // создание сплайна
        UF_CURVE_create_spline_thru_pts(degree,
            periodicity,
            num_points,
            point_data,
            NULL, //с заданием монотонности по умолчанию
            save_def_data,
            &spline_tag);
        UF_terminate();
    }
}
int ufusr_ask_unload(void)
{
    return (UF_UNLOAD_IMMEDIATELY);
}
```

Рисунок 2.3 – Сплайн, полученный в результате выполнения кода, представленного в Листинге 2.2

2.2 Матрицы в компьютерной графике Open API NX

Матрицы и матричное исчисление - важные компоненты компьютерного моделирования. С помощью матриц происходит определение операций преобразований одних координатных систем в другие, вращение объектов их масштабирование и перемещение.

При работе с матрицами требуется понятие базисных (или единичных) векторов, которые еще называют «ортами». Это вектора, обычно обозначаемые символами *i, j, k*, единичной длины в исходном состоянии совпадают с абсолютной системой координат (рисунок 2.4, а). Если предположить, что базисные вектора жестко закреплены за каким-то объектом, то все перемещения и преобразования объекта могут контролироваться и управляться ориентированием и изменением размеров этих векторов.

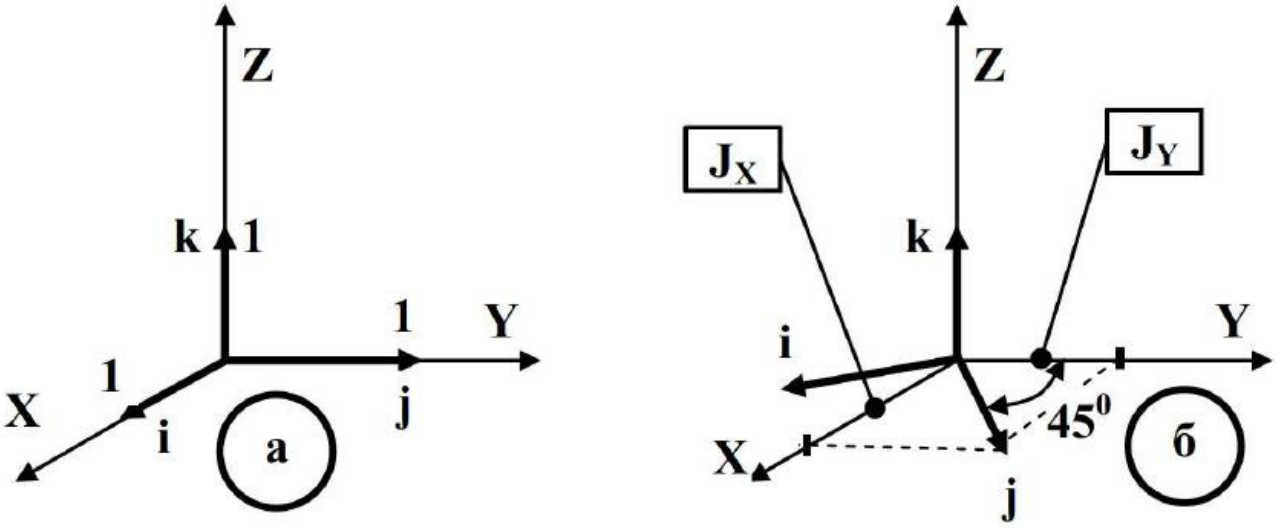


Рисунок 2.4 – Базисные вектора в координатной системе

В математической форме базисные вектора записываются следующим образом:

$$\begin{matrix} i \\ j \end{matrix} = \begin{matrix} I_X & I_Y \\ J_X & J_Y \end{matrix} \text{ – для двумерного пространства,}$$

$$\begin{matrix} i \\ j \\ k \end{matrix} = \begin{matrix} I_X & I_Y & I_Z \\ J_X & J_Y & J_Z \\ K_X & K_Y & K_Z \end{matrix} \text{ – для трехмерного пространства,} \qquad 2.10$$

где N_X, N_Y, N_Z - проекции вектора *N*, соответственно, на оси *X, Y, Z* родительской системы координат (системы, относительно которой производятся преобразования).

Для рисунка 2.4, а базисные вектора примут вид:

$$\begin{matrix} i \\ j \\ k \end{matrix} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}. \qquad 2.11$$

Для рисунка 2.4, б, когда базисные вектора повернуты вокруг оси Z на угол 45°, проекции базисных векторов изменятся и отобразятся следующим образом:

$$\begin{matrix} i \\ j \\ k \end{matrix} = \begin{matrix} \cos 45 & -\sin 45 & 0 \\ \sin 45 & \cos 45 & 0 \\ 0 & 0 & 1 \end{matrix} = \begin{matrix} 0{,}473 & -0{,}473 & 0 \\ 0{,}473 & 0{,}473 & 0 \\ 0 & 0 & 1 \end{matrix}. \qquad 2.12$$

Чтобы некоторая точка абсолютного пространства с координатами (x, y, z) отобразилась в системе повернутых базисных векторов, необходимо рассчитать новые координаты этой точки (x_1, y_1, z_1) по формуле матричного умножения:

$$\begin{matrix} x_1 & y_1 & z_1 \end{matrix} = \begin{matrix} x & y & z \end{matrix} \cdot \begin{matrix} i \\ j \\ k \end{matrix} = \begin{matrix} x & y & z \end{matrix} \cdot \begin{matrix} \cos 45 & -\sin 45 & 0 \\ \sin 45 & \cos 45 & 0 \\ 0 & 0 & 1 \end{matrix}. \qquad 2.13$$

При повороте точки сразу по трем осям (X, Y, Z) (углам Эйлера: α, β, γ) матрица вращения может быть определена по формуле:

$$M(\alpha, \beta, \gamma) = \begin{matrix} \cos\beta\cos\gamma & -\cos\beta\sin\gamma & -\sin\beta \\ \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \sin\gamma\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\cos\beta \\ \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & \sin\alpha\cos\gamma - \cos\alpha\sin\beta\sin\gamma & \cos\alpha\cos\gamma \end{matrix}. \qquad 2.14$$

В Open API NX имеется обширный набор функций по работе с матрицами. Все они описаны в файле uf_mtx.h. В таблице 2.1 перечислены эти функции и дано краткое пояснение по их назначению.

Таблица 2.1 – Функции Open API NX для работы с матрицами

№	Название функции	Назначение функции
1	2	3
Функции для работы с матрицами 2×2		
1	UF_MTX2_copy	Копирование элементов матрицы
2	UF_MTX2_determinant	Вычисление детерминанта
3	UF_MTX2_identity	Создает базовую (единичную)
4	UF_MTX2_initialize	Заполнение элементов матрицы из параметров векторов
5	UF_MTX2_multiply	Умножение матриц 2×2
6	UF_MTX2_multiply_t	Умножение матриц 2×2, при этом первая матрица предварительно транспонируется
7	UF_MTX2_transpose	Транспонирование матрицы
8	UF_MTX2_vec_multiply	Умножение 2D вектора на 2×2 матрицу
9	UF_MTX2_vec_multiply_t	Умножение 2D вектора на 2×2 матрицу, предварительно транспонированную.
10	UF_MTX2_x_vec	Извлекает X-вектор из 2×2 матрицы
11	UF_MTX2_y_vec	Извлекает Y-вектор из 2×2 матрицы
Функции для работы с матрицами 3×3		
12	UF_MTX3_copy	Копирование элементов матрицы 3×3
13	UF_MTX3_determinant	Вычисление детерминанта матрицы 3×3
14	UF_MTX3_ identity	Создает базовую (единичную) 3×3 матрицу
15	UF_MTX3_initialize	Заполнение элементов матрицы из параметров векторов X и Y. Вектор Z достраивается перпендикулярно заданным векторам.
16	UF_MTX3_initialize_x	Создает новую матрицу 3×3 и инициализирует в ней строку X элементов по значениям заданного вектора.

Продолжение таблицы 2.1

1	2	3
17	UF_MTX3_initialize_z	Создает новую матрицу 3×3 и инициализирует в ней строку Z элементов по значениям заданного вектора.
18	UF_MTX3_mtx4	Конвертирует 3×3 матрицу в 4×4 с коэффициентом масштабирование равным 1 и нулевым вектором трансляции (перемещения).
19	UF_MTX 3_ multiply	Умножение матриц 3×3
20	UF_MTX3_multiply_t	Умножение матриц 3×3, при этом первая матрица предварительно транспонируется
21	UF_MTX3_ortho_normalize	Преобразует поданную на вход 3×3матрицу в матрицу с ортогональными векторами
22	UF_MTX3_rotate_about_axis	Создает матрицу вращения для указанного вектора оси и заданного угла
23	UF_MTX3_transpose	Транспонирование матрицы
24	UF_MTX 3_vec_ multiply	Умножение 3D вектора на 3×3матрицу
25	UF_MTX 3_vec_ multiply _t	Умножение 3D вектора на 3×3 матрицу, предварительно транспонированную
26	UF_MTX3_x_vec	Извлекает X-вектор из 3×3 матрицы
27	UF_MTX3_y_vec	Извлекает Y-вектор из 3×3 матрицы
28	UF_MTX3_z_vec	Извлекает Z-вектор из 3×3 матрицы
Функции для работы с матрицами 4×4		
29	UF_MTX4_ask_rotation	Возвращает 3×3 матрицу вращения, извлеченную из 4×4матрицы
30	UF_MTX4_ask_scale	Извлекает масштабный коэффициент из 4×4 матрицы

Продолжение таблицы 2.1

1	2	3
31	UF_MTX4_ask_translation	Извлекает вектор перемещения из 4×4 матрицы
32	UF_MTX4_copy	Копирование элементов матрицы
33	UF_MTX4_edit_rotation	Редактирование матричной компоненты вращения в матрице 4×4 по заданной матрице вращения 3×3
34	UF_MTX4_edit_scale	Редактирование фактора масштабирования в матрице 4×4 по заданной величине
35	UF_MTX4_edit_translation	Редактирование матричной компоненты перемещения в матрице 4×4 по заданному 3D вектору
36	UF_MTX4_identity	Создает базовую (единичную) 4×4 матрицу
37	UF_MTX4_initialize	Заполнение элементов матрицы из параметров векторов
38	UF_MTX 4_ multiply	Умножение матриц 4×4
39	UF_MTX4_multiply_t	Умножение матриц 4×4, при этом первая матрица предварительно транспонируется
40	UF_MTX4_ortho_normalize	Преобразует поданную на вход 4×4 матрицу в матрицу с ортогональными векторами
41	UF_MTX4_transpose	Транспонирование матрицы 4×4
42	UF_MTX4_vec3_multiply	Умножение 3D вектора на 4×4 транспонированную матрицу (при умножении вектор расширяется до 4D вектора добавлением 1)
43	UF_MTX4_x_vec	Извлекает X-вектор из 4×4 матрицы
44	UF_MTX4_y_vec	Извлекает Y-вектор из 4×4 матрицы
45	UF_MTX4_z_vec	Извлекает Z-вектор из 4×4 матрицы

Продолжение таблицы 2.1

1	2	3
46	UF_MTX4_scaling	Создает матрицу для выполнения указанного масштабирования
47	UF_MTX4_rotation	Создает матрицу вращения вокруг заданной точки
48	UF_MTX4_mirror	Создает матрицу для задания средней плоскости объекта
49	UF_MTX4_csys_to_csys	Создает матрицу, которая может быть использована для отображения объектов из одной координатной системы в другую
50	UF_MTX4_invert	Создает матрицу, инвертированную по отношению к заданной

Как видно из таблицы, в работе задействованы матрицы с количеством элементов 2×2 (для двумерного моделирования), 3×3 - для определения вращения трехмерных объектов (как было рассмотрено выше) и 4×4 - для выполнения вращения трехмерных объектов с одновременным их масштабированием и перемещением.

С помощью матриц 3×3, кроме вращения, можно еще осуществить масштабирование, если изменять длину базисных векторов. Масштабирующая матрица выглядит следующим образом:

$$\begin{matrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{matrix}, \qquad 2.15$$

где S_1, S_2, S_3 – коэффициенты масштабирования.

Если $S < 1$, значит происходит уменьшение объектов, если $S > 1$, происходит увеличение объектов.

Чтобы к операциям вращения и масштабирования добавить операцию перемещения, требуется расширить матрицу преобразования до размера 4×4. Вектор пе-

ремещения записывается в последней строке матрицы, а дополнительная колонка заполняется нулями:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ V_X & V_Y & V_Z & 1 \end{matrix}, \qquad 2.16$$

где V_x, V_y, V_z - перемещения по соответствующим координатам.

Общая (полная) матрица преобразований, объединяющая в себе и вращение и масштабирование и перемещение получается перемножением дополнений матрицы вращения и матрицы перемещения и имеет результирующий вид:

$$\begin{matrix} I_X & I_Y & I_Z & 0 \\ J_X & J_Y & J_Z & 0 \\ K_X & K_Y & K_Z & 0 \\ V_X & V_Y & V_Z & 1 \end{matrix}, \qquad 2.17$$

В качестве практических упражнений по ознакомлению с использованием матриц в NX можно провести ряд вычислений по сравнению результатов работы матричных функций NX и теоретических формул.

Например, сравним результаты поворотов осей координат на угол 45° вокруг осей *X* и *Y*, выполненных через матрицы вращения, посчитанных функциями NX из таблицы 2.1 и по формуле (2.14) на основе кода построения окружности, приведенного в листинге 1.3.

Для поворота осей координат на заданные углы первым методом можно использовать код, приведенный в листинге 2.3.

Листинг 2.3.

```
#include   <uf.h>          //файл описаний общих функций
#include   <uf_curve.h>    //файл описаний функций кривых
#include   <uf_csys.h>     //файл описаний функций работы с координатами
#include   <uf_mtx.h> //файл описаний функций работы с матрицами
void  ufusr(char   *param, int   *retcode, int  paramLen)
```

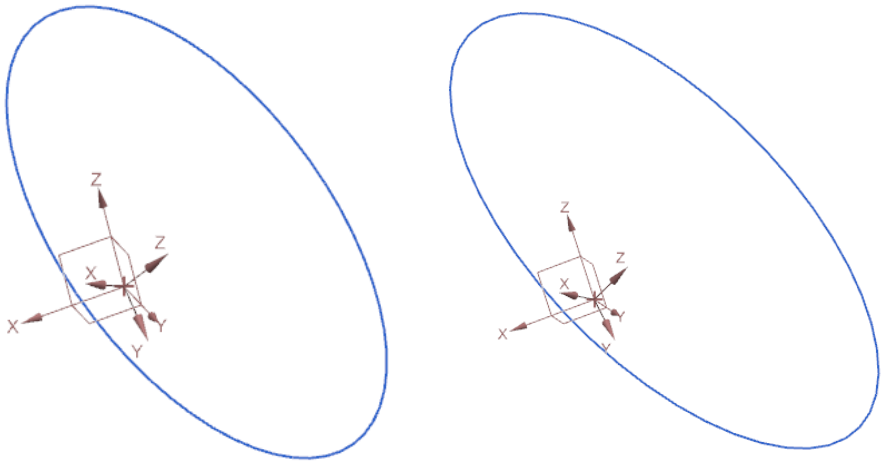
```
{
    // данные для окружности
    tag_t     arc_id, wcs_tag;        //тэги окружности
                                      //и мировой  системы координат
    UF_CURVE_arc_t  arc_coords; //структура свойств дуги
    // объявление данных для матрицы поворота
    //зададим угол 45° и пересчитаем его в радианы
    double ugol_Y = 45 * DEGRA;
    // объявим вектора осей X и У
    double vec_X[3] = { 1, 0, 0 }, vec_Y[3] = { 0, 1, 0 };
    //объявим матрицы поворотов:
    double     mtxP[9], //вокруг оси X
               mtyP[9], //вокруг оси У
               mt[9]; //суммарную матрицу поворота
         //объявим теги систем координат
    tag_t teg_wcs, csys_id;
    // объявим точку начала координат
    double center[3] = { 0,0,0 };
    //если активизация функций Open API не прошла - прервать программу
    if (UF_initialize())  return;
    // начало кода работы с матрицами
   // получим матрицу поворота на 45° вокруг оси У
    UF_MTX3_rotate_about_axis(vec_Y, ugol_Y, mtyP);
    // получим матрицу поворота на 45 вокруг оси X
    UF_MTX3_rotate_about_axis(vec_X, ugol_Y, mtxP);
    // перемножением получим суммарную матрицу поворота по обоим углам
    UF_MTX3_multiply(mtxP, mtyP, mt);
    // создадим тег суммарной матрицы поворота
    UF_CSYS_create_matrix(mt, &teg_wcs);
    // создадим тег системы координат на базе полученной матрицы
    UF_CSYS_create_csys(center, teg_wcs, &csys_id);
    // установим на экране созданную рабочую систему координат
    UF_CSYS_set_wcs(csys_id);
    // начало кода построения окружности
```

```
    arc_coords.start_angle = 0.0; //начальный угол окружности
    arc_coords.end_angle = 360.0   *   DEGRA;   //конечный угол
    arc_coords.arc_center[0] = 0.0;    //координата центра X
    arc_coords.arc_center[1] = 0.0; //координата центра Y
    arc_coords.arc_center[2] = 20.0; //координата центра Z
    arc_coords.radius = 50.0;   //радиус окружности
    UF_CSYS_ask_wcs(&wcs_tag); //получение абсолютных координат
    //"перенос" абсолютных координат на создаваемую окружность
    UF_CSYS_ask_matrix_of_object(wcs_tag, &arc_coords.matrix_tag);
    //построение окружности
    UF_CURVE_create_arc(&arc_coords, &arc_id);
    UF_terminate();
}
int   ufusr_ask_unload(void) //функция выхода из приложения
{
    return    (UF_UNLOAD_IMMEDIATELY);
}
```

Результат выполнения библиотеки приведен га рисунке 2.5, а. В примере показано использование функций NX работы с матрицами, в частности:

`UF_MTX3_rotate_about_axis ( ... )` - получение матрицы вращения вокруг одной оси;

`UF_MTX3_multiply ( ... )` - умножение матриц.



а б

Рисунок 2.5 – Окружность после поворота на 45° вокруг осей *X* и *Y*

Вторая функция необходима для того, чтобы из двух матриц вращения по разным осям получить суммарную матрицу вращения. Как известно, общая матрица вращения получается перемножением матриц вращения по отдельным осям.

Причем важен порядок перемножения:

$$M = M_X \times M_Y \times M_Z, \qquad 2.18$$

где M - общая матрица вращения;

M_N - матрицы вращения вокруг осей: $N = \{X\}, \{Y\}, \{Z\}$.

Для поворота осей координат вторым методом (на основе теоретических формул), воспользуемся кодом, приведенным в листинге 2.4, который реализует формулу (2.14).

Листинг 2.3.

```
#include   <uf.h>          //файл описаний общих функций
#include   <uf_curve.h>    //файл описаний функций кривых
#include   <uf_csys.h>     //файл описаний функций работы с координатами
#include   <uf_mtx.h> //файл описаний функций работы с матрицами
#include   <math.h>
void  ufusr(char   *param, int   *retcode, int  paramLen)
{
     // данные для окружности
     tag_t     arc_id, wcs_tag;        //тэги окружности
                                       //и мировой  системы координат
     UF_CURVE_arc_t  arc_coords; //структура свойств дуги
     // объявление данных для матрицы поворота
     double mt[9]; //суммарная матрица вращения
     double mt_t[9]; //транспонированная суммарная матрица вращения
     tag_t teg_wcs, csys_id;
     double center[3] = { 0,0,0 },
          vec_X[3],
          vec_Y[3],
          vec_Z[3];
```

```
    //углы поворота по осям X и У
    double aX = 45 * DEGRA, aY = 45 * DEGRA, aZ = 0;
    //если активизация функций Open API не прошла - прервать программу
    if (UF_initialize())  return;
    // начало кода работы с матрицами
    // задание вектора X по формуле
    vec_X[0] = cos(aY)*cos(aZ);
    vec_X[1] = -cos(aY)*sin(aZ);
    vec_X[2] = -sin(aY);
    // задание вектора У по формуле
    vec_Y[0] = -cos(aX)*sin(aZ) + sin(aX)*sin(aY)*cos(aZ) ;
    vec_Y[1] = sin(aX)*sin(aY)*sin(aZ) + cos(aX)*cos(aZ);
    vec_Y[2] = sin(aX)*cos(aY);
    // инициализация матрицы вращения(третий вектор(Z) функция строит сама)
    UF_MTX3_initialize(vec_X, vec_Y, mt);
    UF_MTX3_transpose(mt, mt_t);
    // построение системы координат на основе матрицы вращения
    UF_CSYS_create_matrix(mt_t, &teg_wcs);
    UF_CSYS_create_csys(center, teg_wcs, &csys_id);
    UF_CSYS_set_wcs(csys_id);
    // начало кода построения окружности
    arc_coords.start_angle = 0.0; //начальный угол окружности
    arc_coords.end_angle = 360.0   *   DEGRA;   //конечный угол
    arc_coords.arc_center[0] = 0.0;    //координата центра X
    arc_coords.arc_center[1] = 0.0; //координата центра Y
    arc_coords.arc_center[2] = 20.0; //координата центра Z
    arc_coords.radius = 50.0;   //радиус окружности
    UF_CSYS_ask_wcs(&wcs_tag); //получение абсолютных координат
    //"перенос" абсолютных координат на создаваемую окружность
    UF_CSYS_ask_matrix_of_object(wcs_tag, &arc_coords.matrix_tag);
    //построение окружности
    UF_CURVE_create_arc(&arc_coords, &arc_id);
    UF_terminate();
}
```

```
int   ufusr_ask_unload(void) //функция выхода из приложения
{
      return    (UF_UNLOAD_IMMEDIATELY);
}
```

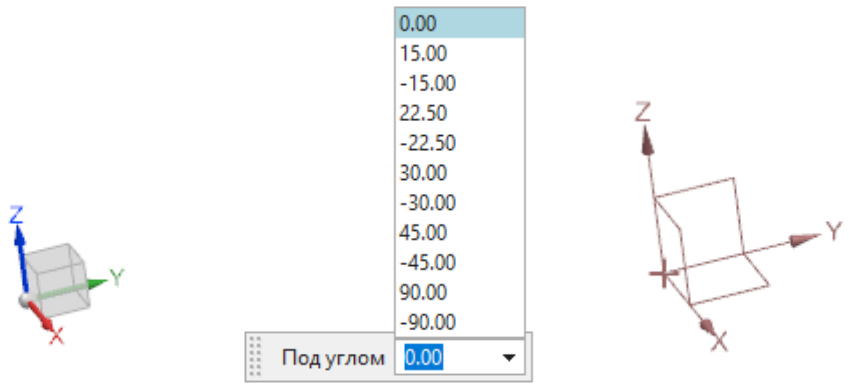
Этот код создает поворот системы координат (рисунок 2.5, б) в точности соответствующий рисунку 2.5, а.

Как правило, матрицы работают в совокупности с такими элементами сцены, как координатные оси, плоскости, вектора. При этом в функциях фигурируют не сами матрицы, а их теги - абстрактные указатели, используемые в качестве параметров функций Open API. Рассмотрим их с помощью библиотеки координатных функций NX.

2.3 Координатные функции NX Open API

В NX присутствует несколько систем координат.

Абсолютная система координат (ACS – рисунок 2.6, а) - система, в которой выполняются все построения рабочей сцены. Абсолютная система координат едина и неизменна для всех элементов всех моделей. Направление обзора ACS можно, по желанию, настраивать, поворачивая его вокруг любой из координатных осей ACS (2, рисунок 2.6, б).



а б в

Рисунок 2.6 – Системы координат NX

Рабочая система координат (WCS – рисунок 2.6, в) - система, в которой выполняются текущие построения. В начальный момент времени обе системы совпадают, но проектировщик в любой момент может изменить положение WCS для удобства выполняемых построений.

Программист, же, как правило (исключение - окружность и спираль), при создании или редактировании объекта, в качестве координатной системы использует только ACS.

Программным путем можно еще создавать «местные» координатные системы (CSYS) - привязанные к определенному месту 3D-модели и «временные» (temporary) координатные системы - как промежуточный элемент при выполнении операций построения 3D-модели. Таких координатных систем в модели может быть много, но рабочая система координат, относительно которой идут текущие построения, всегда одна.

Для работы с системами координат в NX Open API имеется набор функций, описанных в файле uf_csys.h. Перечень и краткое описание назначения этих функций представлены в таблице 2.2.

Таблица 2.2 – Функции Open API NX для работы с координатными системами

№	Название функции	Назначение функции
1	2	3
1	UF_CSYS_map_point	Переносит точку из одной системы координат в другую
2	UF_CSYS_create_matrix	Создает тег матрицы вращения
3	UF_CSYS_ask_matrix_values	Получает компоненты матрицы вращения по ее тегу
4	UF_CSYS_ask_matrix_of_object	Получает матрицу вращения заданного объекта в рабочей системе координат
5	UF_CSYS_create_csys	Создает местную систему координат
6	UF_CSYS_create_temp_csys	Создает временную систему координат

Продолжение таблицы 2.2

1	2	3
7	UF_CSYS_set_wcs	Устанавливает рабочую систему координат в указанную местную систему координат
8	UF_CSYS_ask_wcs	Получает матрицу вращения рабочей системы координат
9	UF_CSYS_ask_csys_info	Получает информацию об указанной местной системе координат
10	UF_CSYS_edit_matrix_of_object	Изменяет (редактирует) матрицу вращения указанного объекта
11	UF_CSYS_set_origin	Задает начальную точку для указанной системы координат
12	UF_CSYS_set_wcs_display	Отображает (или скрывает) на экране рабочую систему координат

Для знакомства с применением координатных функций рассмотрим приме, в котором создается местная система координат в указанном пользователем месте, а затем выполняется построение окружности (Листинг 2.4).

Листинг 2.4.

```
#include <stdio.h>
#include <uf_curve.h>
#include <uf.h>
#include <uf_mtx.h>
#include <uf_csys.h>
#include <uf_ui.h>
#include <uf_obj.h>
char msg[128] = "Задайте плоскость для создания окружности";
void ufusr(char *param, int *retcode, int param_len)
{
    if (!UF_initialize())
    {
```

```
int mode = 5, resp;
tag_t mtx_id, csys_id, arc_id, plane_tag, saved_wcs;
double plane_matrix[9], plane_origin[3];
UF_CURVE_arc_t arc;
arc.start_angle = 0.0;
arc.end_angle = TWOPI;
arc.radius = 50;
// запуск диалога создания плоскости
UF_UI_specify_plane(
      msg,
      &mode,
      1,
      &resp,
      plane_matrix,
      plane_origin,
      &plane_tag);
//если на диалоге нажата кнопка ОК
if (resp == 3)
{
      UF_CSYS_ask_wcs(&saved_wcs);
// если плоскость построена не в режиме «абсолютные координаты»
      if (mode != 5)
      {
            UF_CSYS_create_matrix(plane_matrix, &mtx_id);
            UF_CSYS_create_csys(plane_origin, mtx_id, &csys_id);
            UF_CSYS_set_wcs(csys_id);
      }
      arc.matrix_tag = mtx_id;
UF_MTX3_vec_multiply(plane_origin, plane_matrix, arc.arc_center);
      arc.arc_center[0] += 30.0;
      arc.arc_center[1] += 25.0;
      UF_CURVE_create_arc(&arc, &arc_id);
// если плоскость построена не в режиме «абсолютные координаты»
      if (mode != 5)
```

```
                {
                        UF_CSYS_set_wcs(saved_wcs);
                        UF_OBJ_delete_object(csys_id);
                }
        }
        UF_terminate();
    }
}
int ufusr_ask_unload(void)
{
    return (UF_UNLOAD_IMMEDIATELY);
    }
```

Для начала необходимо запомнить исходное состояние рабочей системы координат сцены с помощью функции:

```
UF_CSYS_ask_wcs (&saved_wcs);
```

где параметр функции - заранее описанная ячейка (`tag_t saved_wcs`) тэга системы координат.

Сохранение исходного тега рабочих координат нужно для того, чтобы после выполнения действий с системами координат в программе, вернуть все на прежнее место.

Следующий шаг - это перенос рабочей системы координат на созданную плоскость. Для этого сначала необходимо получить тег матрицы вращения созданной плоскости. В листинге 2.4 эта матрица хранится в переменной `plane_matrix`. Тег получаем функцией:

```
UF_CSYS_create_matrix(plane_matrix, &mtx_id).
```

Второй параметр функции описан в листинге как `tag_t mtx_id`.

Теперь следует создать местную систему координат в созданной плоскости в указанной опорной точке и с указанной матрицей поворота:

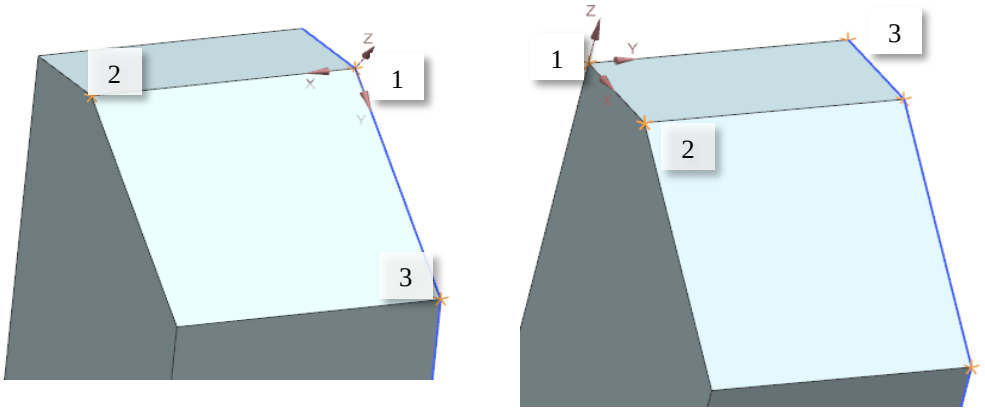
```
UF_CSYS_create_csys(plane_origin, mtx_id,&csys_id).
```

Опорную точку (`plane_origin`) для этой функции взята из диалога создания плоскости `UF_UI_specify_plane`, а третий параметр – это тег местной системы координат, описанный как `tag_t csys_id`.

Заключительное действие - перемещение рабочей системы координат сцены в точку местной системы координат выполняет функция:

`UF_CSYS_set_wcs(csys_id).`

Для визуализации работы описанного блока программного модуля, на экране удобно создать какое-либо тело, например блок (параллелепипед) со скошенной гранью (рисунок 2.7).



а б

Рисунок 2.7 – Пример перемещения рабочей системы координат сцены в точку местной системы координат

Затем необходимо построить через его вершины плоскость (на рисунке 2.7 - точки 1, 2, 3) с опорной точкой 1 и после отработки описанных команд можно наблюдать, как орты рабочей системы координат переместились в опорную точку, а вектора *x*, *y* расположились в созданной плоскости.

На рисунке 2.7 приведено два варианта построения плоскости: плоскость на скошенной грани (рисунок 2.7, а) и плоскость на верхней грани (рисунок 2.7, б). Направление векторов координатных осей (после отработки программы) зависит от порядка обхода задающих точек 1, 2, 3 при создании плоскости.

Теперь необходимо привязать к плоскости будущую кривую. По заданию требуется создание окружности. Создадим окружность с помощью функции `UF_CURVE_create_arc ( ... )`, рассмотренной в п. 1.5.1.

Положим, что нам надо создать полную окружность с радиусом 50 мм и центром с координатами x1=30 мм, y1 =25 мм, z1=0 мм в рабочей системе координат созданной плоскости. Если мы обозначим `UF _ CURVE _ arc_t arc`, то три поля из этой структуры мы можем задать в любой момент программы:

```
arc.start_angle = 0.0; //полная окружность
arc.end_angle = TWOPI; //угол в два PI
arc.radius = 50; //и радиус
```

Тэг матрицы поворота можно получить из матрицы поворота созданной плоскости:

```
UF_CSYS_create_matrix(plane_matrix, &mtx_id);
```

хотя эту величину мы уже получали ранее в виде `mtx_id`, поэтому вместо использования функции для задания тега матрицы вращения окружности можно сразу написать: `arc.matrix_tag=mtx_id`;

Координаты центра окружности можно получить, развернув через матрицу поворота координаты опорной точки плоскости:

```
UF_MTX3_vec_multiply(plane_origin, plane_matrix, arc.arc_center);
```

получим центр рабочей системы координат плоскости и добавим к нему заданные x_1, y_1, z_1:

```
arc.arc_center[0] += 30.0;
arc.arc_center[1] += 25.0.
```

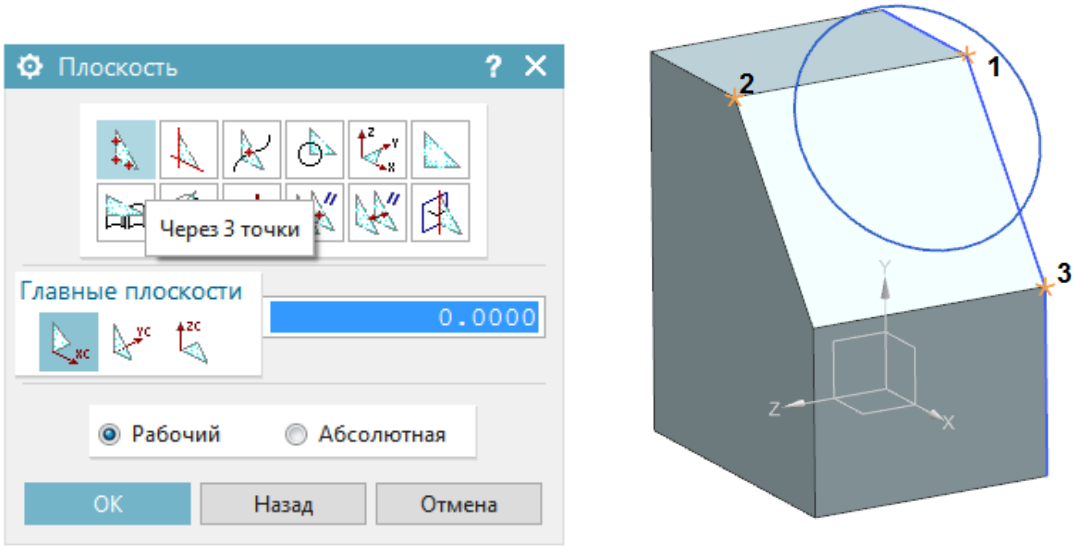
Завершающими командами модуля будет восстановление рабочей системы координат сцены в исходное состояние:

```
UF_CSYS_set_wcs(saved_wcs);
```

и удаление, ставшим ненужным, объекта местной системы координат:

```
UF_OBJ_delete_object(csys_id).
```

Для проверки работы библиотеки рекомендуется создать фигуру, как показано на рисунке 2.7. После запуска библиотеки в NX появится диалоговое окно (рисунок 2.8) в котором следует выбрать способ построения «Через 3 точки» и последовательно выбрать точки 1, 2, 3, рисунок 2.8, б. Должна построиться окружность, как показано на рисунке 2.8, б.



а б

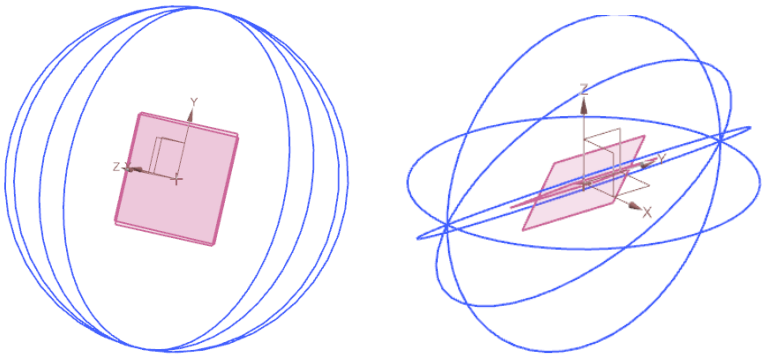
Рисунок 2.8 – Результат выполнения листинга 2.4

2.4 Задание для лабораторной работы

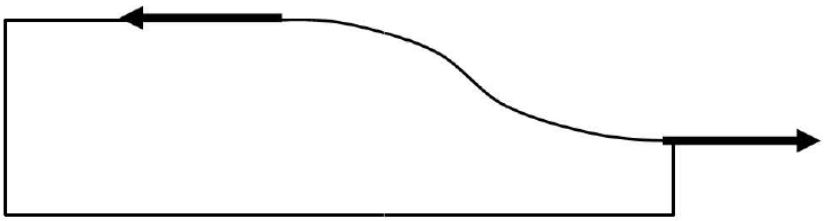
1 Изучить теоретический материал.

2 Разработать прикладную библиотеку NX для заданного варианта.

Вариант 1. Разработать модуль, строящий в 3D сцене NX каркас шара из четырех образующих, состоящих из окружностей, расположенных в плоскостях с шагом в 60 градусов.



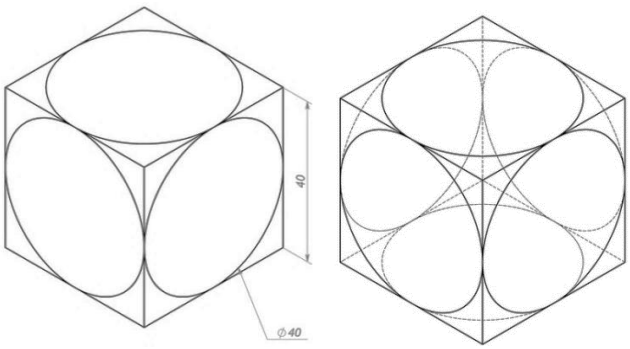
Вариант 2. Разработать модуль, строящий контур молотка из четырёх линий и одного сплайна. Векторы касательных на концах сплайна показаны на эскизе. Размеры молотка задать самостоятельно.



Вариант 3. Разработать модуль, строящий систему координат, повернутую вокруг каждой из координатных осей на 45°, и в ней окружность (произвольного диаметра). К окружности добавить осевую линию, проходящую через центр окружности перпендикулярно её плоскости.

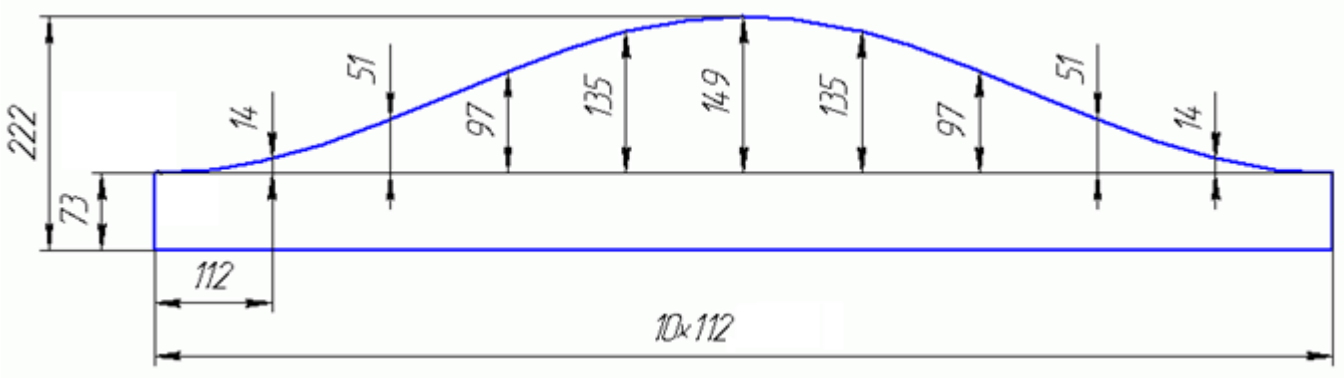
Вариант 4. Разработать модуль, строящий контур из линий, окружностей и сплайнов, формирующий законченное изображение по своему усмотрению с использованием матриц поворота или смещения системы координат.

Вариант 5. Разработать модуль, строящий куб со вписанными на его гранях окружностями.



Вариант 6. Изменить листинг 2.2, так, чтобы сплайн строился в отрицательных значениях оси Y (ниже оси X) и под углом 30 градусов к плоскости XOZ.

Вариант 7. Разработать модуль построения фигуры в соответствии с рисунком.



Вариант 8. Разработать модуль, строящий куб, на гранях которого изображены сплайны (достаточно трех точек).

Вариант 9. Разработать модуль, строящий изображение из задания, приведенного в разделе 1 для своего варианта, на заданной грани трехмерного объекта, как приведено в примере на рисунке 2.8.

Вариант 10. Разработать модуль построения фигуры в соответствии с рисунком для варианта 7, но в отрицательных значениях оси X и под углом минус 30 градусов к плоскости XOZ.

Вариант 11. Разработать модуль, строящий контур молотка в соответствии с рисунком. Векторы касательных на концах сплайна показаны на эскизе. Размеры молотка задать самостоятельно.

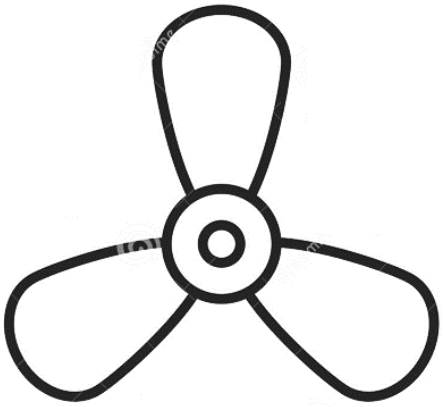
Вариант 12. Разработать модуль построения фигуры в соответствии с рисунком для варианта 7 на заданной грани трехмерного объекта, как приведено в примере на рисунке 2.8.

Вариант 13. Разработать модуль построения винта в соответствии с рисунком, размеры элементов и касательные для сплайна задать самостоятельно.

Вариант 14. Разработать модуль построения винта в соответствии с рисунком для варианта 13, но лопасти которого располагаются под углом 90 градусов к плоскости окружностей.

Вариант 15. Разработать модуль, строящий систему координат, повернутую вокруг каждой из координатных осей на 45°, и в ней окружность (произвольного диаметра). К окружности добавить сплайн, согласно Листингу 2.2, берущий свое начало из центра окружности перпендикулярно её плоскости.

Вариант 16. Разработать модуль построения винта в соответствии с рисунком, размеры элементов и касательные для сплайна задать самостоятельно.



4 Подготовить ответы на контрольные вопросы.

2.5 Содержание отчета

В отчете по лабораторной работе согласно СТО 02069024. 101 – 2015 [5] должны содержаться следующие пункты:

- название лабораторной работы;
- цель работы;
- задание на лабораторную работу;
- код разработанной программы с комментариями;

– экранные формы с отображением результата выполнения программы в системе NX;

– выводы;

– список использованных источников.

2.6 Контрольные вопросы

1) Что такое сплайн, кривая Безье, Б-сплайн, С-сплайн?
2) Функция для построения прямой линии в NX для языка С.
3) Два способа построения сплайна в Open API NX.
4) Периодичный и не периодичный сплайны.
5) Функция `UF_CURVE_create_spline_thru_pts`.
6) Функция `UF_CURVE_create_spline`.
7) Область применения матриц в компьютерном моделировании.
8) Понятие базисных векторов.
9) Назначение матриц размерностью 2x2, 3x3, 4x4.
10) Функции для работы с матрицами 2x2.
11) Функции для работы с матрицами 3x3.
12) Функции для работы с матрицами 4x4.
13) Функция матриц вращения вокруг одной оси.
14) Функция умножения матриц.
15) Типы систем координат в NX.
16) Функции Open API NX для работы с координатными системами.

3 Моделирование объектов и действий над ними функциями Open NX

3.1 Цилиндр

Чтобы в NX смоделировать какой-либо типовой объект, например, цилиндр (параллелепипед, шар, конус) достаточно одной функции. Для цилиндра это функция `UF_MODL_create_cyl1 ( ... ).` У функции шесть параметров.

На входе в функцию:

- первый параметр:

- `UF_FEATURE_SIGN sign`, определяет тип булевой операции, которая должна произойти с создаваемым телом, задается предопределенными константами:

- `UF_NULLSIGN` - если создается новое твердое тело;

- `UF_POSITIVE` - если создаваемое тело объединяется с уже существующим;

- `UF_NEGATIVE` - если создаваемое тело вычитается из существующего;

- `UF_UNSIGNED` - если создаваемое тело пересекается с существующим;

- второй параметр:

- `double origin[3]`, задает координаты точки привязки центра окружности основания цилиндра;

- третий параметр:

- `char* height`, указатель на текстовую строку, где прописана высота цилиндра;

- четвертый параметр:

- `char* diam`, указатель на текстовую строку с диаметром выстраиваемого цилиндра;

- пятый параметр:

- `double direction[3]`, задает вектор направления главной оси цилиндра.

На выходе функции всего один параметр:

- `tag_t* cyl_obj_id`, тег вновь созданного цилиндра.

Для опробования функции создадим небольшой программный модуль, который должен создать цилиндр, который должен опираться на точку с координатами *X*=5 мм, *Y*=10 мм и *Z*=15 мм, а направление оси цилиндра должно совпадать с главной диагональю координатного пространства системы. Исходный код построения цилиндра приведен в листинге 3.1.

Листинг 3.1.

```
//Программа построения цилиндра
#include <stdio.h>
#include <uf.h>
#include <uf_modl.h>
void ufusr(char *param, int *retcode, int paramLen) {
	// определим создание нового самостоятельного тела
	UF_FEATURE_SIGN sign = UF_NULLSIGN;
	//это тело должно начинаться в заданных координатах X, У, Z
	double cyl_orig[3] = { 5.0,10.0,15.0 };
	char height[125] = "70"; //высоту цилиндра зададим в 70 мм
	char diam[125] = "50"; //диаметр-в 50 мм
	//и орты направления главной диагонали координат будут по 1
	double direction[3] = { 1.0, 1.0, 1.0 };
	tag_t cyl_obj; //определим переменную для будущего тега цилиндра
	if (!UF_initialize()) {
		//выполним построение цилиндра
		UF_MODL_create_cyl1(sign, cyl_orig, height, diam, direction,
&cyl_obj);
		UF_terminate();
	}
} //типовая функция завершения приложения
int ufusr_ask_unload(void) {
	return (UF_UNLOAD_IMMEDIATELY);
	}
```

В результате выполнения программы, приведенной в Листинге 3.1 будет построен цилиндр в соответствии с рисунком 3.1.

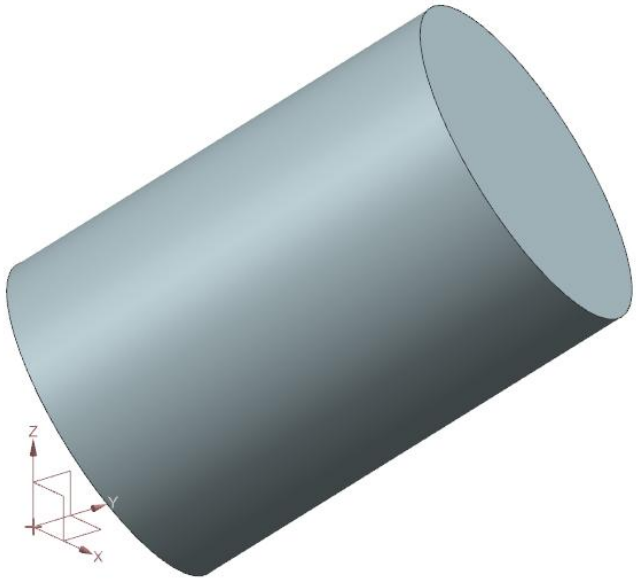


Рисунок 3.1 – Построение цилиндра

3.2 Тело вращения

Более половины деталей машиностроения являются телами вращения. Для построения моделей таких деталей удобно использовать операцию вращения. Создадим программный модуль, формирующий тело вращения из сечения, представленного на рисунке 3.2. Вектор вращения будет совпадать с осью *Y*.

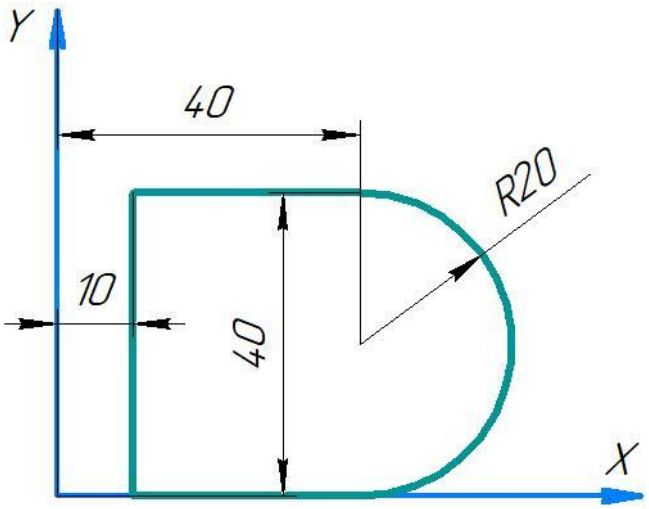


Рисунок 3.2 – Размеры и расположение тела вращения

Для программирования операции используем функцию:

```
int UF_MODL_create_revolved(
uf_list_p_t obj_id_list, // перечень объектов вращения
char ** limit, //пределы вращения
double point [3], //точка начала вектора вращения
double direction [3], //направление вектора вращения
UF_FEATURE_SIGN sign, /*логические операции при создании тела вращения*/
uf_list_p_t *feature_list //перечень созданных объектов
)
```

Такие операции как вытягивание, вращение и другие на программном уровне проводятся с набором кривых, предварительно занесённых в специальный массив типа `uf_list_p_t` - перечень объектов.

Перечень объектов предварительно создается функцией `UF_MODL_create_list(uf_list_p_t list)`, а затем заполняется функцией:

```
int UF_MODL_put_list_item(
uf_list_p_t list,// указатель на заполняемый перечень
tag_t obj_id). // тег объекта, заносимого в перечень
```

Пределы вращения в функции `UF_MODL_create_revolved` - это массив из двух строк, в которых в текстовом виде заносятся значения начального и конечного угла вращения сечения в градусах.

Точка начала вектора вращения и его направление - два стандартных массива по три элемента. Объединение `UF_FEATURE_SIGN`, это новый элемент программирования. Объединение содержит список констант, определяющих логическую операцию с окружающими тело объектами в момент его создания. Константы следующие:

- `UF_NULLSIGN` - создается новое самостоятельное тело;
- `UF_POSITIVE` - тело объединяется с пересекаемыми телами;
- `UF_NEGATIVE` - тело вычитается из пересекаемых тел;
- `UF_UNSIGNED` - создается тело пересечения создаваемого тела с имеющимися.

Исходный код построения тела вращения, для сечения, представленного на рисунке 3.2, приведен в листинге 3.2. Начальный угол вращения – 0^0, конечный угол вращения - 270^0.

Листинг 3.2.

```
//Программа построения тела вращения
#include <uf.h>
#include <uf_curve.h>
#include <uf_csys.h>
#include <uf_modl.h>
char a[] = "0.0";
char b[] = "270.0";
void ufusr(char *param, int *retcode, int param_len) {
	/* Определяется массив линий с координатами их
	концов, используемых в эскизе вращаемого сечения */
	UF_CURVE_line_t line[3] = {
		{{10.0,0.0,0.0}, {10.0,40.0,0.0}},
		{{10.0,0.0,0.0}, {40.0,0.0,0.0}},
		{{10.0,40.0,0.0}, {40.0,40.0,0.0}} };
	UF_CURVE_arc_t arc; //структура данных дуги
	tag_t objarray[4], //теги для трех линий и дуги
		wcs_tag; //тег для системы координат
		//предельные углы вращения сечения
	char *limit[2] = { a, b };
	//точка начала вектора вращения
	double point[3] = { 0.0, 0.0, 0.0 };
	//орты вектора вращения
	double direction[3] = { 0.0, 1.0, 0.0 };
	//признак создания самостоятешьного тела
	UF_FEATURE_SIGN sign = UF_NULLSIGN;
	//заготовки указателей на перечии объектов
	uf_list_p_t loop_list, features;
	int i;
	if (!UF_initialize()) {
		// задание пераметров дуги в сечении
```

```
            arc.start_angle = -90.0 * DEGRA;
            arc.end_angle = 90.0 * DEGRA;
            arc.arc_center[0] = 40.0;
            arc.arc_center[1] = 20.0;
            arc.arc_center[2] = 0.0;
            arc.radius = 20.0;
            for (i = 0; i < 3; i++) //построение трех линий сечения
                  UF_CURVE_create_line(&line[i], &objarray[i]);
            //получение матрицы абсолютных координат
            UF_CSYS_ask_wcs(&wcs_tag);
            //связывание матрицы поворота дуги с абсолютными координатами
            UF_CSYS_ask_matrix_of_object(wcs_tag, &arc.matrix_tag);
            //построение дуги сечения
            UF_CURVE_create_arc(&arc, &objarray[i]);
            //создание пустого перечия объектов
            UF_MODL_create_list(&loop_list);
            //заполнение перечия тремя линиями и одной дугой
            for (i = 0; i < 4; i++)
                  UF_MODL_put_list_item(loop_list, objarray[i]);
            //создание тела вращения
            UF_MODL_create_revolved(loop_list, limit, point, direction, sign,
&features);
            UF_terminate();
      }
}
int ufusr_ask_unload(void) {//типовая функция завершения приложения
      return (UF_UNLOAD_IMMEDIATELY);
}
```

В результате выполнения программного кода, приведенного в Листинге 3.2, будет построено тело вращения, как показано на рисунке 3.2.

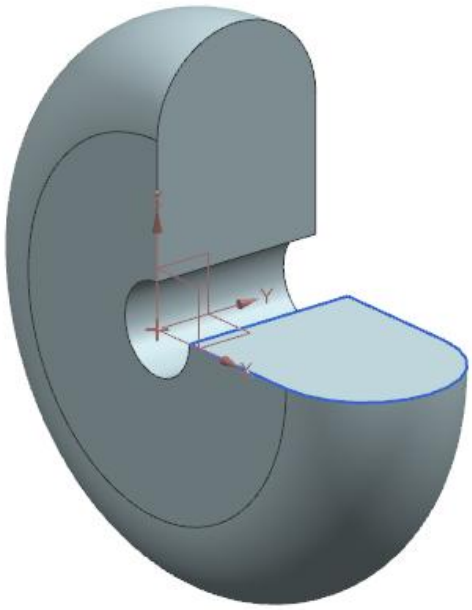


Рисунок 3.3 – Тело, построенное с применением операции вращения

3.3 Операция выдавливания

Одной из часто используемых операций в процессе 3D-моделирования является операция выдавливания. Для программирования операции выдавливания используем функцию:

```
int UF_MODL_create_extruded(
uf_list_p_t objects ,/* <I> Список объектов, подлежащих выдавливанию (элементы сечения) */
char * taper_angle ,/* <I> Угол конусности в градусах. */
char * limit[2] ,/* <I> Предел выдавливания. Объявляется как:
char *limit[2]. Первое значение - это начальное значение выдавливания, а второе – конечное значение расстояния выдавливания
*/
double point[3] ,/* <I> Не используется */
double direction[3] ,/* <I> Орты вектора направления выдавливания. */
```

```
    UF_FEATURE_SIGN sign ,/* <I> Логические операции при создании
тела вращения */
    uf_list_p_t * features  /* Перечень созданных объектов */
    ).
```

Данная функция по большинству параметров идентична функции UF_MODL_create_revolved(…), другие параметры описаны в комментариях. Рассмотрим применение операции выдавливания для эскиза, приведенного на рисунке 3.2, начало выдавливания зададим на расстоянии 30 мм от эскиза, общее расстояние выдавливания – 80 мм. Исходной код, в котором реализуется данная задача приведен в Листинге 3.3.

Листинг 3.3.

```
//Программа построения тела
//при помощи операции выдавливания
#include <uf.h>
#include <uf_curve.h>
#include <uf_csys.h>
#include <uf_modl.h>
char a[] = "30.0";
char b[] = "80.0";
char t_a[] = "0.0";
void ufusr(char *param, int *retcode, int param_len) {
    /* Определяется массив линий с координатами их
    концов, используемых в эскизе выдавливаемого сечения */
    UF_CURVE_line_t line[3] = {
        {{10.0,0.0,0.0}, {10.0,40.0,0.0}},
        {{10.0,0.0,0.0}, {40.0,0.0,0.0}},
        {{10.0,40.0,0.0}, {40.0,40.0,0.0}} };
    UF_CURVE_arc_t arc; //структура данных дуги
    tag_t objarray[4], //теги для трех линий и дуги
        wcs_tag; //тег для системы координат
    int i;
     //параметры операции выдавливания
    //расстояние начала и расстояние окончания выдавливания
```

```
    char *limit[2] = { a, b };
    //орты вектора направления выдавливания
    double direction[3] = { 0.0, 0.0, 1.0 };
    //признак создания самостоятельного тела
    UF_FEATURE_SIGN create = UF_NULLSIGN;
    //заготовки указателей на перечии объектов
    uf_list_p_t loop_list, //Список объектов, подлежащих выдавливанию (эле-
менты сечения)
    features; //Список созданных идентификаторов объектов
    //угол конусности в градусах
    char *taper_angle = t_a;
    //параметр не используется
    double ref_pt[3];
    if (!UF_initialize()) {
        // задание параметров дуги в сечении
        arc.start_angle = -90.0 * DEGRA;
        arc.end_angle = 90.0 * DEGRA;
        arc.arc_center[0] = 40.0;
        arc.arc_center[1] = 20.0;
        arc.arc_center[2] = 0.0;
        arc.radius = 20.0;
        for (i = 0; i < 3; i++) //построение трех линий сечения
            UF_CURVE_create_line(&line[i], &objarray[i]);
        //получение матрицы абсолютных координат
        UF_CSYS_ask_wcs(&wcs_tag);
        //связывание матрицы поворота дуги с абсолютными координатами
        UF_CSYS_ask_matrix_of_object(wcs_tag, &arc.matrix_tag);
        //построение дуги сечения
        UF_CURVE_create_arc(&arc, &objarray[i]);
        //создание пустого перечия объектов
        UF_MODL_create_list(&loop_list);
        //заполнение перечия тремя линиями и одной дугой
        for (i = 0; i < 4; i++)
            UF_MODL_put_list_item(loop_list, objarray[i]);
```

```
            //создание операции выдавливания
            UF_MODL_create_extruded(loop_list, taper_angle, limit,
                  ref_pt, direction, create, &features);
            UF_terminate();
      }
}
int ufusr_ask_unload(void) {//типовая функция завершения приложения
      return (UF_UNLOAD_IMMEDIATELY);
}
```

В результате выполнения приведенного кода будет построено тело в соответствии с рисунком 3.4.

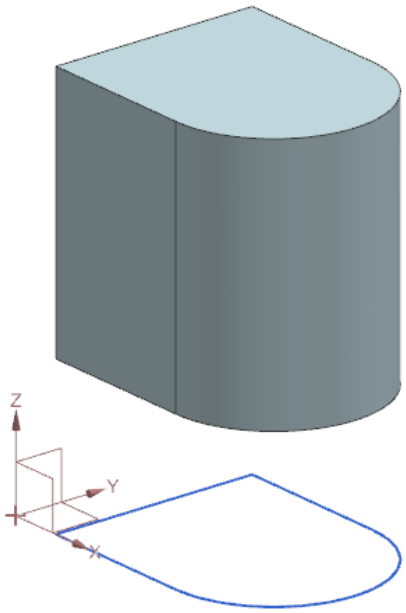


Рисунок 3.4 – Тело, построенное при помощи операции выдавливания

3.4 Удаление объектов

Если требуется удалить какой либо из объектов сцены, можно использовать функцию `UF_OBJ_delete_object ( ... )`, где в качестве единственного входного параметра используется тег тела, подлежащего удалению. Однако, следует помнить, что не допускается использование этой функции при переборе объектов с помощью `UF_OBJ_cycle ...` и им подобных. Удаление объекта в момент перебора с

помощью указанных функций, вызовет нарушение работы программы. Рекомендуется сначала собрать информацию об удаляемых тегах в массив, с помощью тех же функций перебора, а уже затем удалить эти объекты.

Сценарий работы примера будет следующий: перед запуском модуля пользователь должен создать на сцене несколько одиночных тел. При запуске модуля в системе должно появиться диалоговое окно, позволяющее пользователю указать тело, подлежащее удалению. Это действие выполняется мышью, указанное тело должно подсветиться, после чего пользователь подтверждает удаление, нажатием кнопки на диалоге и тело исчезает со сцены. Чтобы реализовать такой сценарий нам потребуется функция для создания стандартного системного диалога выбора тела в NX. Здесь мы используем функцию `UF_UI_select_single ( ... )`.

На примере этой функции рассмотрим возможные варианты поиска информации о функциях Open API в установленной системе NX. Чтобы найти интересующую функцию необходимо задать поиск среди *.h файлов в каталоге UGOPEN системы NX с опцией, реализующей поиск текста внутри файлов. В качестве искомого текста задать имя функции. Искомая функция находится в файле `uf_ui.h`.

Пример стандартного описания функции в *.h файлах:

```
    extern UGOPENINTEXPORT int UF_UI_select_single(
    char * message,/* <I> Cue line message to display. */
    UF_UI_selection_options_p_t opts,/* <I>   Selection options.
See the Data structures section of this chapter.   */
    int * response,/* <O> response:
    1 = Back
    2 = Cancel
    4 = Object selected by name
    5 = Object selected  */
    tag_p_t object,/* <O> Object identifier of selected object */
    double cursor[3],/* <O>   Cursor position. This is undefined
if response is 4  (object selected by name).   */
```

```
    tag_p_t view  /* <O>   View of selection. This is NULL_TAG if
response is   4 (object selected by name).   */
    ).
```

Из описания этой функции становится понятным, что в ней шесть параметров и предназначена она для выбора на сцене NX только одного объекта. Те параметры функции, которые являются входными обозначены символом <*I*> , а те, которые являются выходными - символом <*O*>. Таким образом, у рассматриваемой функции два входных и четыре выходных параметра.

Первый входной параметр - указатель на текстовую строку с сообщением, которое появляется в строке статуса системы при активизации диалога. Здесь обычно пишут фразу - указание пользователю, что он должен сделать при появлении диалогового окна.

Второй входной параметр указатель на структуру **UF_UI_selection_options_t**, определяющую шаблон выбора объекта на сцене системы. То, что это указатель, а не сама структура, задается префиксом **_p_** в имени переменной (это общее соглашение при обозначении переменных в NX). Чтобы ознакомиться с этой структурой необходимо снова выполнить поиск по файлам *.h, в каталоге UGOPEN. В качестве искомого текста задать имя структуры (рекомендуется поиск структур проводить по имени без префиксов, то есть искать **UF_UI_selection_option**). Искомая структура находится в файле uf_ui_types.h:

```
    struct UF_UI_selection_option_s
    {
    int num_mask_triples;
    UF_UI_mask_p_t mask_triples; /* <len:num_mask_triples> */
    int scope; /* scopes are listed in uf_ui.h */
    int other_options; /* initially ignored (set to 0) */
    void *reserved;    /* initially ignore (set to NULL) */
    }.
```

Из описания видно, что в структуре пять полей. Первое поле не описано, но по комментариям ко второму полю становится понятным, что первое поле содержит количество структур масок, задаваемых при выборе объекта на сцене. Отсюда ясно, что хотя функция выбирает всего один объект, но типов объектов при выборе может быть задано несколько.

Второе поле в структуре, следовательно, указатель на массив масок, каждая из которых сама представляет структуру типа `UF_UI_mask_t`. Далее указывается, что описание третьего поля находится в файле uf_ui.h, в котором говорится, что `scope` это константа, определяющая область выбора объекта:

```
Selection scope
UF_UI_SEL_SCOPE_NO_CHANGE
UF_UI_SEL_SCOPE_ANY_IN_ASSEMBLY
UF_UI_SEL_SCOPE_WORK_PART
UF_UI_SEL_SCOPE_WORK_PART_AND_OCC
```

Из названий констант видно, что:

- первая определяет выбор объекта без его последующего изменения,
- вторая определяет выбор объекта на сборочной модели,
- третья на рабочей сцене,
- четвертая на рабочей сцене или подсборке.

Четвертое и пятое поле структуры `UF_UI_selection_option` не задаются.

Чтобы узнать, что из себя представляет структура маски `UF_UI_mask`, необходимо снова выполнить поиск с текстовым параметром, соответствующим имени маски. Искомая структура также описана в файле uf_ui_types.h:

```
struct UF_UI_mask_s
{
int object_type; /* This can be one of the object types that
are listed in
uf_object_types.h or UF_pseudo_object_type */
```

```
    int object_subtype; /* This can either be UF_all_subtype (not
with UF_pseudo_object_type), or one of the corresponding supported
subtypes of the object type specified.
    This is ignored for certain types like UF_solid_type */
    int solid_type;  /* This should be named detail_type and is
only meaningful for certain object_types like
    UF_solid_type, UF_feature_type or UF_pseudo_type.
    This should be set to one of the corresponding detail types
    of the object type and object_subtype specified.
    This is ignored for UF_all_subtype. */
    }.
```

Из описания следует, что первое поле структуры должно содержать код типа выбираемого объекта, а коды эти описаны в файле `uf_object_types`. Второе поле структуры - это код подтипа объекта (из того же описательного файла), а третье поле применяется только для твёрдого тела или его элементов и игнорируется для всех подтипов объектов.

У рассматриваемой функции `UF_UI_select_single` остались не изучены выходные параметры.

Третий параметр (`int * response`) - при закрытии диалога получает код нажатой на нем кнопки.

Четвертый параметр (`tag_p _ t object`) - получает тег выбранного объекта.

Пятый параметр (`double cursor[3]`) - получает координаты курсора в момент закрытия диалога.

Шестой (`tag_p_t view`) - тег области просмотра, в которой отображен выбранный объект (этот тег используется в ряде функций из библиотеки работы с дисплеем NX, описание располагается в файле uf_ view.h).

В завершение разбора функции одиночного выбора следует отметить, что и маску и сценарий выбора объектов в ней можно динамически менять, в ходе выполнения программы, с помощью функций `UF_UI_set_sel_mask ( ... )` и `UF_UI_select_with_single_dialog ( ... )`.

На основе полученной информации разработаем программу удаления объектов в соответствии с Листингом 3.4.

Листинг 3.4.

```
//Программа удаления выбранного тела со сцены
#include <uf.h>
#include <uf_ui.h>
#include <uf_obj.h>
char mes[128] = "Выделите твердое тело";
void ufusr(char *param, int *retcode, int param_len) {
	// определяем две переменвые тегов первую - для удаляемого тела
	//вторую - для области просмотра объекта
	tag_t body, view;
	// определяем структуру для настройки диалога
	UF_UI_selection_options_t opts;
	// определяем структуру для маски выбора и сразу заносим в ее поля, что
// объектом выбора является твердое тело и элементы (FEATURE) твердого тела
	UF_UI_mask_t mask = { UF_solid_type,0, UF_UI_SEL_FEATURE_BODY };
	///определяем переменную для приема кода нажатой кнопки на диалоге
	//и переменную для приема кода ошибки выполнения функции
	int response = 2, error = 0;
	//и массив для получения текущих координат мыши
	double cursor[3];
	//далее заполняем поля управляющей структуры диалога
		//сначала-определим, что у нас будет только одна маска
	opts.num_mask_triples = 1;
	//адрес этой маски занесем в требуемое поле структуры
	opts.mask_triples = &mask;
	// определим, что выбор будет происходить на текущей рабочей сцене
	opts.scope = UF_UI_SEL_SCOPE_WORK_PART;
	if (!UF_initialize()) {// откроем диалог выбора объекта
		error = UF_UI_select_single(mes, &opts, &response, &body, cursor,
&view);
		// если не было ошибки при работе функции выбора
		// и пользователь не нажимал ни кнопку НАЗАД ни ОТМЕНА
```

```
            if (!error && response != 1 && response != 2)
                  //то производим удаление выбранного объекта
                  UF_OBJ_delete_object(body);
            //в любом случае завершаем работу библиотеки Open API
            UF_terminate();
      }
}
//типовая функция завершения приложения
int ufusr_ask_unload(void) {
      return (UF_UNLOAD_IMMEDIATELY);
}
```

Для проверки работы приложения необходимо предварительно построить на сцене несколько твердых тел. Например, три куба и три конуса. При запуске приложения появляется диалог выбора объектов. Указателем мыши нужно выбрать объект (на рисунке 3.5, а – конус) и он удаляется со сцены, программа завершает свою работу. Результат выполнения листинга 3.4 приведен на рисунке 3.5.

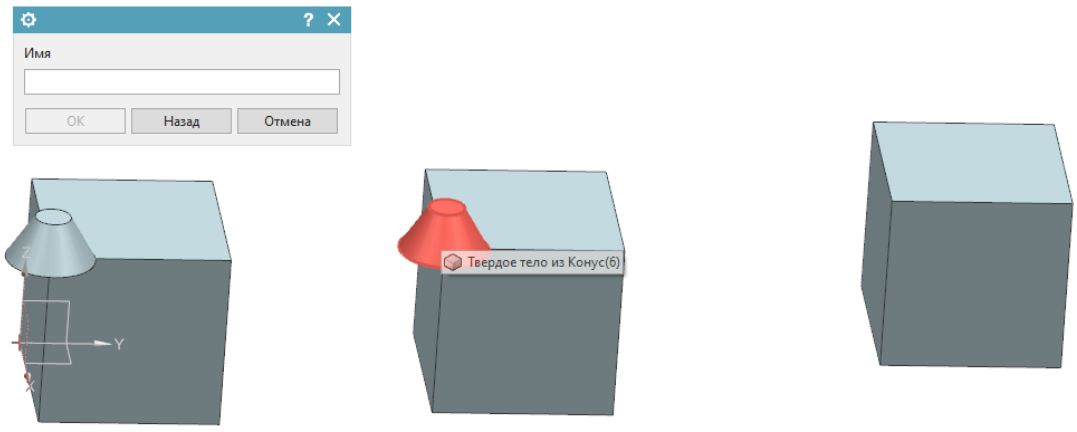
3.5 Копирование объектов

Рассмотрим процесс программного копирования объектов в NX на примере простого объекта из раздела «Конструктивный элемент», например-шара. Исходный элемент может находиться в произвольной точке системы координат (в данном примере расположим объект в ее начале) и, предположим, что необходимо получить 10 копий этого элемента на равных расстояниях друг от друга вдоль оси *X*.

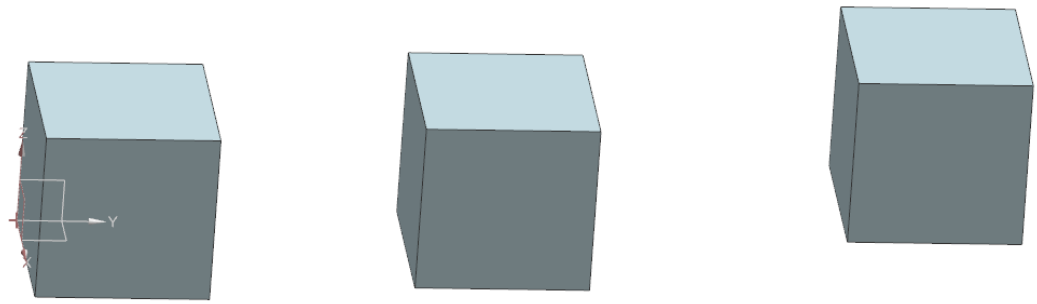
Для решения задачи воспользуемся двумя функциями:

- UF_MODL_copy_paste_features (...) - для получения копии заданного элемента;

- UF_MODL_move_feature (...) - для перемещения копии в заданную точку сцены.



а)



б)

а – сцена NX в процессе удаления объектов;

б – сцена NX после удаления объектов;

Рисунок 3.5 – Удаление объектов со сцены модели NX

У первой функции восемь параметров, из них семь первых - входные и один, последний, выходной.

```
int UF_MODL_copy_paste_features
(
tag_t    *feature_array,/* <I,len:num_features> Array of fea-
tures identifier*/
int      num_features, /* <I> Number of features */
tag_t    *old_parents, /* <I,len:num_parents> Array of old
parents */
```

```
    tag_t   *new_parents, /* <I,len:num_parents> Array of new
parent */
    int     num_parents, /* <I> Number of refrences */
    int     expression_transfer_mode, /* <I> Type of expression
copy - 0 = New, 1 = link, 2 = instance */
    int     parent_transfer_mode, /* <I> Type of reference copy -
0 = New, 1 = link, 2 = instance */
    tag_p_t * new_feature_array /* <OF,len:num_features> Array of
new features */
    ).
```

Первый параметр - указатель на массив тегов объектов, подлежащих копированию. Это удобно, когда копируется объект, состоящий из нескольких типовых элементов. В рассматриваемом примере копируется один элемент и указатель будет адресовать только его.

Второй параметр - количество элементов в предыдущем массиве (у нас - один).

Третий параметр - указатель на массив тегов «родителей» копируемых элементов. В рассматриваемом примере у шара их нет, поэтому этот параметр будет равен нулю.

Более правильно было бы программным путем убедиться, что у копируемого элемента нет родителей, а если они есть, то предварительно провести копирование их. Для получения тегов родителей элемента можно воспользоваться функцией:

```
    int UF_MODL_ask_references_of_features
    (
    tag_t   *feature_array, /* <I> Указатель на массив элементов,
у которых проверяются родители */
    int     num_features, /* <I> Количество этих элементов  */
    tag_t   **parents, /* <OF> Указатель на массив указателей те-
гов родителей. Этот указатель впоследствии должен быть освобожден
функцией UF_free.   */
```

```
    char    ***parent_names, /* <OF> Указатель на массив указате-
лей строк с именами каждого родителя. Этот указатель впоследствии
должен быть освобожден функцией   UF_free_string_array.  */
    int     *num_parents /* <O>  Количество найденных родителей
*/
    ).
```

Четвертый параметр - указатель на массив «новых родителей» для скопированного элемента. Очевидно, что если бы у шара были «родители», от которых он бы зависел, то для его копирования сначала надо было бы скопировать этих «родителей», получить теги этих копий и вставить их в указатель «новых родителей» при копировании самого шара. Но в примере шар будет абсолютно самостоятельным телом, поэтому в этом параметре тоже устанавливается ноль.

Пятый параметр должен хранить число «родителей» копируемого элемента. Их в примере - ноль.

Шестой параметр - тип преобразования объекта при копировании. Допустимые коды типа:

- 0 - самостоятельный объект;

- 1 - объект, связанный с источником;

- 2 - объект со случайной привязкой.

Седьмой параметр - определяет аналогичный тип преобразования, но уже по отношению к родителям копируемого объекта.

Восьмой параметр (выходной) - получает указатель на массив тегов скопированных объектов.

Скопированный объект появляется на сцене в том же самом месте, что и оригинал (совмещен с ним). Чтобы увидеть новый объект его нужно отодвинуть в сторону от оригинала. Это действие выполняет функция **UF_MODL_move_feature (...).** У этой функции всего три параметра.

```
    int UF_MODL_move_feature(
    uf_list_p_t cmtag,/* <I> Linked list of tags for features to
be moved. */
```

```
    int mode,/* <I> Option specifying how move is to be done.
                    POINT_TO_POINT  ( 0 )
                    AXIS_TO_AXIS    ( 1 )
                    CSYS_TO_CSYS    ( 2 ) */
    double  real_data[2][12]  /* <I> CSYS describing the move.
real_data[0] holds 12 element reference coordinate system matrix
(first three elements translation, remainder is rotation).
                    real_data[1] holds 12 element destination co-
ordinate system matrix. */
    ).
```

Первый - представляет собой стандартный список с тегами перемещаемых элементов.

Второй - константа, задающая порядок перемещения объекта:

- 0 - перемещение типа ТОЧКА –ТОЧКА;

- 1 - перемещение типа ОСЬ – ОСЬ;

- 2 - перемещение в текущей системе координат.

Третий – определяет матрицу перемещения объекта. Формат матрицы следующий:

p			X			Y			Z		
X_s	Y_s	Z_s	1	0	0	0	1	0	0	0	1
X_d	Y_d	Z_d	1	0	0	0	1	0	0	0	1

Матрица имеет две строки и двенадцать столбцов. Первые три элемента в нулевой строке - координаты некоторой исходной точки копирования p (индекс s-источник, d-приемник). Далее в строке по три элемента - орты, определяющие направления осей соответственно X, Y, Z исходной системы координат. Во второй строке матрицы те же величины, но для целевой точки копирования (координаты точки и орты поворота осей координат при копировании).

Например, требуется переместить цилиндр из начала системы координат вдоль оси *Y* на 100 мм и повернуть его при этом на 90° вокруг оси *X*. Матрица перемещения для этого случая должна иметь вид:

	p		X			Y			Z			
0	0	0	1	0	0	0	1	0	0	0	0	– источник
0	100	0	1	0	0	0	0	−1	0	1	1	– целевой объект

где во второй строке заданы новые координаты опорной точки (0, 100, 0), затем орт оси *X* - не изменился (по сравнению с первой строкой), орт оси *Y* стал направлен в противоположную сторону исходной оси *Z*, а орт оси *Z* совпадает с исходным состоянием оси *Y* (рисунок 3.6).

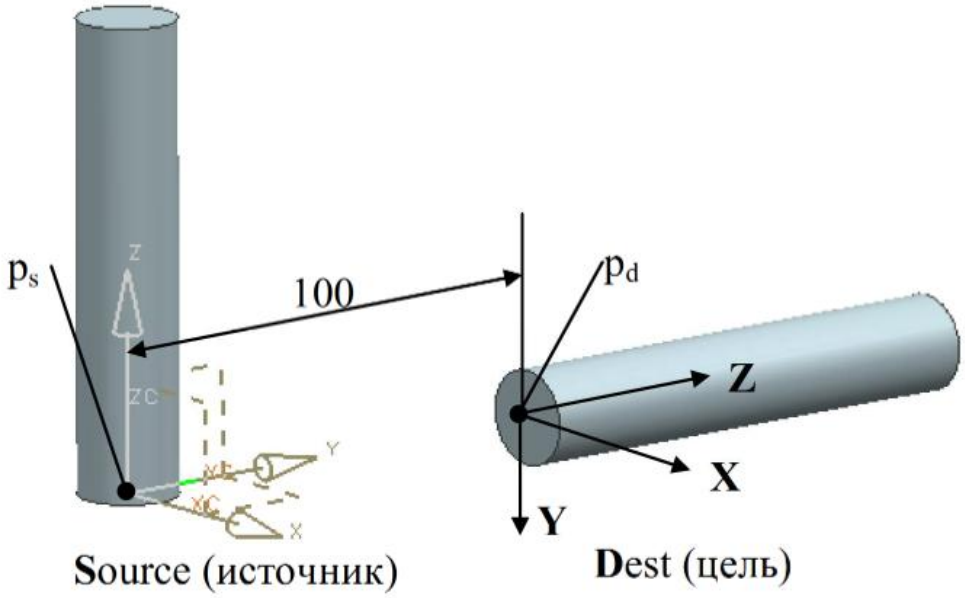


Рисунок 3.6 – Перемещение объекта, совещенное с поворотом

На основе полученной информации разработаем программу копирования простых объектов по осям *X* и *Y* в соответствии с Листингом 3.5.

Листинг 3.5.

```
//Модуль копирования простого тела на сцене
#include <uf.h>
#include <uf_modl.h>
```

```
void ufusr(char *param, int *retcode, int paramLen) {
	//определим точку Ps
	double cyl_orig[3] = { 0.0,0.0,0.0 };
	//определим переменную и зададим диаметр шара в 100 мм
	char diam[] = "100";
	//определим переменную для тега шара (оригинала)
	tag_t sph_obj;
	//определим указатель на стандартный список системы NX
	uf_list_p_t list;
	//зададим матрицу переноса шара на 100 мм вдоль оси Y без поворота
	double real_data[2][12] = {
	{0.0, 0.0, 0.0,	1.0, 0.0, 0.0,  0.0, 1.0, 0.0,  0.0, 0.0, 1.0},
	{0.0,100.0,0.0,	1.0, 0.0, 0.0,  0.0, 1.0, 0.0,  0.0, 0.0, 1.0 } };
	// для создания первого шара требуется константа булевой операции
	UF_FEATURE_SIGN sign = UF_NULLSIGN;
	int i;
	//определим указатель на массив тегов копий шара
	tag_p_t new_feature_array[10];
	//проводим инициализацию Open API
	if (!UF_initialize()) {
		//создаем на сцене исходный шар
		UF_MODL_create_sphere1(sign, cyl_orig, diam, &sph_obj);
		//в цикле создаем копии исходного шара и сразу перемещаем их
		for (i = 0; i < 10; i++) {
			UF_MODL_copy_paste_features(&sph_obj, 1, 0, 0, 0, 0, 0,
&new_feature_array[i]);
			UF_MODL_create_list(&list); //создаем чистый список
			//заносим в него тег текущей копии шара
			UF_MODL_put_list_item(list, new_feature_array[i][0]);
			//выполняем перемещение шара на новое место
			UF_MODL_move_feature(list, CSYS_TO_CSYS, real_data);
			/* обновляем базу данных объектов системы, чтобы изменения
отобразились на экране */
			UF_MODL_update();
```

```
                UF_MODL_delete_list(&list); //уничтожаем список
                //увеличиваем координату Y в матрице перемещения на 100 мм
                real_data[1][1] += 100.0;
            }
            //зададим матрицу переноса шара на 100 мм вдоль оси X без поворота
            double real_data[2][12] = {
{0.0, 0.0, 0.0,       1.0, 0.0, 0.0,  0.0, 1.0, 0.0,  0.0, 0.0, 1.0},
{100.0, 0.0,0.0,      1.0, 0.0, 0.0,  0.0, 1.0, 0.0,  0.0, 0.0, 1.0 } };
            //Остальные действия выполняются по примеру с осью Y
            for (i = 0; i < 10; i++) {
                UF_MODL_copy_paste_features(&sph_obj, 1, 0, 0, 0, 0, 0,
&new_feature_array[i]);
                UF_MODL_create_list(&list); //создаем чистый список
                //заносим в него тег текущей копии шара
                UF_MODL_put_list_item(list, new_feature_array[i][0]);
                //выполняем перемещение шара на новое место
                UF_MODL_move_feature(list, CSYS_TO_CSYS, real_data);
                /* обновляем базу данных объектов системы, чтобы изменения
отобразились на экране */
                UF_MODL_update();
                UF_MODL_delete_list(&list); //уничтожаем список
                //увеличиваем координату X в матрице перемещения на 100 мм
                real_data[1][0] += 100.0;
            }
            UF_terminate();
      }
}
//типовая функция завершения приложения
int ufusr_ask_unload(void) {
      return (UF_UNLOAD_IMMEDIATELY);
      }
```

В результате работы программы, приведенной в Листинге 3.5 будет скопировано 20 шаров, как показано на рисунке 3.7.

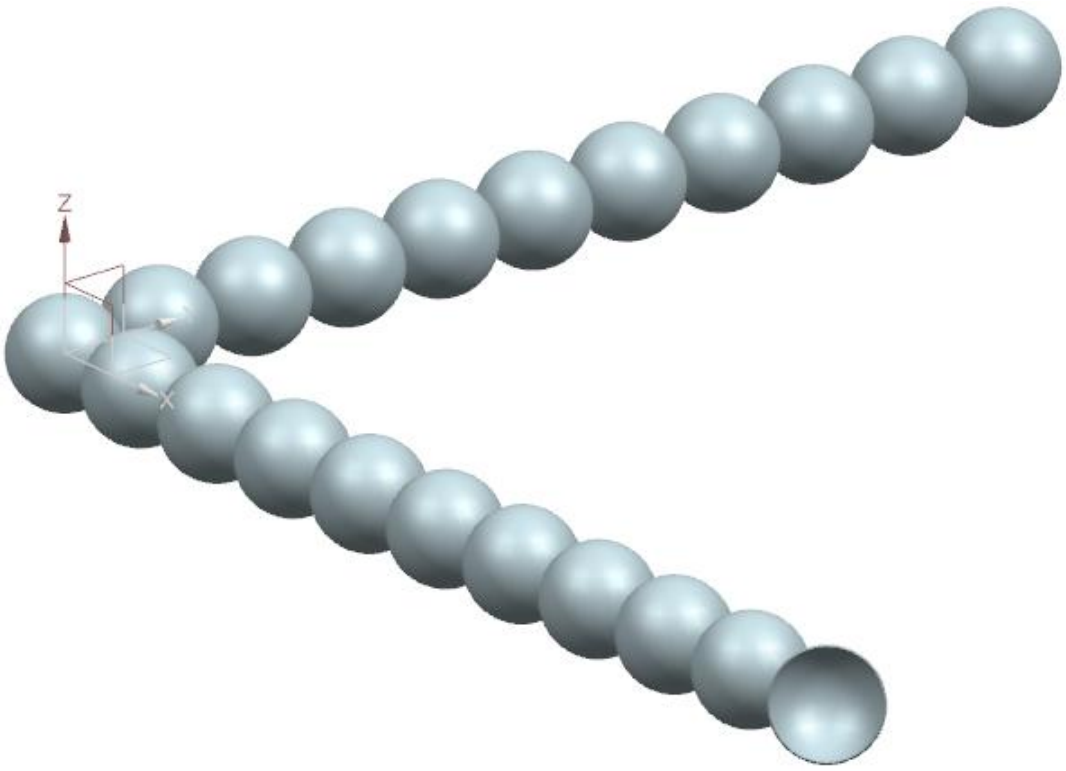


Рисунок 3.7 – Результат программного копирования простых объектов

3.6 Анализ 3D тела в NX

Часто, при программной обработке 3D-модели, требуется знание структуры этой модели для принятия решения о каких-то дальнейших действиях над ней. Если схематично изобразить общую концепцию формирования твердого тела в NX, можно составить схему, представленную на рисунке 3.8.

Тело (body), с одной стороны, определяется геометрией, состоящей из граней (face), рёбер (ege), точек (point), с другой – порядком построения тела операциями (feature): построение линии, вытягивание, сглаживание, фаска, булева операция и другие. Следует понимать, что геометрия и действия-построения – это разные объекты, пронизывающие друг друга, хотя могут существовать и независимо. Так, например, если полностью разрушить параметризацию, то останется одна геометрия. Она самодостаточна. И, наоборот, если в параметрической модели подавить все операции то исчезнет вся геометрия, но дерево построения останется (пунктирный

каркас на рисунке 3.8). Для существования дерева построения геометрия тоже не нужна.

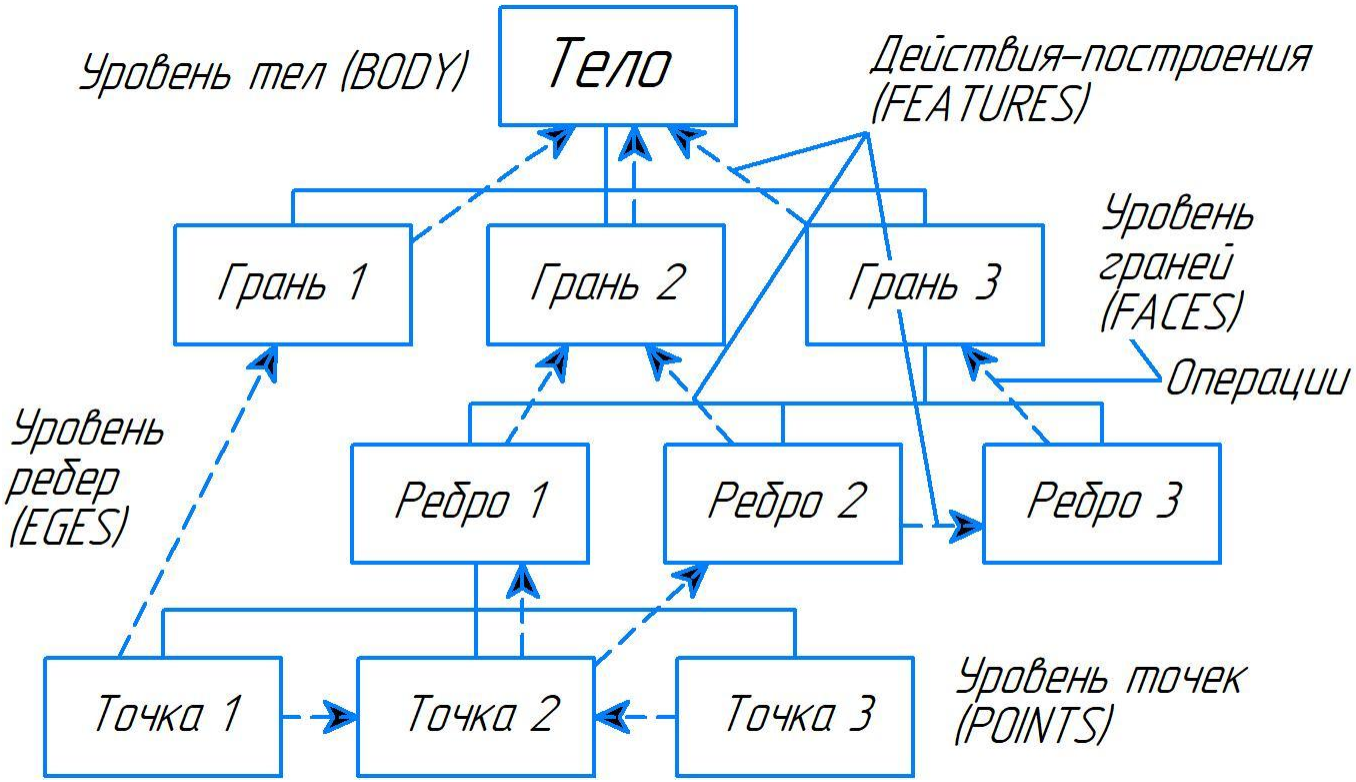


Рисунок 3.8 – Обобщенное дерево структуры твердого тела

Нижние уровни схемы называются «родительскими» элементами («родителями»), верхние – «детьми».

При анализе структуры тела следует вскрыть всю цепочку построения от самых нижних уровней до самого верхнего и извлечь из каждого уровня как можно больше информации о геометрических параметрах элементов и условиях выполнения операций.

Поиск нужных функций для анализа в системе NX обычно не представляет труда. Все файлы описаний функций (*.h), расположенные в каталоге UGOPEN, установленного пакета NX, довольно подробно документированы. В состав имен функций входят все признаки, необходимые для поиска:

- приставка - означающая имя модуля, в который входит функция;

- префикс - определяющий действие функции (для рассматриваемой темы это префикс _ask_ - спросить (запросить));

- окончание - имя (или аббревиатура) элемента, с которым производится действие.

Для ознакомления с функциями Open API, позволяющими выполнить анализ структуры объекта построим тестовое тело (рисунок 3.9), состоящее из нескольких, часто встречающихся, элементов: сферы, цилиндра, скругления.

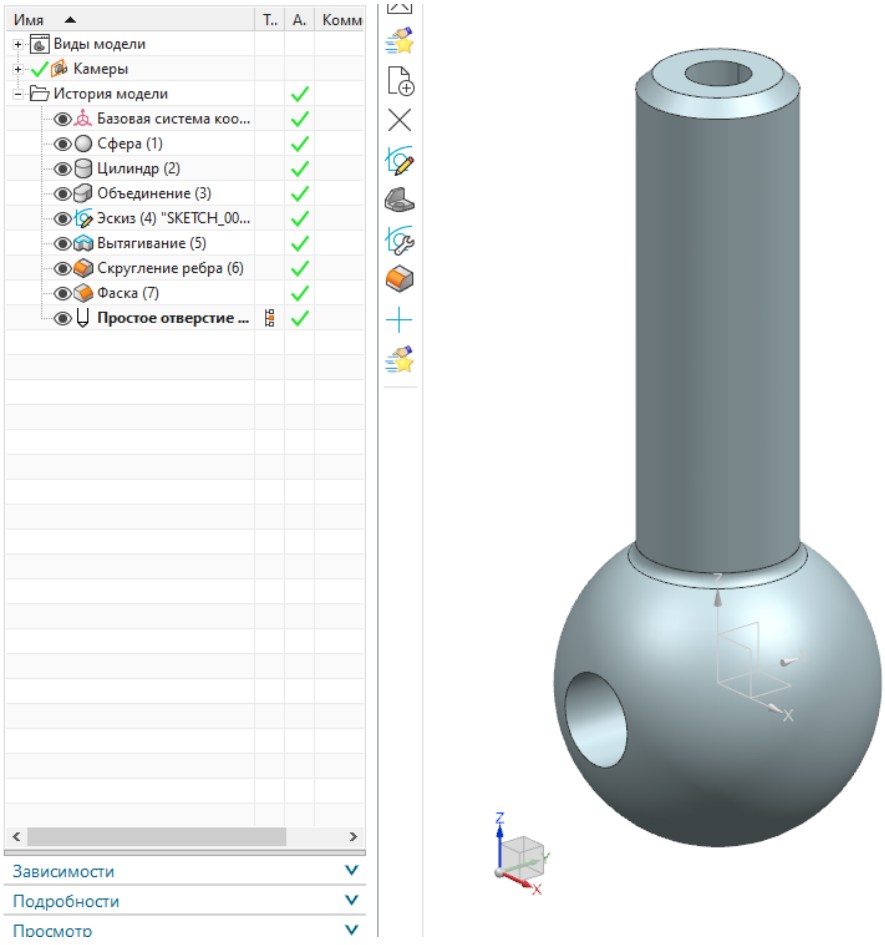


Рисунок 3.9 – Тестовое тело для изучения функций анализа 3D-модели

Разрабатываемый модуль должен реализовывать следующий сценарий работы:

- запрашивать выбор исследуемого тела на рабочей сцене;
- определять набор элементов, из которых состоит тело;
- для каждого элемента:
- определять его геометрические характеристики;

- определять «родителей» элемент;

- определять «детей» элемента.

Набор процедур, которые использованы в модуле, может быть проиллюстрирован укороченной схемой алгоритма его работы, представленной на рисунке 3.10.

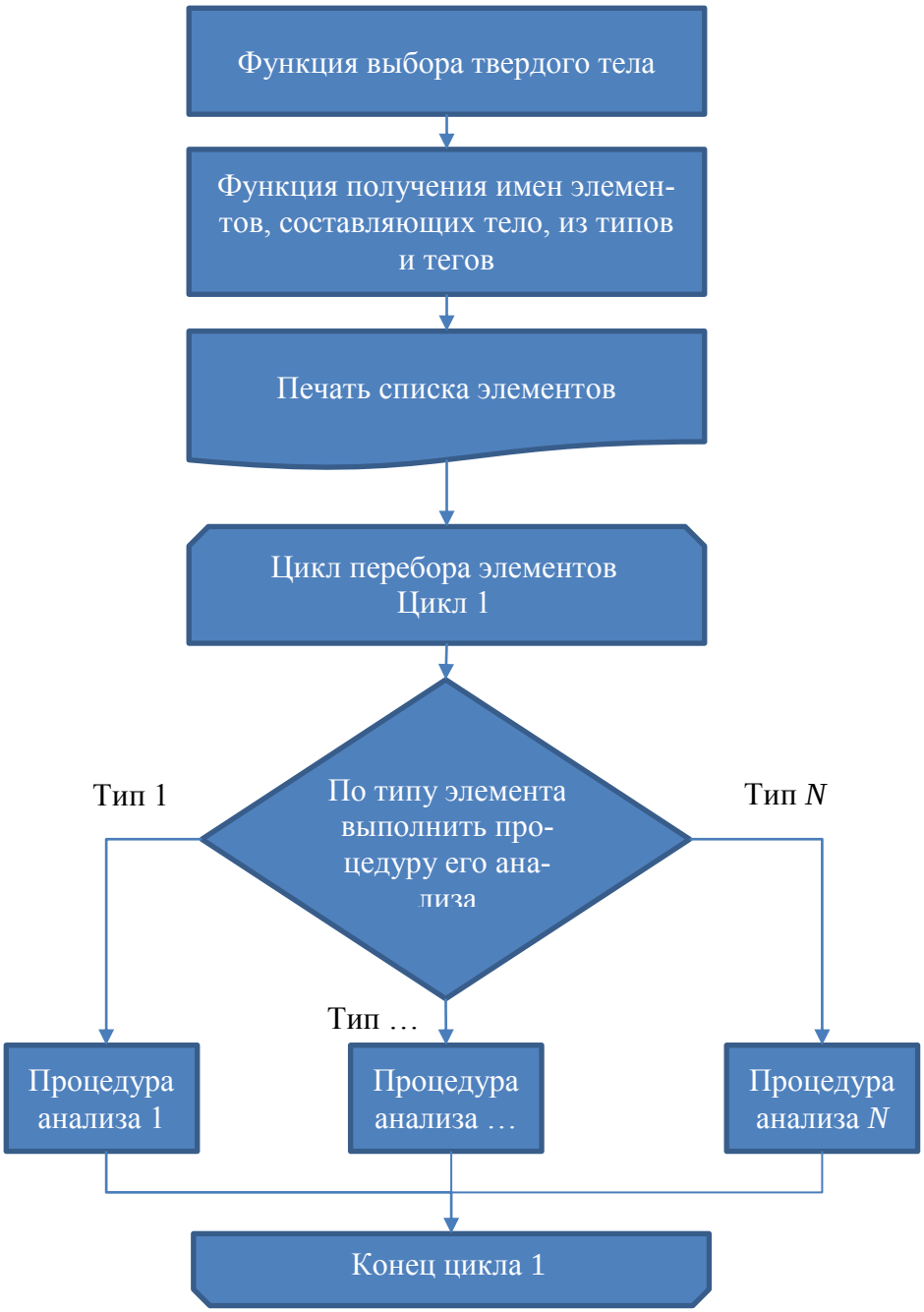


Рисунок 3.10 – Схема взаимодействия функций и процедур при анализе тела

Как видно из схемы, работа процедуры начинается с функции выбора тела. Ранее были использованы некоторые функции для выбора объектов на сцене, здесь будет использоваться одна из них - `UF_UI_select_single (...)`, но с настройкам для выбора только твердых тел. Для этого структуру маски функции необходимо заполнить типом тела `UF_solid_type` и видом твердого тела – `UF_UI_SEL_FEATURE_BODY`.

Для следующего шага - получения элементов, составляющих тела, можно использовать функцию: `UF_MODL_ask_body_features (...)`, в которой имеется три параметра: один входной - тег разбираемого на части тела, и два выходных:

- количество найденных составляющих элементов;

- указатель на массив структур с тегами, типами и именами элементов.

Каждый тип элемента анализируется по-особому, поэтому в программе дальше начинается цикл (по количеству элементов), в котором в зависимости от типа элемента запускается соответствующая процедура его обработки.

Разработаем исходный код главного модуля, реализующий описанный алгоритм и приведенный в Листинге 3.6.

Листинг 3.6.

```
//Программа, демонстрирующая набор функций UG / Open API,
//предназначенных для анализа структуры детали в виде твердого тела
#include<stdlib.h>
#include<stdio.h>
#include<uf.h>
#include<uf_disp.h>
#include<uf_modl.h>
#include<uf_ui.h>
#include<uf_object_types.h>
#include<uf_modl_types.h>
#include<uf_sket_types.h>
#include    <uf_sket.h>
//процедуры анализа конкретных элементов
void SphereSub(tag_p_t ftags, int nt);//анализ сферы
void BlendSub(tag_p_t ftags, int nt);//анализ скругления
```

```
void CylinderSub(tag_p_t ftags, int nt);//анализ цилиндра
void PatternAndChildrenSub(tag_t ftag); //определение родителей и детей
char mes[131];//переменная для текстовых сообщений
char msg[131] = "Выберите твердое тело";
int cnt, i; char* tip;
int feat_count;
UF_MODL_features_p_t *features_node;
//переменные для функции выбора тела
double cursor[3];
int err = 0, response = 4;
UF_UI_selection_options_t opts;
UF_UI_mask_t mask = { UF_solid_type,0, UF_UI_SEL_FEATURE_BODY };
int body_type;
tag_t body, view;
/*Главная точка входа*/
void ufusr(char *param, int *retcode, int param_len)
{
      if (!UF_initialize()) {//заполнение структуры для функции выбора тела
            opts.num_mask_triples = 1;
            opts.mask_triples = &mask;
            opts.scope = UF_UI_SEL_SCOPE_WORK_PART;
            //активизация информационного окна системы для вывода текста
            UF_UI_open_listing_window();
            //начать цикл
            do {//открыть диалог выбора тела
                  err = UF_UI_select_single(msg, &opts, &response, &body, cur-
sor, &view);
                  //если на диалоге нажата кнопка ОК или ПРИНЯТЬ,
                  if ((response == 4) || (response == 5))
                  {//то следует определить тип выбранного тела
                        UF_MODL_ask_body_type(body, &body_type);
                        //если выбрано твердое тело,
                        if (body_type == UF_MODL_SOLID_BODY)
            {//то следует получить массив тегов элементов, составляющих тело
```

```
                UF_MODL_ask_body_features(body, &feat_count, &features_node);
                        //цикл по количеству найденных элементов
                        for (i = 0; i < feat_count; i++)
                        {//запомнить текущее количество и тип элемента
                              cnt = features_node[i]->feat_count;
                              tip = features_node[i]->feat_type;
                //вывести в информационное окно тип и количество элемента
                              UF_UI_write_listing_window(tip);
                              sprintf_s(mes, "- %d шт.\n", cnt);
                              UF_UI_write_listing_window(mes);
                //выбрать по типу элемента процедуру для его обработки
                              if (strcmp(tip, "SPHERE") == 0)
                              SphereSub(features_node[i]->feat_tags, cnt);
                              if (strcmp(tip, "BLEND") == 0)
                              BlendSub(features_node[i]->feat_tags, cnt);
                              if (strcmp(tip, "CYLINDER") == 0)
                        CylinderSub(features_node[i]->feat_tags, cnt);
                        } //освободить память от динамической переменной
                        UF_free(features_node);
                        //убрать цвет выделения с выбранного тела сцены
                        UF_DISP_set_highlight(body, 0);
                     }
                     else //иначе, сообщить об ошибке
                UF_UI_write_listing_window("\n Это НЕ твердое тело! \n");
      //продолжать цикл, пока пользователь нажимает кнопки ОК или ПРИНЯТЬ
                }
           } while ((response == 4) || (response == 5));
           UF_terminate();
      }
}
//типовая функция завершения приложения
int ufusr_ask_unload(void) {
      return (UF_UNLOAD_IMMEDIATELY);
}
```

```
//процедура анализа параметров сферы
//на входе в процедуру передается:
// параметр 1 - указатель на массив элементов типа сфера
// параметр 2 - количество элементов в массиве параметра 1
void SphereSub(tag_p_t ftags, int nt)
{
	int i, edit = 1;
	char * diameter;
	//выполняется цикл по количеству переданных в процедуру сфер
	for (i = 0; i < nt; i++)
	{
		//получить параметры сферы
		UF_MODL_ask_sphere_parms(ftags[i], edit, &diameter);
		//вывести параметры в информационное окно
		sprintf_s(mes, " Диаметр: %s мм.\n", &diameter[0]);
		UF_UI_write_listing_window(mes);
		//освободить полученную строку с диаметром сферы
		UF_free(diameter);
		//запустить процедуру поиска «родителей» и «детей» текущей сферы
		PatternAndChildrenSub(ftags[i]);
	}
	return;
}
//Процедура анализа скругления
//на входе: параметр 1 - указатель на массив тегов скруглений
//параметр 2 - количество элементов в массиве параметра 1
void BlendSub(tag_p_t ftags, int nt) {
	int i;
	int edit = 1;
	char *radius;
	//цикл по количеству скруглений
	for (i = 0; i < nt; i++)
	{ //функция получения параметров скругления
		UF_MODL_ask_blend_parms(ftags[i], edit, &radius);
```

```
            //вывод на экран текста полученного параметра
            sprintf_s(mes, " Радиус : %s мм.\n", radius);
            //освобождение полученных динамических переменных
            UF_UI_write_listing_window(mes);
            UF_free(radius);
            //вызов процедуры определения родителей и детей
            PatternAndChildrenSub(ftags[i]);
      }
      return;
}
//Процедура анализа цилиндра
//на входе: параметр 1 - указатель на массив тегов цилиндров
//параметр 2 - количество элементов в массиве параметра 1
void CylinderSub(tag_p_t ftags, int nt) {
      int i;
      int edit = 1;
      char *diameter;
      char *height;
      //цикл по количеству цилиндров
      for (i = 0; i < nt; i++)
      { //функция получения параметров цилиндра
            UF_MODL_ask_cylinder_parms(ftags[i], edit, &diameter, &height);
            //вывод на экран текста полученного параметра
            sprintf_s(mes, " Диаметр : %s мм Высота : %s мм\n", diameter,
height);
            //освобождение полученных динамических переменных
            UF_UI_write_listing_window(mes);
            UF_free(diameter); UF_free(height);
            //вызов процедуры определения родителей и детей
            PatternAndChildrenSub(ftags[i]);
      } return;
}
//процедура определения «родителей» и «детей» элементов 3D тела
//параметр процедуры на входе: тег элемента, «родителей» и «детей»
```

```
// которого надо найти
void PatternAndChildrenSub(tag_t ftag)
{
	int j, k, num_parents, num_children;
	tag_t *parent_array, *children_array;
	char *feature_type;
	int num_cons;
	tag_t *con_tags;
	uf_list_p_t object_list;
	UF_SKET_con_type_t con_type;
	char name[UF_OBJ_NAME_LEN + 1];
	//определение родителей и детей
	UF_MODL_ask_feat_relatives(ftag, &num_parents,
		&parent_array, &num_children, &children_array);
	//вывод в информационное окно переченя родителей
	sprintf_s(mes, " Родителей=%d шт.\n", num_parents);
	UF_UI_write_listing_window(mes);
	for (j = 0; j < num_parents; j++) //цикл по количеству родителей
	{//определение типа родителя
		UF_MODL_ask_feat_type(parent_array[j], &feature_type);
		//вывод типа родителя в информационное окно
		sprintf_s(mes, "%s\n", feature_type);
		UF_UI_write_listing_window(mes);
		sprintf_s(mes, "%s\n", "тип родителя");
		UF_UI_write_listing_window(mes);
		//если родитель ЭСКИЗ
		if (strcmp(feature_type, "SKETCH") == 0) {
			//то дополнительно запросим состав элементов эскиза
			UF_SKET_ask_feature_sketches(parent_array[j], &object_list);
			//на первом элементе эскиза проверим наличие размеров
			UF_SKET_ask_constraints_of_sketch(object_list->eid,
(UF_SKET_con_class_t)UF_SKET_dim_cons,
				&num_cons, &con_tags);
			//выводим информацию о размерах эскиза на экран
```

```
        sprintf_s(mes, " Размерных элементов скетча = %d шт.\n", num_cons);
                UF_UI_write_listing_window(mes);
                //очищаем все полученные динамические переменные
                UF_free(con_tags);
                UF_MODL_delete_list(&object_list);
        }
        //очищаем полученную в начале процедуры переменную типа родителя
        UF_free(feature_type);
    }
    //аналогично составляем перечень «детей»
    //выводим на экран число «детей»
    sprintf_s(mes, " Количество детей=%d шт.\n", num_children);
    UF_UI_write_listing_window(mes);
    //и для каждого «ребёнка»
    for (j = 0; j < num_children; j++) {//определяем его тип
        UF_MODL_ask_feat_type(children_array[j], &feature_type);
        //и выводим этот тип на экран
        sprintf_s(mes, "%s\n", feature_type);
        UF_UI_write_listing_window(mes);
        //освобождаем переменную с полученным типом
        UF_free(feature_type);
    }
    //выводим декоративную черту для отделения одного элемента от другого
    UF_UI_write_listing_window("---------------------- - \n");
    //освобождаем массивы «родителей» и «детей»
    UF_free(parent_array);
    UF_free(children_array);
}
```

В результате выполнения программного кода будет получен результат анализа твердого тела, приведенный на рисунке 3.11.

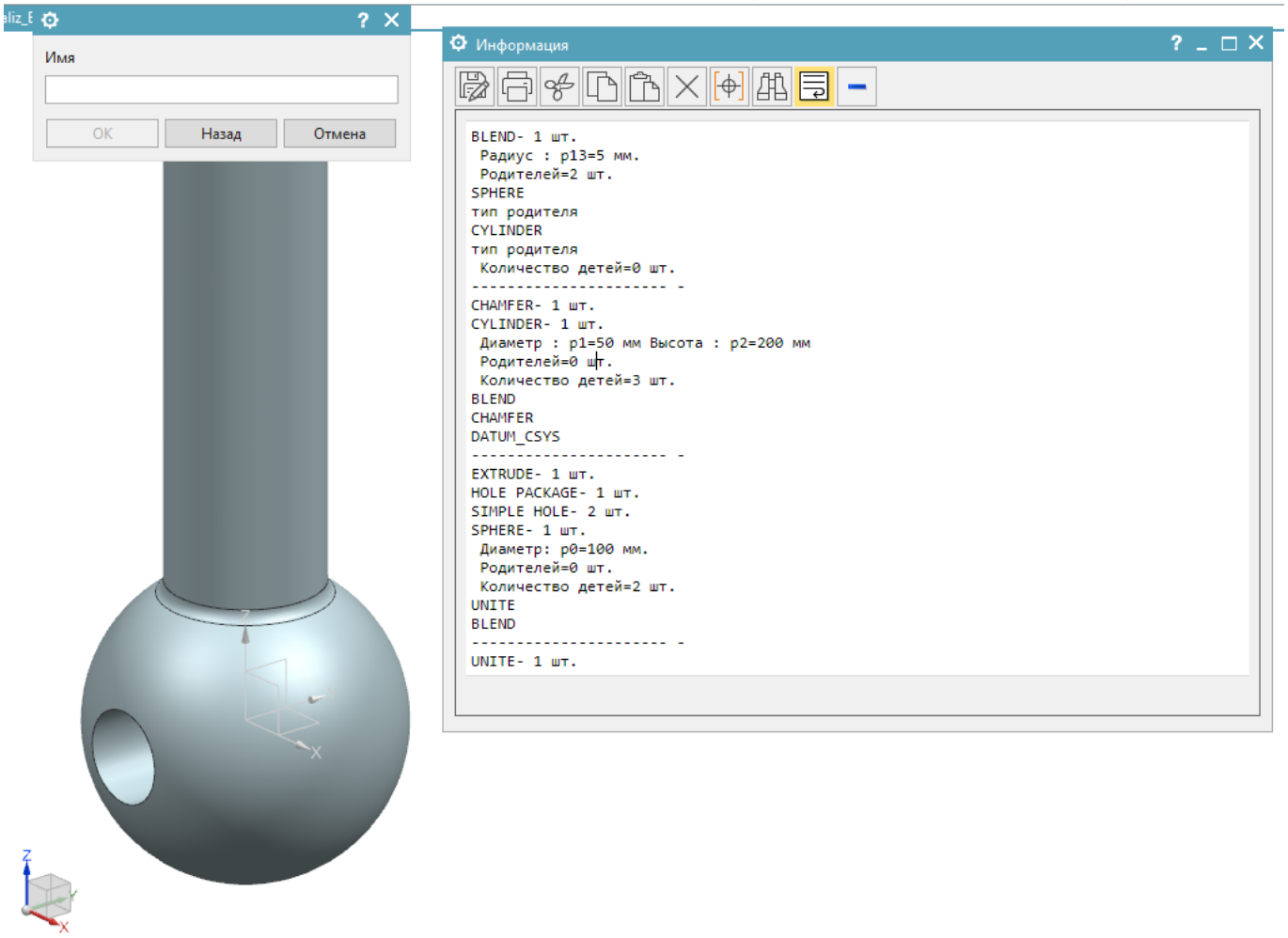


Рисунок 3.11 – Результат работы модуля анализа твердого тела

3.7 Задание для лабораторной работы

1 Изучить теоретический материал.

2 Разработать прикладную библиотеку NX для заданного варианта.

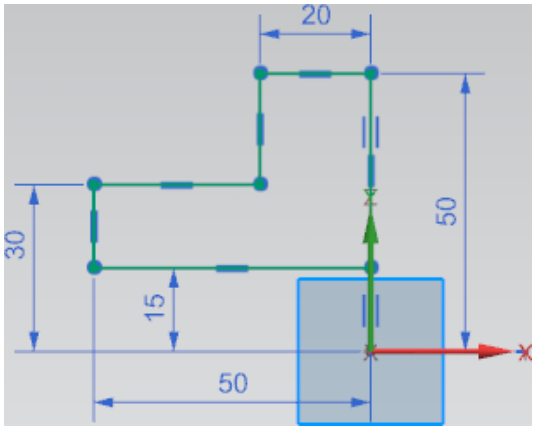
Вариант 1. Создайте программный модуль, строящий три цилиндра, пересекающиеся друг с другом под прямым углом (наподобие осей системы координат).

Вариант 2. Постройте программным путём тор, вращая окружность вокруг заданной оси. Ось должна выбираться пользователем.

Вариант 3. Создайте программу, которая автоматически удаляет с рабочей сцены все тела заданного типа (например-сферы).

Вариант 4. Составьте программу для автоматического построения квадрата, два параллельных ребра которого составлены из сфер, а вторы два ребра составлены из кубов. Размеры элементов задаются программой.

Вариант 5. Составьте программу для автоматического построения фланца в соответствии с рисунком, используя операцию вращения.



Вариант 6. Составьте программу для автоматического построения параллелепипеда, ребра которого составлены из сфер заданного диаметра.

Вариант 7. В листинге 3.5 добавьте анализ логической операции объединения.

Вариант 8. В листинге 3.5 добавьте анализ операции выдавливания.

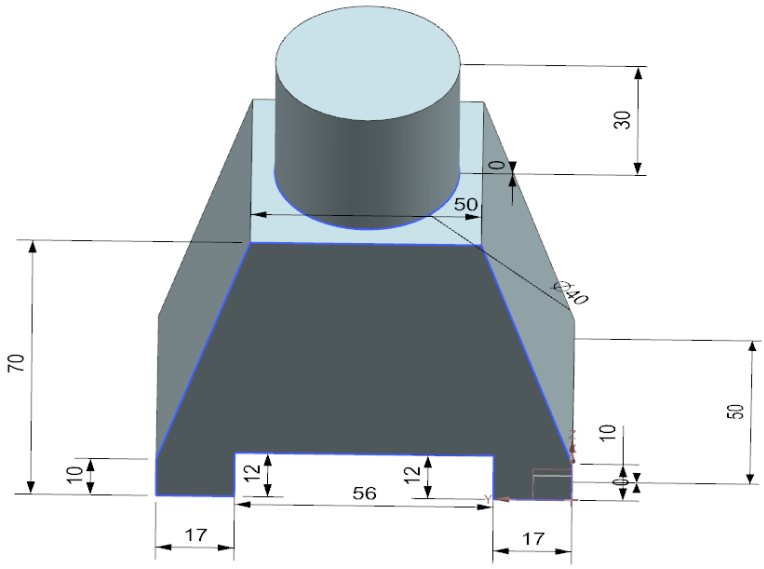
Вариант 9. В листинге 3.5 добавьте анализ отверстия.

Вариант 10. В листинге 3.5 добавьте анализ параллелепипеда.

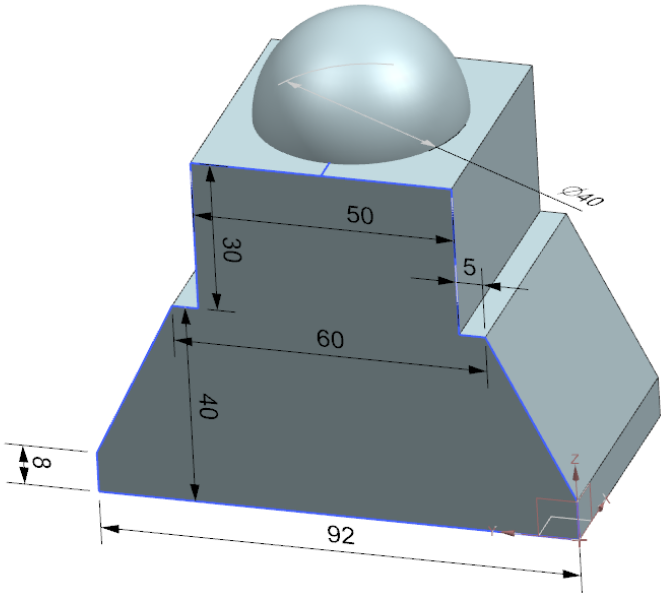
Вариант 11. Напишите программный модуль, который в автоматическом режиме находит на сцене NX все сферы и увеличивает их диаметр на 20 %.

Вариант 12. Напишите программный модуль, который в автоматическом режиме находит на сцене NX все цилиндры и увеличивает их диаметр на 30 % и длину на 50 %.

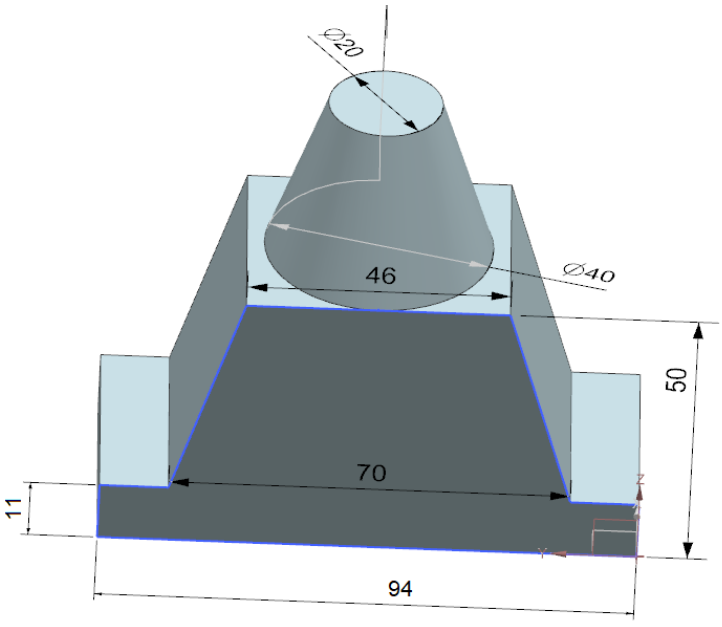
Вариант 13. Напишите программный модуль, который строит тело, показанное на рисунке



Вариант 14. Напишите программный модуль, который строит тело, показанное на рисунке



Вариант 15. Напишите программный модуль, который строит тело, показанное на рисунке



Вариант 16. Составьте программу для автоматического построения пирамиды, ребра которой составлены из сфер заданного диаметра.

4 Подготовить ответы на контрольные вопросы.

3.8 Содержание отчета

В отчете по лабораторной работе согласно СТО 02069024. 101 – 2015 [5] должны содержаться следующие пункты:

- название лабораторной работы;
- цель работы;
- задание на лабораторную работу;
- код разработанной программы с комментариями;
- экранные формы с отображением результата выполнения программы в системе NX;
- выводы;
- список использованных источников.

3.9 Контрольные вопросы

1) Функция построения цилиндра `UF_MODL_create_cyl1`.
2) Функция вращения `UF_MODL_create_revolved`.
3) Функция выдавливания `UF_MODL_create_extruded`.
4) Удаление объектов `UF_OBJ_delete_object`.
5) Диалог выбора `UF_UI_select_single.`
6) Принцип поиска информации о функциях Open API в установленной системе NX.
7) Структура `UF_UI_selection_options_t.`
8) Какие параметры области выбора объектов `scope` возможны?
9) Структура маски `UF_UI_mask.`
10) Копирование объектов.

11) Копирование составного объекта.

12) Опишите дерево структуры твердого тела.

13) Функция получения элементов, составляющих тело `UF_MODL_ask_body_features.`

14) Имена типов элементов, встречающиеся при анализе тела в NX.

4 Разработка интерфейса пользователя

4.1 Общие сведения об инструменте - «Разработчик пользовательского интерфейса»

Начиная с NX v.6.0 в NX появился новый инструмент создания диалоговых окон - «Разработчик пользовательского интерфейса», который запускается из меню «Дополнительно» на закладке «Приложения» (рисунок 4.1).

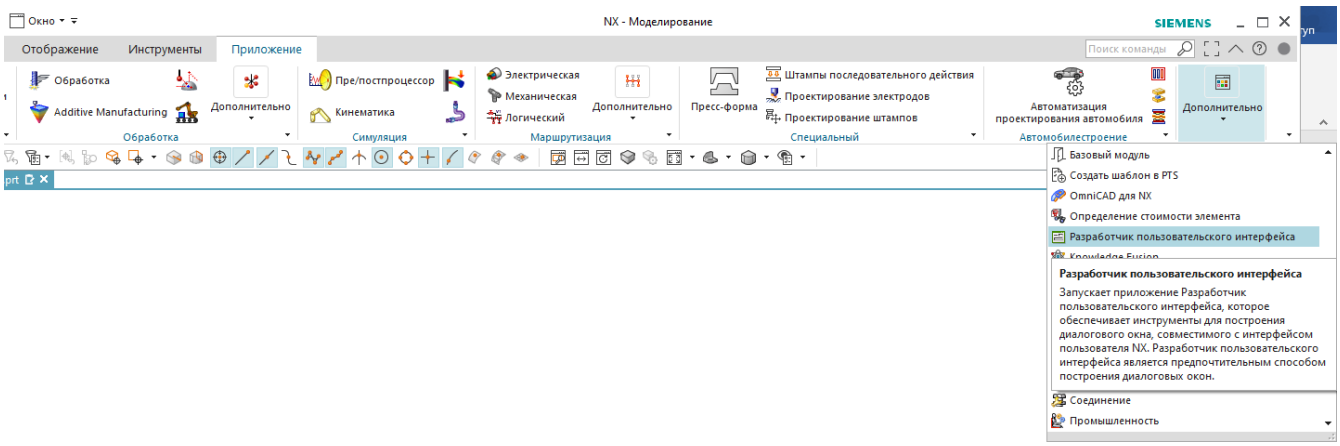


Рисунок 4.1 – Запуск инструмента «Разработчик пользовательского интерфейса»

В результате запустится инструмент как показано на рисунке 4.2.

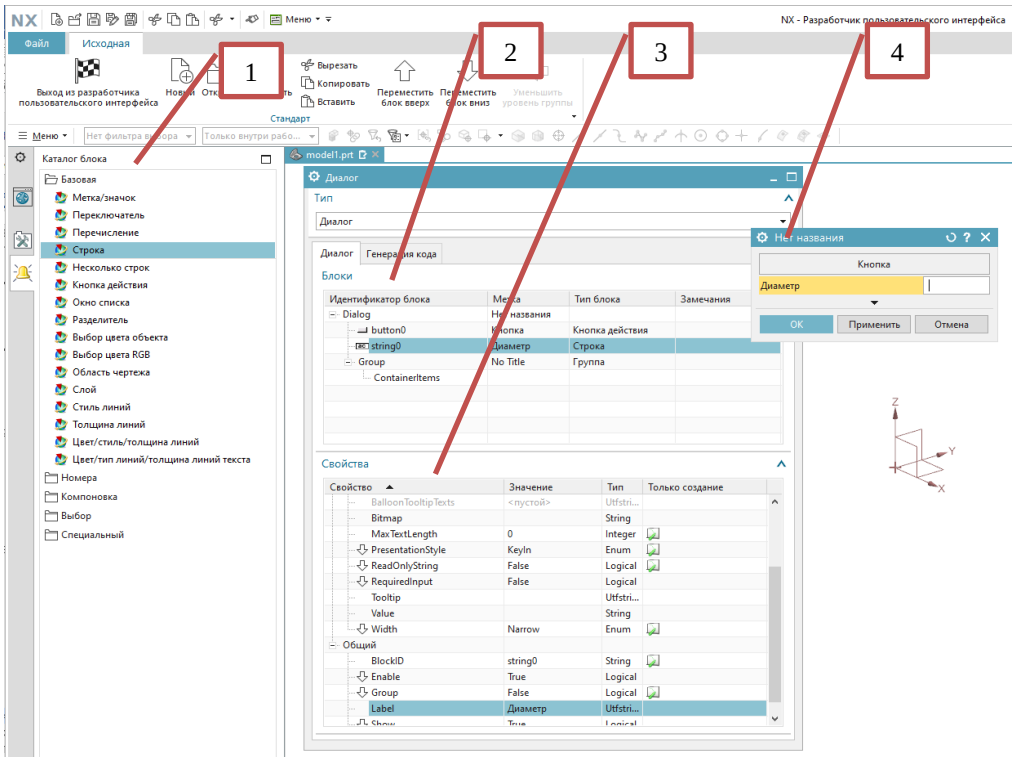


Рисунок 4.2 – Внешний вид разработчика пользовательского интерфейса

Все управляющие элементы, которые можно разместить на диалоге, структурированы, сгруппированы и размещены на раскрывающихся вкладках в «Каталоге блока», рисунок 4.2, п.1. Окно конструктора диалога, рисунок 4.2, п.2, разбито на две закладки:

- «Диалог» - с двумя списками: «Блоки», где отображается текущая структура и состав компонентов строящегося диалога и «Свойства», рисунок 4.2, п.3, где устанавливаются все параметры выбранного элемента;

- «Генерация кода» - (рисунок 4.3), где настраиваются особенности генерирования программного кода модуля диалога, выбирается язык программирования и задаётся наличие точек входа обработчиков событий для диалога. Обработчики событий управляющих элементов в конструкторе не настраиваются, они создаются в коде автоматически.

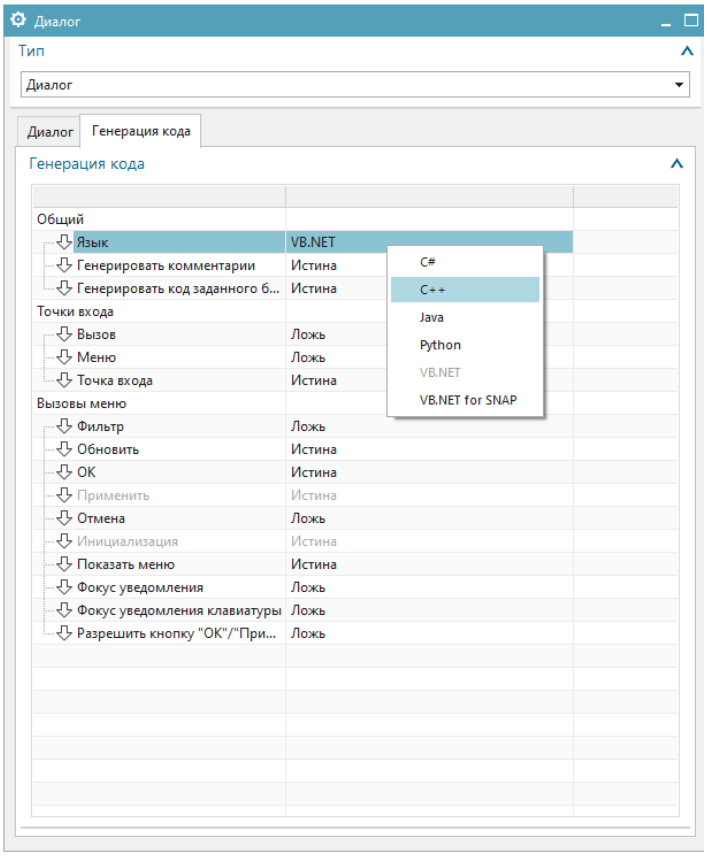


Рисунок 4.3 – Вкладка «Генерация кода»

При выборе элемента в конструкторе, этот элемент активизируется и на диалоге рисунок 4.2, п.4. Для некоторых элементов работает и обратная активизация.

«Разработчик пользовательского интерфейса» был создан для разработки интерфейса для внутренних программных модулей, разрабатываемых с использованием механизмов объектно-ориентрованного программирования. Поэтому среди диалектов языка С он ориентирован на С++ (С++.Net) и С#. Программный код, который генерирует «Разработчик пользовательского интерфейса» под созданный диалог, использует классы, свойства и методы объектов.

Для примера рассмотрим процесс разработки диалога для программы построения цилиндра, приведенный на рисунке 4.4. В качестве входных параметров выступают длина и диаметр цилиндра. Построение будет выполняться при нажатии на кнопку «Построить».

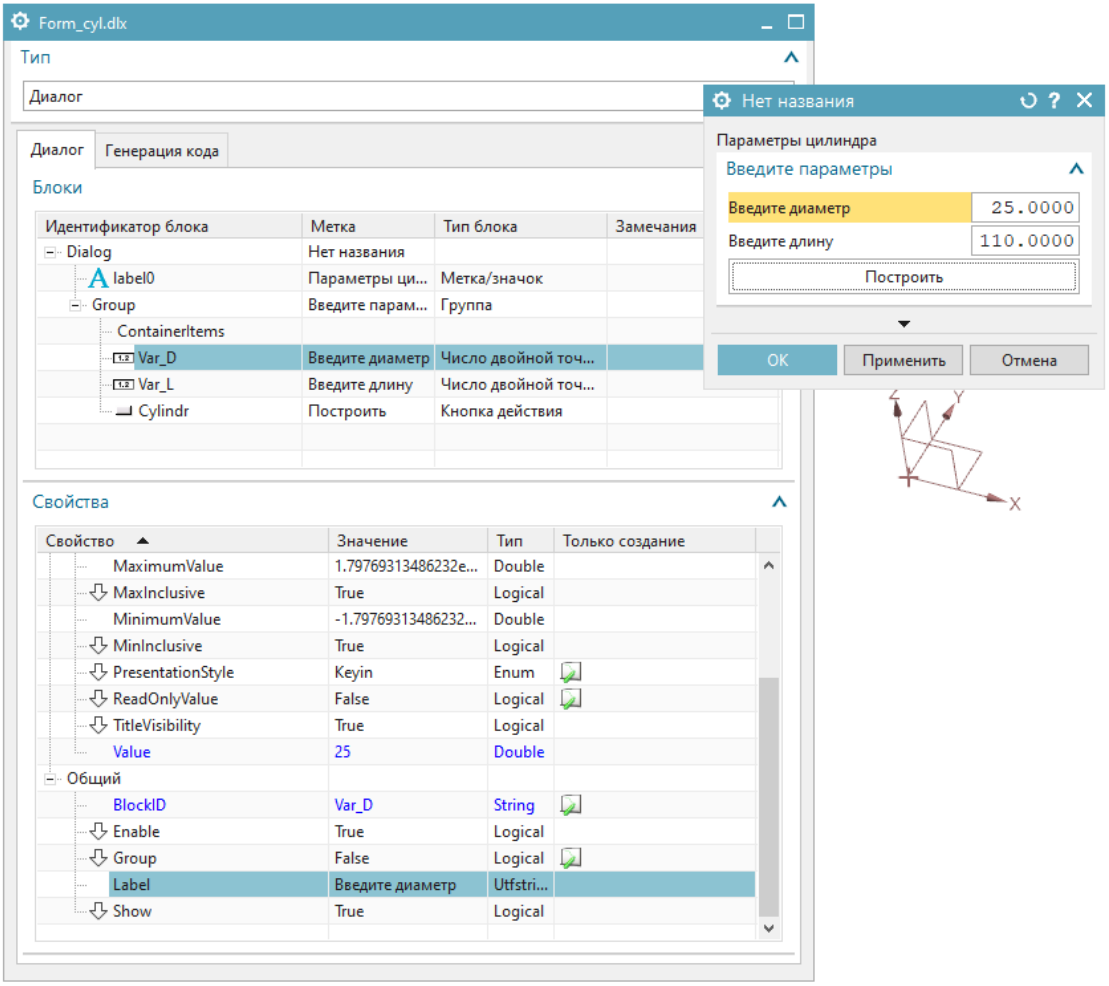


Рисунок 4.4 – Диалог построения цилиндра

Зададим свойства элементов в соответствии с рисунком 4.4. Важным свойством при проектировании диалога является свойство «BlockID». В дальнейшем (в программном коде) оно становится именем объекта со всеми его свойствами и методами. В представленном примере кнопка имеет идентификатор Cylindr, а строки для ввода параметров - *Var_D* и *Var_L*.

После сохранения, система сгенерирует файлы с расширениями *.cpp, *.hpp и *.dlx. В файле *.hpp приведено описание имен всех обработчиков событий на диалоге, в файле *.cpp – программный код для работы с диалогом, выполненный в стиле объектно-ориентированного программирования. Участок этого кода, где нам необходимо было вставить собственный программный код - реакцию на щелчки мыши на кнопке «Построить», приводится в листинге 4.1.

Листинг 4.1.

```
//Программа построения цилиндра
#include <stdio.h>
#include <uf.h>
#include <uf_modl.h>
#include <stdlib.h>
#include <iostream>
//...
//Здесь расположен пропущенный фрагмент кода, который не изменялся
//...
//------------------------------------------------------------------------------
//Callback Name: apply_cb
//------------------------------------------------------------------------------
//stringstream sstr;
int Form_cyl::apply_cb()
{
      int errorCode = 0;
      try
      {
            //---- Enter your callback code here -----
            }
      catch (exception& ex)
      {
            //---- Enter your exception handling code here -----
            errorCode = 1;
            Form_cyl::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
      }
      return errorCode;
};
//------------------------------------------------------------------------------
//Callback Name: update_cb
//------------------------------------------------------------------------------
int Form_cyl::update_cb(NXOpen::BlockStyler::UIBlock* block)
```

```
{
	try
	{
		if (block == label0)
		{
			//---------Enter your code here-----------
		}
		else if (block == Var_D)
		{
			//---------Enter your code here-----------
		}
		else if (block == Var_L)
		{
			//---------Enter your code here-----------

		}
		else if (block == Cylindr)
		{
			//---------Enter your code here-----------
					// определим создание нового самостоятельного тела
			UF_FEATURE_SIGN sign = UF_NULLSIGN;
			//это тело должно начинаться в заданных координатах X, У, Z
			double cyl_orig[3] = { 5.0,10.0,15.0 };
			char height[125]; //высота цилиндра
		//в переменную height передаем значение из строки диалога Var_L
			sprintf(height, "%f", Var_L->Value());
			char diam[125]; //диаметр
			//в переменную diam передаем значение из строки диалога Var_D
			sprintf(diam, "%f", Var_D->Value());
			//орты направления главной диагонали координат будут по 1
			double direction[3] = { 1.0, 1.0, 1.0 };
		tag_t cyl_obj; //определим переменную для будущего тега цилиндра
			if (!UF_initialize()) {
			//выполним построение цилиндра
```

```
        UF_MODL_create_cyl1(sign, cyl_orig, height, diam, direction, &cyl_obj);
                UF_terminate();
                }
            }
        }
        catch (exception& ex)
        {
            //---- Enter your exception handling code here -----
            Form_cyl::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
        }
        return 0;
}
//------------------------------------------------------------------------------
//Callback Name: ok_cb
//------------------------------------------------------------------------------
int Form_cyl::ok_cb()
{
        int errorCode = 0;
        try
        {
            errorCode = apply_cb();
        }
        catch (exception& ex)
        {
            //---- Enter your exception handling code here -----
            errorCode = 1;
            Form_cyl::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
        }
        return errorCode;
        }
```

Как видно из Листинга 4.1, для обработки событий нажатия основных кнопок диалога («ОК», «ПРИНЯТЬ») используются отдельные процедуры (методы), соответственно: `ok_cb()` , `apply_cb()`, а для обработки событий на элементах диалога, созданных пользователем - общая процедура (метод) `update_cb (...)`. Параметром этой процедуры является идентификатор элемента - `UIBlock* block`, по имени которого в процедуре делается разветвление на обработчики событий от этого элемента.

Так как кнопка «Построить» имеет идентификатор Cylindr, то программный код создания цилиндра должен быть размещен внутри операторных скобок

```
else if (block == Cylindr)
        {
/*программный код построения цилиндра */
    }
```

Все процедуры состоят из связок `try { } catch () { },` что позволяет эффективно проводить обработку возникающих системных ошибок.

Стиль написания программного кода соответствует объектному программированию в классах C++, поэтому видится рациональным использование инструмента «Разработчик пользовательского интерфейса» для разработки диалоговых окон программных модулей именно такого стиля (а так же C++.Net, C#).

Файл *.dlx содержит описание, собственно, разработанного диалога. Для того чтобы прикладная библиотека с разработанным пользовательским интерфейсом запустилась в NX, необходимо файл с расширением *.dlx скопировать в каталог UGII, расположенный в каталоге с установленной программой, например, c:\Program Files\Siemens\NX_v_\UGII, где _v_ - версия NX.

4.2 Построение трехмерной модели по заданным размерам

Ознакомившись с инструментом - «Разработчик пользовательского интерфейса» рассмотрим пример создания диалогового окна для создания трехмерной модели

по заданным размерам. Для этого разместим компонент «Область чертежа», позволяющий загрузить в него графическое изображение (рисунок 4.5).

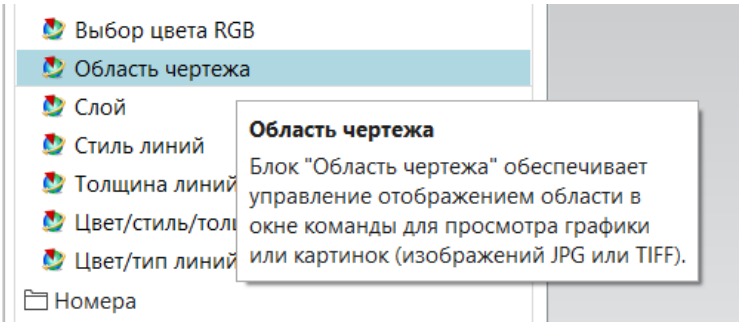


Рисунок 4.5 – Выбор компонента «Область чертежа»

В настройках этого компонента необходимо задать только одно свойство: Image, в котором прописывается путь к файлу с требуемым изображением. Изображение необходимо предварительно подготовить в графическом редакторе. Для примера рассмотрим изображение трехмерной модели с проставленными размерами в соответствии с рисунком 4.6.

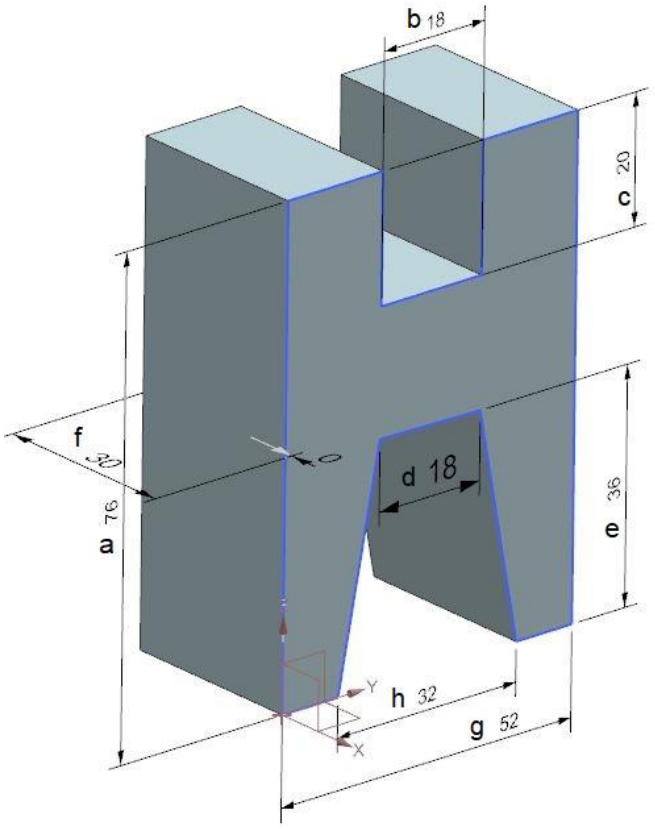


Рисунок 4.6 – Изображение трехмерной модели для диалога пользователя

На рисунке 4.6 видно, что каждому размеру поставлено в соответствие символьное значение. Это сделано для того, чтобы можно было корректно задать идентификаторы переменных. При задании компонентов воспользуемся двумя возможными типами компонентов – «Строка» из раздела «Базовая» и «Число двойной точности» из раздела «Номера». Для компонентов ввода информации зададим параметры в соответствии с таблицей 4.1.

Таблица 4.1 – Параметры компонентов ввода информации

Размер	Тип блока	BlockID	Label	Value
a	Строка	Ra	a=	76
b	Строка	Rb	b=	18
c	Число двойной точности	Rc	c=	20
d	Число двойной точности	Rd	d=	18
e	Число двойной точности	Re	e=	36
f	Число двойной точности	Rf	f=	30
g	Число двойной точности	Rg	g=	52
h	Число двойной точности	Rh	h=	32

Построение модели будет выполняться при нажатии на кнопку «Построить модель» с идентификатором block_create. Итоговый внешний вид разрабатываемого диалога представлен на рисунке 4.7.

После создания исходных файлов *.cpp, *.hpp и *.dlx необходимо разработать программный код построения трехмерной модели с учетом введенных пользователем размеров. Так как весь код расположен внутри процедуры `update_cb()` то в Листинге 4.2 представим только ее значимый фрагмент. Полностью программный код модуля приведен в Приложении Б.

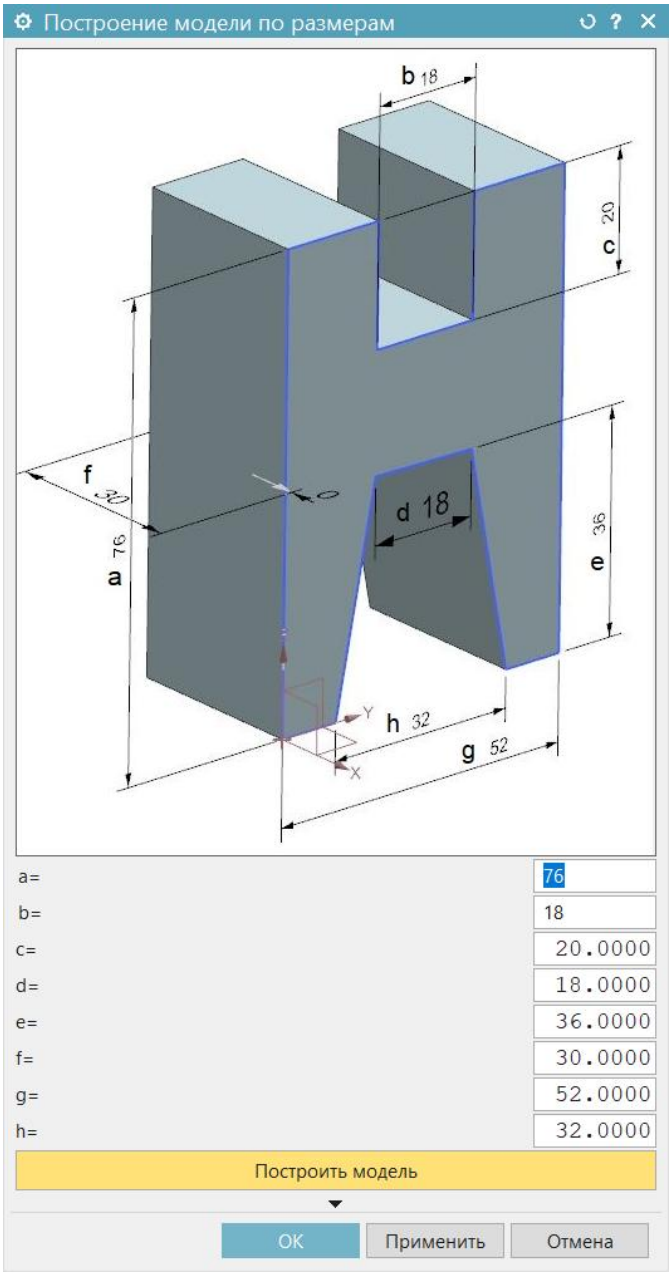


Рисунок 4.7 – Разработанный диалог построения трехмерной модели по заданным размерам

Листинг 4.2.

```
//Программа построения тела
//при помощи операции выдавливания
#include <uf.h>
#include <uf_curve.h>
#include <uf_csys.h>
#include <uf_modl.h>
```

```
//------------------------------------------------------------------------
//Callback Name: update_cb
//------------------------------------------------------------------------
/* Определяется массив линий с координатами их
концов, используемых в эскизе выдавливаемого сечения */
UF_CURVE_line_t line[12];
tag_t objarray[12]; //теги для трех линий и дуги
// процедура построения отрезка
void DrawElem(int i, double y, double z, double y1, double z1)
{
      //Так как сечение будет располагаться в плоскости ZY
      // координата X для всех объектов будет равна 0
      double x = 0;
      line[i].start_point[0] = x;// X1
      line[i].start_point[1] = y;// Y1
      line[i].start_point[2] = z;// Z1
      line[i].end_point[0] = x;// X2
      line[i].end_point[1] = y1;// Y2
      line[i].end_point[2] = z1;// Z2
      // построение линии
      UF_CURVE_create_line(&line[i], &objarray[i]);
}
int Form_H::update_cb(NXOpen::BlockStyler::UIBlock* block)
{
      try
      {
            //...
            //обработчики событий для других блоков не показаны
            //...
            }
            else if (block == but_create)
            {
                  //---------Enter your code here-----------
```

```
                //параметры операции выдавливания
              //расстояние начала и расстояние окончания выдавливания
                char a[] = "0.0";
                char b[125];
                sprintf(b, "%f", Rf->Value());
                char *limit[2] = { a, b };
                //орты вектора направления выдавливания
                double direction[3] = { 1.0, 0.0, 0.0 };
                //признак создания самостоятельного тела
                UF_FEATURE_SIGN create = UF_NULLSIGN;
                //заготовки указателей на перечни объектов
          uf_list_p_t loop_list, //Список объектов, подлежащих выдавливанию
(элементы сечения)
                     features; //Список созданных идентификаторов объектов
                //угол конусности в градусах
                char t_a[] = "0.0";
                char *taper_angle = t_a;
                //параметр не используется
                double ref_pt[3];
                double y, z, y1, z1;
                if (!UF_initialize())
                {
                     //левая вертикаль
                     y = 0; z = 0; y1 = 0;
                     z1 = atof(Ra->Value().getText());
                     DrawElem(0, y, z, y1, z1);
                     //верхняя левая горизонталь
                     y = 0; z = z1;
                     y1 = (Rg->Value() - atof(Rb->Value().getText())) / 2;
                     DrawElem(1, y, z, y1, z1);
                     //вниз левая короткая
               y = y1; z = z1; z1 = atof(Ra->Value().getText())-Rc->Value();
                     DrawElem(2, y, z, y1, z1);
                     //верхняя короткая
```

```
                    y = y1; z = z1; y1 = y1+atof(Rb->Value().getText());
                    DrawElem(3, y, z, y1, z1);
                    //вверх правая короткая
                    y = y1; z = z1; z1 = atof(Ra->Value().getText());
                    DrawElem(4, y, z, y1, z1);
                    //верхняя правая горизонталь
                    y = y1; z = z1; y1= Rg->Value();
                    DrawElem(5, y, z, y1, z1);
                    //правая вертикаль вниз
                    y = y1; z = z1; z1 = 0;
                    DrawElem(6, y, z, y1, z1);
                    //правая нижняя короткая
     y = y1; z = z1; y1 = Rg->Value() - (Rg->Value() - Rh->Value()) / 2;
                    DrawElem(7, y, z, y1, z1);
                    //нижняя наклонная правая
     y = y1; z = z1; y1 = Rg->Value() - (Rg->Value() - Rd->Value()) / 2; z1
= Re->Value();
                    DrawElem(8, y, z, y1, z1);
                    //нижняя средняя
                    y = y1; z = z1; y1 = (Rg->Value() - Rd->Value()) / 2;
                    DrawElem(9, y, z, y1, z1);
                    //нижняя наклонная левая
               y = y1; z = z1; y1 = (Rg->Value() - Rh->Value()) / 2; z1 =0;
                    DrawElem(10, y, z, y1, z1);
                    //левая нижняя короткая
                    y = y1; z = z1; y1 = 0;
                    DrawElem(11, y, z, y1, z1);
                    //создание пустого перечня объектов
                    UF_MODL_create_list(&loop_list);
                    //заполнение перечня тремя линиями и одной дугой
                    for (int i = 0; i < 12; i++)
                         UF_MODL_put_list_item(loop_list, objarray[i]);
                    //создание операции выдавливания
                    UF_MODL_create_extruded(loop_list, taper_angle, limit,
```

```
                    ref_pt, direction, create, &features);
                UF_terminate();
            }
        }
    }
    catch (exception& ex)
    {
        //---- Enter your exception handling code here -----
        Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
    }
    return 0;
    }
```

В коде создана отдельная процедура `DrawElem()` для построения отрезка по координатам начальной и конечной точки в двух плоскостях, так как координата X для всех объектов будет равна 0. После создания отрезка его тег помещается в массив `objarray[]` для дальнейшего формирования перечня элементов для операции выдавливания.

В примере намерено в некоторых случаях задано несоответствие типов переменных. Так, при задании параметров первого отрезка, при указании координаты $z1$, используется размер *a*, свойство *Value()*, которого имеет тип `NXString`, чтобы получить значение этого свойства необходимо использовать метод `getText()`, тогда задание значения переменной $z1$ с использованием функции перевода строкового типа в вещественный `atof()` будет выглядеть следующим образом:

```
z1 = atof(Ra->Value().getText());
```

Для перевода вещественного типа в символьный используется функция `sprintf()`:

```
sprintf(b, "%f", Rf->Value());
```

Переменная *b* используется для задания предела выдавливания, который должен иметь тип `char`.

Результат работы программы, представленной в Листинге 4.2 для разных исходных данных приведен на рисунке 4.8.

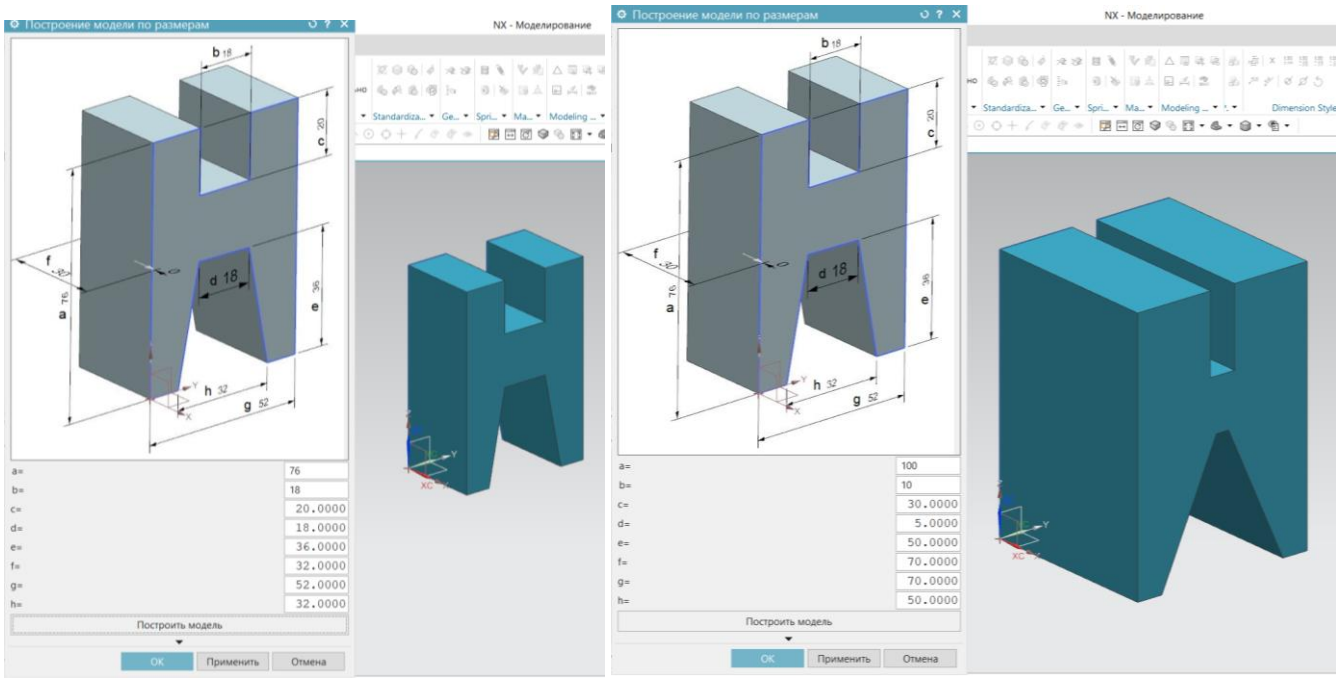


Рисунок 4.8 – Построение трехмерной модели по различным входным параметрам

4.3 Задание для лабораторной работы

1 Изучить теоретический материал.

2 Разработать интерфейс пользователя для прикладной библиотеки NX для заданного варианта трехмерной модели, приведенной в таблице В.1, приложение В.

4 Подготовить ответы на контрольные вопросы.

4.4 Содержание отчета

В отчете по лабораторной работе согласно СТО 02069024. 101 – 2015 [4] должны содержаться следующие пункты:

– название лабораторной работы;

– цель работы;

– задание на лабораторную работу;

– код разработанной программы с комментариями;

– экранные формы с отображением результата выполнения программы в системе NX;

– выводы;

– список использованных источников.

4.5 Контрольные вопросы

1) Как в NX создается интерфейс пользователя?
2) Как выбирается язык генерации программного кода?
3) Для чего используется свойство «BlockID»?
4) Какие файлы генерируются разработчиком пользовательского интерфейса?
5) В какой каталог необходимо поместить файл *.dlx, для корректной работы диалога пользователя в NX?
6) Принцип поиска информации о функциях Open API в установленной системе NX.
7) Как задается обработчик события нажатия кнопки?
8) Как задать обработку нажатия стандартных кнопок («ОК», «ПРИНЯТЬ»)?
9) Какие блоки позволяет задавать «Разработчик пользовательского интерфейса»?
10) Как создать функцию в C++?
11) Как преобразовать значение типа `NXString` в вещественное число?
12) Как преобразовать вещественное значение в символьный тип?

Список использованных источников

1 Тихомиров, В.А. Разработка приложений для NX на языке C / В.А. Тихомиров. – Комсомольск-на-Амуре : Изд-во ФГБОУ ВПО «КнАГТУ», 2011. – 469 с.

2 Краснов, М. Unigraphics для профессионалов / М. Краснов, Ю. Чигишев. – М.: Изд-во «ЛОРИ», 2004. – 320 с.

3 Getting Started with NX Open. – Siemens Product Lifecycle Management Software, 2016. – 137 p.

4 Лафоре, Р. Объектно-ориентированное программирование в C++ / Р. Лафоре. – 4-е изд. – Санкт-Петербург : Питер, 2014. – 928 с.

5 СТО 02069024.110-2008 Издания для образовательного процесса. Общие требования и правила оформления. – Оренбург : ОГУ. – 2017 г. – 74 с.

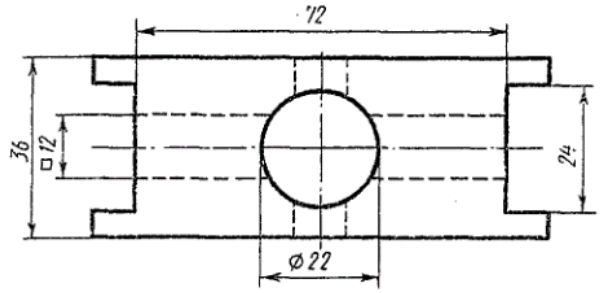
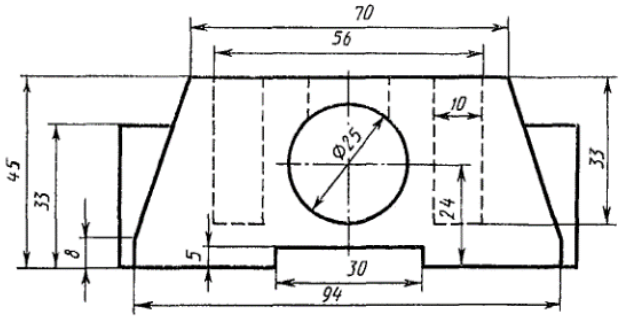
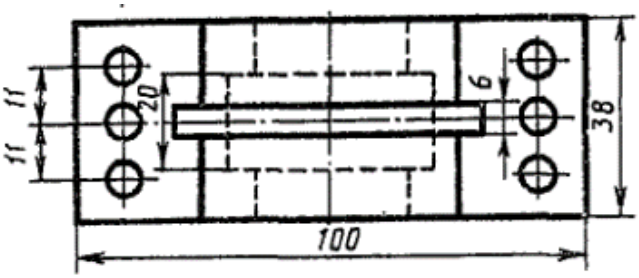
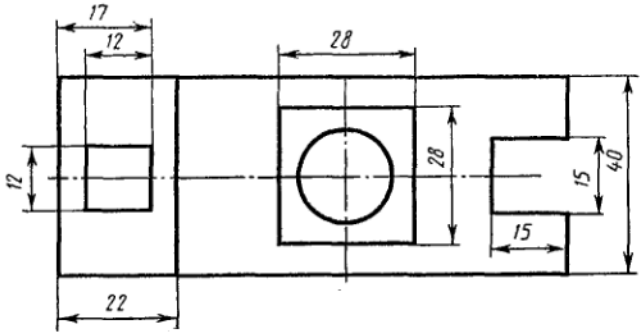
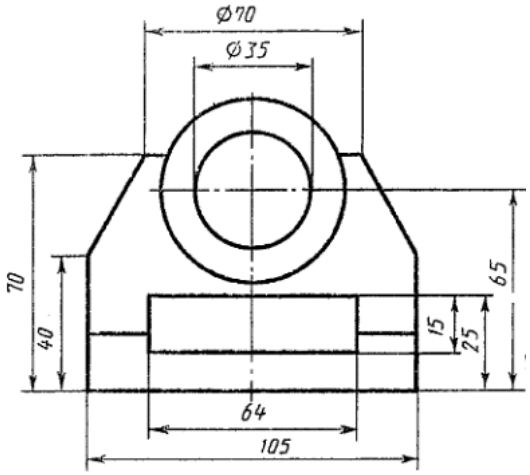
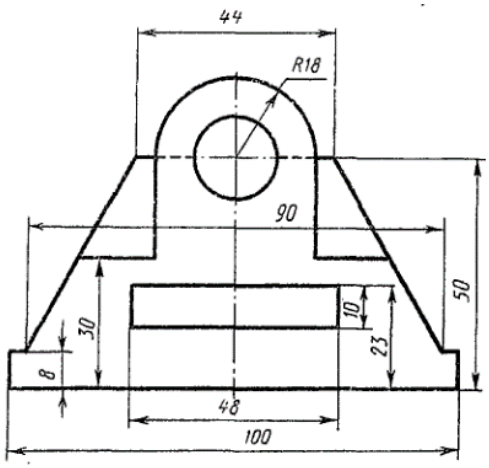
6 Боголюбов С.К. Индивидуальные задания по курсу черчения: Учебное пособие для средних специальных учебных заведений. 3-е изд., стереотипное. Перепечатка со второго издания 1994 г. – М.: ООО ИД «Альянс», 2007. – 368 с.

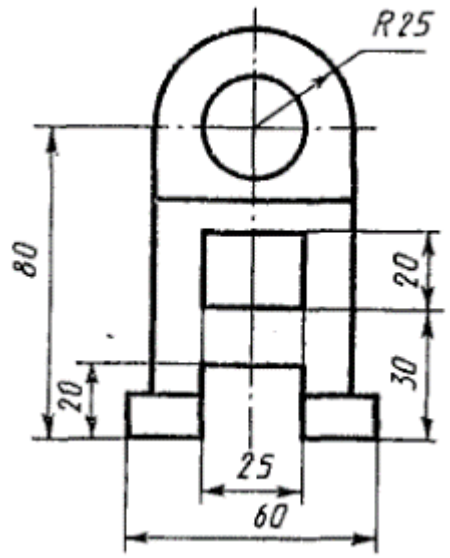
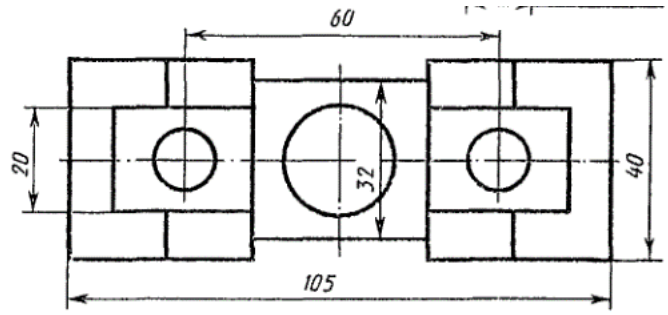
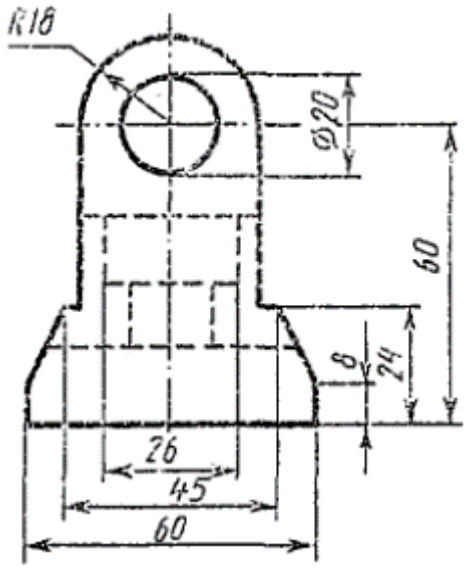
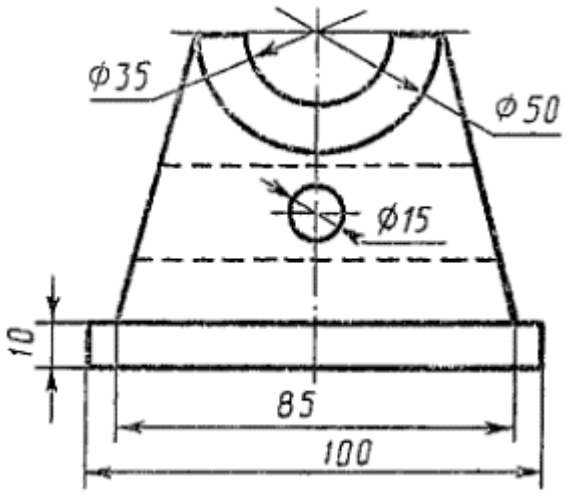
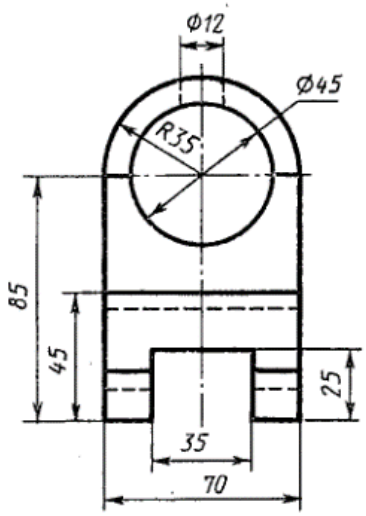
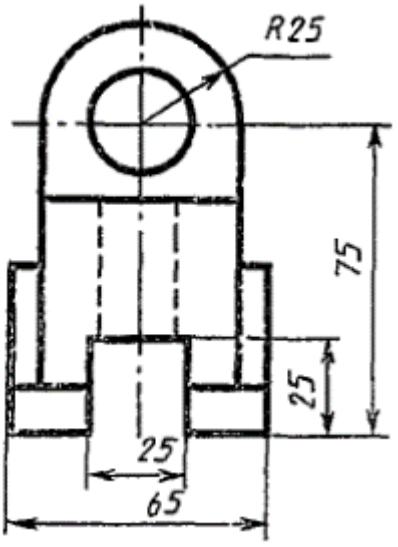
Приложение А
(обязательное)

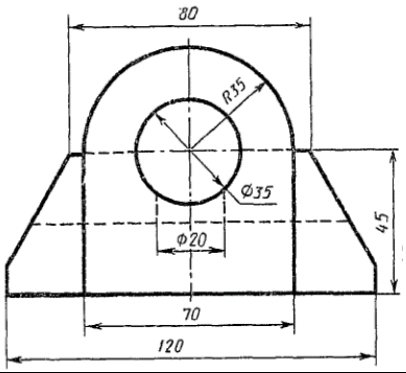
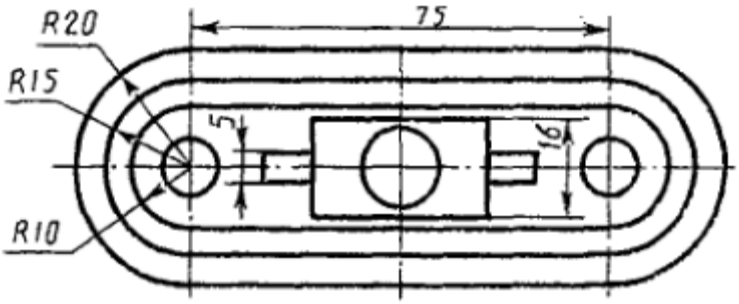
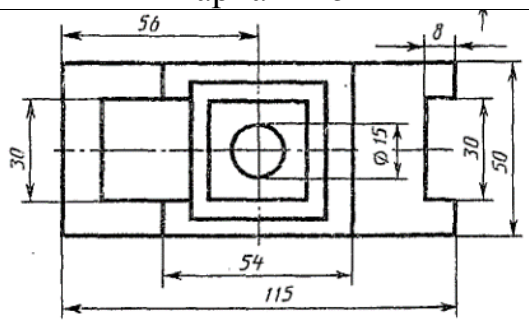
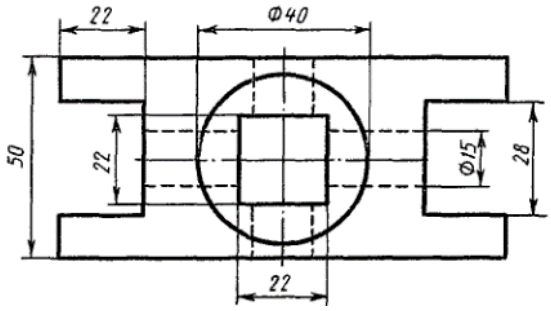
Задания для построения отрезков и дуг окружностей функциями Open NX

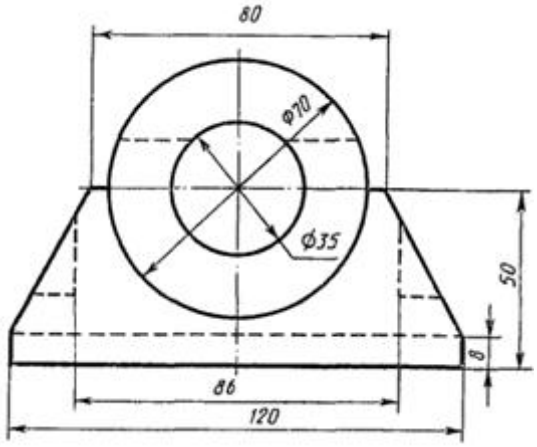
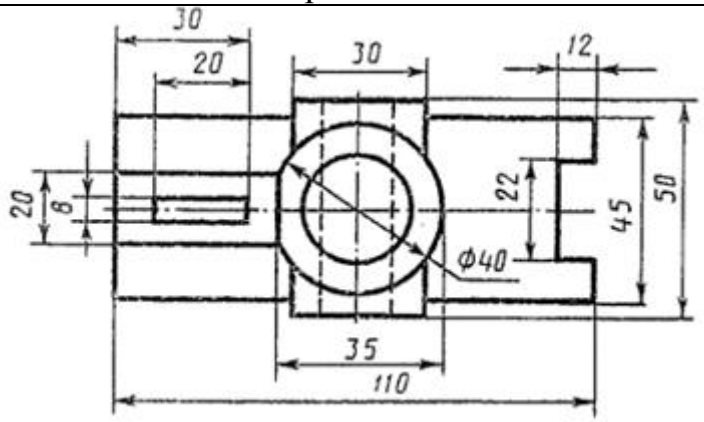
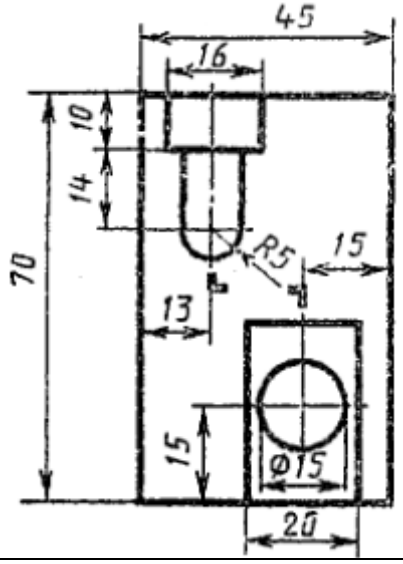
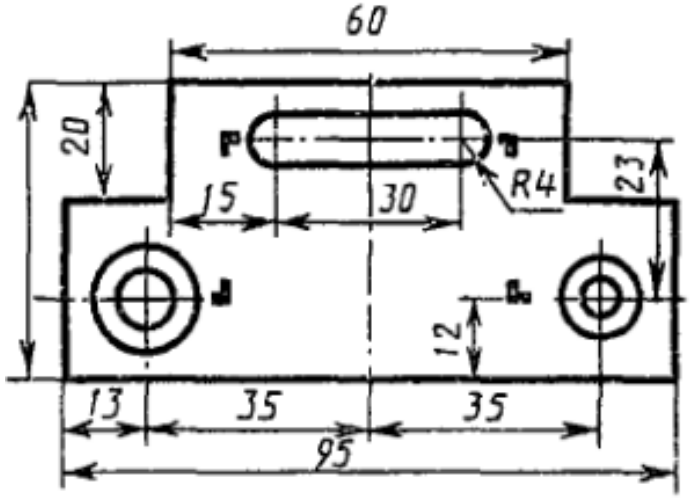
Варианты заданий приведены из учебного пособия [6]. Штриховые линии строить не требуется.

Таблица А.1 – Варианты заданий для построения отрезков и дуг окружностей функциями Open NX

Вариант 1	Вариант 2
	
Вариант 3	**Вариант 4**
	
Вариант 5	**Вариант 6**
	

Вариант 7	Вариант 8
	
Вариант 9	Вариант 10
	
Вариант 11	Вариант 12
	

Вариант 13	Вариант 14
	
Вариант 15	**Вариант 16**
	

Вариант 17	Вариант 18
	
Вариант 19	**Вариант 20**
	

Приложение Б
(обязательное)

Исходный код диалога построения трехмерной модели

```
//Программа построения тела
//при помощи операции выдавливания
#include <uf.h>
#include <uf_curve.h>
#include <uf_csys.h>
#include <uf_modl.h>
//============================================================================
//  WARNING!!  This file is overwritten by the Block UI Styler while generat-
ing
//  the automation code. Any modifications to this file will be lost after
//  generating the code again.
//
//       Filename:  E:\Сергеев\_Работа\_САПР_магистры\3D-модель\Form_H.cpp
//
//        This file was generated by the NX Block UI Styler
//        Created by: Admin
//              Version: NX 1888
//              Date: 03-04-2021  (Format: mm-dd-yyyy)
//              Time: 09:42 (Format: hh-mm)
//
//============================================================================
//============================================================================
//  Purpose:  This TEMPLATE file contains C++ source to guide you in the
//  construction of your Block application dialog. The generation of your
//  dialog file (.dlx extension) is the first step towards dialog construc-
tion
//  within NX.  You must now create a NX Open application that
//  utilizes this file (.dlx).
//
//  The information in this file provides you with the following:
//
//  1.  Help on how to load and display your Block UI Styler dialog in NX
//      using APIs provided in NXOpen.BlockStyler namespace
//  2.  The empty callback methods (stubs) associated with your dialog items
//      have also been placed in this file. These empty methods have been
//      created simply to start you along with your coding requirements.
//      The method name, argument list and possible return values have al-
ready
//      been provided for you.
//============================================================================
//----------------------------------------------------------------------------
//These includes are needed for the following template code
//----------------------------------------------------------------------------
#include "Form_H.hpp"
using namespace NXOpen;
```

```
using namespace NXOpen::BlockStyler;
//------------------------------------------------------------------------------
// Initialize static variables
//------------------------------------------------------------------------------
Session *(Form_H::theSession) = NULL;
UI *(Form_H::theUI) = NULL;
//------------------------------------------------------------------------------
// Constructor for NX Styler class
//------------------------------------------------------------------------------
Form_H::Form_H()
{
	try
	{
		// Initialize the NX Open C++ API environment
		Form_H::theSession = NXOpen::Session::GetSession();
		Form_H::theUI = UI::GetUI();
		theDlxFileName = "Form_H.dlx";
		theDialog = Form_H::theUI->CreateDialog(theDlxFileName);
		// Registration of callback functions
		theDialog->AddApplyHandler(make_callback(this,
&Form_H::apply_cb));
		theDialog->AddOkHandler(make_callback(this, &Form_H::ok_cb));
		theDialog->AddUpdateHandler(make_callback(this,
&Form_H::update_cb));
		theDialog->AddInitializeHandler(make_callback(this,
&Form_H::initialize_cb));
		theDialog->AddDialogShownHandler(make_callback(this,
&Form_H::dialogShown_cb));
	}
	catch (exception& ex)
	{
		//---- Enter your exception handling code here -----
		throw;
	}
}
//------------------------------------------------------------------------------
// Destructor for NX Styler class
//------------------------------------------------------------------------------
Form_H::~Form_H()
{
	if (theDialog != NULL)
	{
		delete theDialog;
		theDialog = NULL;
	}
}
//------------------------------ DIALOG LAUNCHING ------------------------
//
//    Before invoking this application one needs to open any part/empty part
in NX
//    because of the behavior of the blocks.
```

```
//
//    Make sure the dlx file is in one of the following locations:
//        1.) From where NX session is launched
//        2.) $UGII_USER_DIR/application
//        3.) For released applications, using UGII_CUSTOM_DIRECTORY_FILE is
highly
//            recommended. This variable is set to a full directory path to a
file
//            containing a list of root directories for all custom applica-
tions.
//            e.g.,
UGII_CUSTOM_DIRECTORY_FILE=$UGII_BASE_DIR\ugii\menus\custom_dirs.dat
//
//    You can create the dialog using one of the following way:
//
//    1. USER EXIT
//
//        1) Create the Shared Library -- Refer "Block UI Styler programmer's
guide"
//        2) Invoke the Shared Library through File->Execute->NX Open menu.
//
//------------------------------------------------------------------------
extern "C" DllExport void  ufusr(char *param, int *retcod, int param_len)
{
      Form_H *theForm_H = NULL;
      try
      {
            theForm_H = new Form_H();
            // The following method shows the dialog immediately
            theForm_H->Show();
      }
      catch (exception& ex)
      {
            //---- Enter your exception handling code here -----
            Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
      }
      if (theForm_H != NULL)
      {
            delete theForm_H;
            theForm_H = NULL;
      }
      }

//------------------------------------------------------------------------
// This method specifies how a shared image is unloaded from memory
// within NX. This method gives you the capability to unload an
// internal NX Open application or user  exit from NX. Specify any
// one of the three constants as a return value to determine the type
// of unload to perform:
```

```
//    Immediately : unload the library as soon as the automation program has
completed
//    Explicitly  : unload the library from the "Unload Shared Image" dialog
//    AtTermination : unload the library when the NX session terminates
//
//
// NOTE:  A program which associates NX Open applications with the menubar
// MUST NOT use this option since it will UNLOAD your NX Open application im-
age
// from the menubar.
//------------------------------------------------------------------------
extern "C" DllExport int ufusr_ask_unload()
{
     //return (int)Session::LibraryUnloadOptionExplicitly;
     return (int)Session::LibraryUnloadOptionImmediately;
     //return (int)Session::LibraryUnloadOptionAtTermination;
}

//------------------------------------------------------------------------
// Following method cleanup any housekeeping chores that may be needed.
// This method is automatically called by NX.
//------------------------------------------------------------------------
---
extern "C" DllExport void ufusr_cleanup(void)
{
     try
     {
          //---- Enter your callback code here -----
     }
     catch (exception& ex)
     {
          //---- Enter your exception handling code here -----
          Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
     }
}

int Form_H::Show()
{
     try
     {
          theDialog->Show();
     }
     catch (exception& ex)
     {
          //---- Enter your exception handling code here -----
          Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
     }
     return 0;
}
```

```
//------------------------------------------------------------------------------
//----------------------Block UI Styler Callback Functions--------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//Callback Name: initialize_cb
//------------------------------------------------------------------------------
void Form_H::initialize_cb()
{
      try
      {
            group0 = dynamic_cast<NXOpen::BlockStyler::Group*>(theDialog-
>TopBlock()->FindBlock("group0"));
            drawingArea0 = dynam-
ic_cast<NXOpen::BlockStyler::DrawingArea*>(theDialog->TopBlock()-
>FindBlock("drawingArea0"));
            Ra = dynamic_cast<NXOpen::BlockStyler::StringBlock*>(theDialog-
>TopBlock()->FindBlock("Ra"));
            Rb = dynamic_cast<NXOpen::BlockStyler::StringBlock*>(theDialog-
>TopBlock()->FindBlock("Rb"));
            Rc = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Rc"));
            Rd = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Rd"));
            Re = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Re"));
            Rf = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Rf"));
            Rg = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Rg"));
            Rh = dynamic_cast<NXOpen::BlockStyler::DoubleBlock*>(theDialog-
>TopBlock()->FindBlock("Rh"));
            but_create = dynamic_cast<NXOpen::BlockStyler::Button*>(theDialog-
>TopBlock()->FindBlock("but_create"));
      //------------------------------------------------------------------------
      //Registration of StringBlock specific callbacks
      //------------------------------------------------------------------------
      //Ra->SetKeystrokeCallback(make_callback(this, &Form_H::KeystrokeCallback));
      //------------------------------------------------------------------------
      }
      catch (exception& ex)
      {
            //---- Enter your exception handling code here -----
            Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
      }
}

//------------------------------------------------------------------------------
//Callback Name: dialogShown_cb
//This callback is executed just before the dialog launch. Thus any value set
```

```
//here will take precedence and dialog will be launched showing that value.
//------------------------------------------------------------------------------
void Form_H::dialogShown_cb()
{
      try
      {
            //---- Enter your callback code here -----
      }
      catch (exception& ex)
      {
            //---- Enter your exception handling code here -----
            Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
      }
      }
//------------------------------------------------------------------------------
//Callback Name: apply_cb
//------------------------------------------------------------------------------
int Form_H::apply_cb()
{
      int errorCode = 0;
      try
      {
            //---- Enter your callback code here -----
      }
      catch (exception& ex)
      {
            //---- Enter your exception handling code here -----
            errorCode = 1;
            Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
      }
      return errorCode;
}
//------------------------------------------------------------------------------
//Callback Name: update_cb
//------------------------------------------------------------------------------
                /* Определяется массив линий с координатами их
                концов, используемых в эскизе выдавливаемого сечения */
UF_CURVE_line_t line[12];
tag_t objarray[12]; //теги для трех линий и дуги
void DrawElem(int i, double y, double z, double y1, double z1)
{
      //Так как сечение будет располагаться в плоскости ZY
      // координата X для всех объектов будет равна 0
      double x = 0;
      line[i].start_point[0] = x;// X1
      line[i].start_point[1] = y;// Y1
      line[i].start_point[2] = z;// Z1
      line[i].end_point[0] = x;// X2
      line[i].end_point[1] = y1;// Y2
```

```
	line[i].end_point[2] = z1;// Z2
	// построение линии
	UF_CURVE_create_line(&line[i], &objarray[i]);
}
int Form_H::update_cb(NXOpen::BlockStyler::UIBlock* block)
{
	try
	{
		if (block == drawingArea0)
		{
			//---------Enter your code here-----------
		}
		else if (block == Ra)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rb)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rc)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rd)
		{
			//---------Enter your code here-----------
		}
		else if (block == Re)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rf)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rg)
		{
			//---------Enter your code here-----------
		}
		else if (block == Rh)
		{
			//---------Enter your code here-----------
		}
		else if (block == but_create)
		{
			//---------Enter your code here-----------
			//параметры операции выдавливания
		   //расстояние начала и расстояние окончания выдавливания
			char a[] = "0.0";
			char b[125];
```

```
                sprintf(b, "%f", Rf->Value());

                char *limit[2] = { a, b };
                //орты вектора направления выдавливания
                double direction[3] = { 1.0, 0.0, 0.0 };
                //признак создания самостоятельного тела
                UF_FEATURE_SIGN create = UF_NULLSIGN;
                //заготовки указателей на перечии объектов
                uf_list_p_t loop_list, //Список объектов, подлежащих выдавли-
ванию (элементы сечения)
                     features; //Список созданных идентификаторов объектов
                //угол конусности в градусах
                char t_a[] = "0.0";
                char *taper_angle = t_a;
                //параметр не используется
                double ref_pt[3];
                double y, z, y1, z1;

                if (!UF_initialize())
                {
                     //левая вертикаль
                     y = 0; z = 0; y1 = 0;
                     z1 = atof(Ra->Value().getText());//
                     DrawElem(0, y, z, y1, z1);
                     //верхняя левая горизонталь
                     y = 0; z = z1;
                     y1 = (Rg->Value() - atof(Rb->Value().getText())) / 2;
                     DrawElem(1, y, z, y1, z1);
                     //вниз левая короткая
                y = y1; z = z1; z1 = atof(Ra->Value().getText())-Rc->Value();
                     DrawElem(2, y, z, y1, z1);
                     //верхняя короткая
                     y = y1; z = z1; y1 = y1+atof(Rb->Value().getText());
                     DrawElem(3, y, z, y1, z1);
                     //вверх правая короткая
                     y = y1; z = z1; z1 = atof(Ra->Value().getText());
                     DrawElem(4, y, z, y1, z1);
                     //верхняя правая горизонталь
                     y = y1; z = z1; y1= Rg->Value();
                     DrawElem(5, y, z, y1, z1);
                     //правая вертикаль вниз
                     y = y1; z = z1; z1 = 0;
                     DrawElem(6, y, z, y1, z1);
                     //правая нижняя короткая
     y = y1; z = z1; y1 = Rg->Value() - (Rg->Value() - Rh->Value()) / 2;
                     DrawElem(7, y, z, y1, z1);
                     //нижняя наклонная правая
                     y = y1; z = z1; y1 = Rg->Value() - (Rg->Value() - Rd-
>Value()) / 2; z1 = Re->Value();
                     DrawElem(8, y, z, y1, z1);
                     //нижняя средняя
```

```
                    y = y1; z = z1; y1 = (Rg->Value() - Rd->Value()) / 2;
                    DrawElem(9, y, z, y1, z1);
                    //нижняя наклонная левая
               y = y1; z = z1; y1 = (Rg->Value() - Rh->Value()) / 2; z1 =0;
                    DrawElem(10, y, z, y1, z1);
                    //левая нижняя короткая
                    y = y1; z = z1; y1 = 0;
                    DrawElem(11, y, z, y1, z1);
                    //создание пустого перечня объектов
                    UF_MODL_create_list(&loop_list);
                    //заполнение перечня тремя линиями и одной дугой
                    for (int i = 0; i < 12; i++)
                         UF_MODL_put_list_item(loop_list, objarray[i]);
                    //создание операции выдавливания
                    UF_MODL_create_extruded(loop_list, taper_angle, limit,
                         ref_pt, direction, create, &features);
                    UF_terminate();
               }

          }
     }
     catch (exception& ex)
     {
          //---- Enter your exception handling code here -----
          Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
     }
     return 0;
     }
//------------------------------------------------------------------------------
//Callback Name: ok_cb
//------------------------------------------------------------------------------
int Form_H::ok_cb()
{
     int errorCode = 0;
     try
     {
          errorCode = apply_cb();
     }
     catch (exception& ex)
     {
          //---- Enter your exception handling code here -----
          errorCode = 1;
          Form_H::theUI->NXMessageBox()->Show("Block Styler",
NXOpen::NXMessageBox::DialogTypeError, ex.what());
     }
     return errorCode;
}
//------------------------------------------------------------------------------
//StringBlock specific callbacks
//------------------------------------------------------------------------------
```

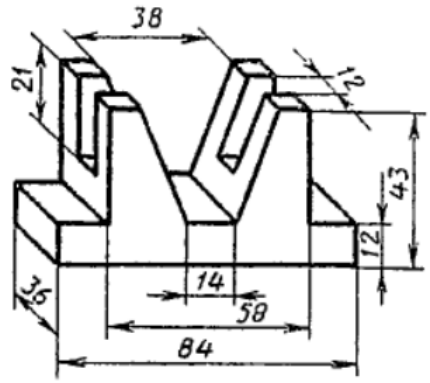
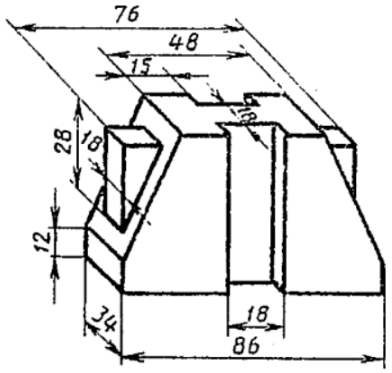
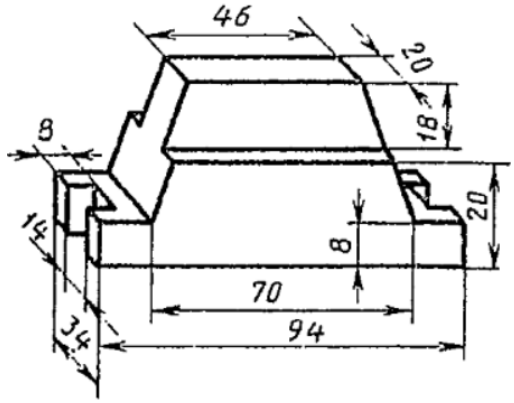
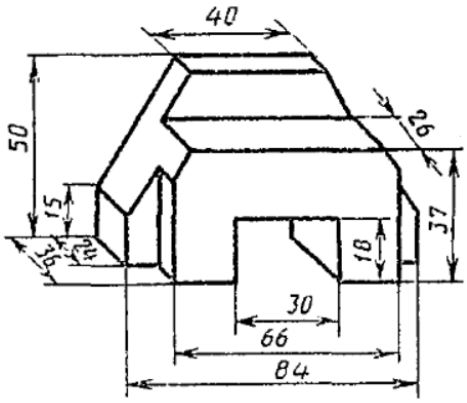
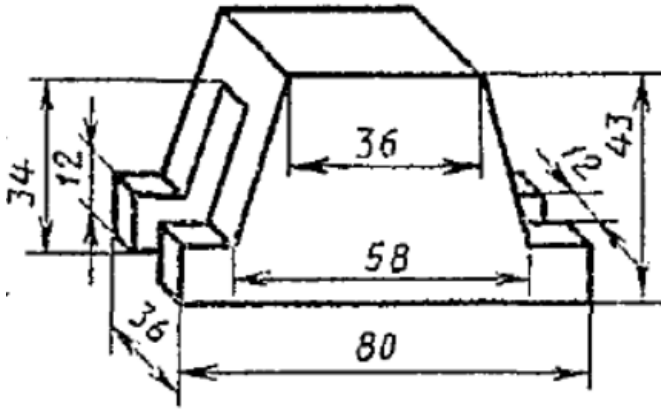
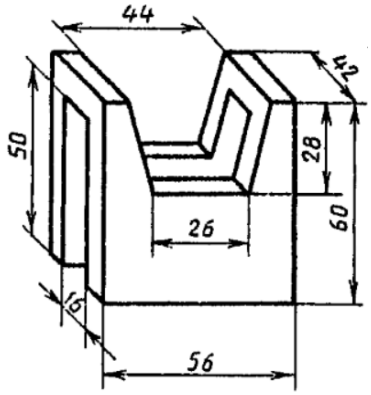
```
//int Form_H::KeystrokeCallback (NXOpen::BlockStyler::StringBlock*
string_block, NXString uncommitted_value)
//{
//}
//-------------------------------------------------------------------------
//Function Name: GetBlockProperties
//Description: Returns the propertylist of the specified BlockID
//-------------------------------------------------------------------------
PropertyList* Form_H::GetBlockProperties(const char *blockID)
{
      return theDialog->GetBlockProperties(blockID);
      }
```

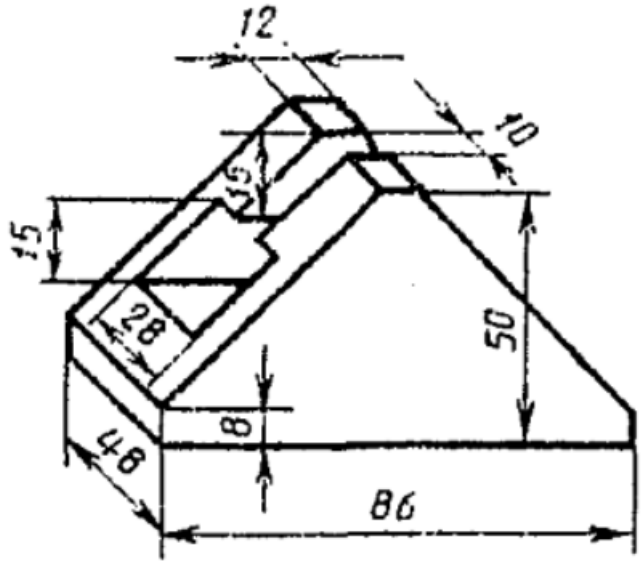
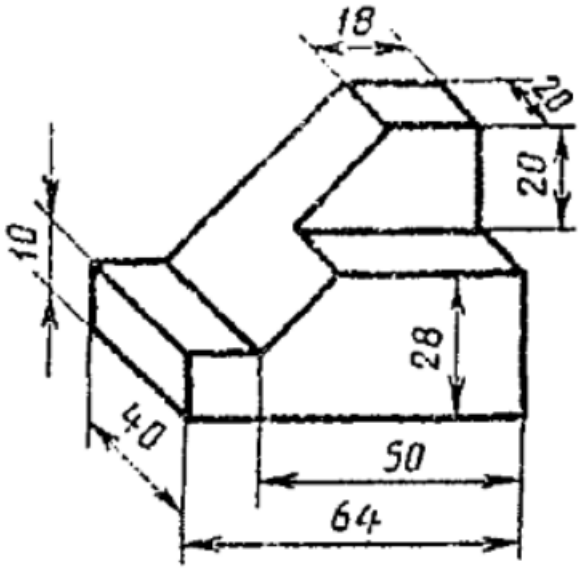
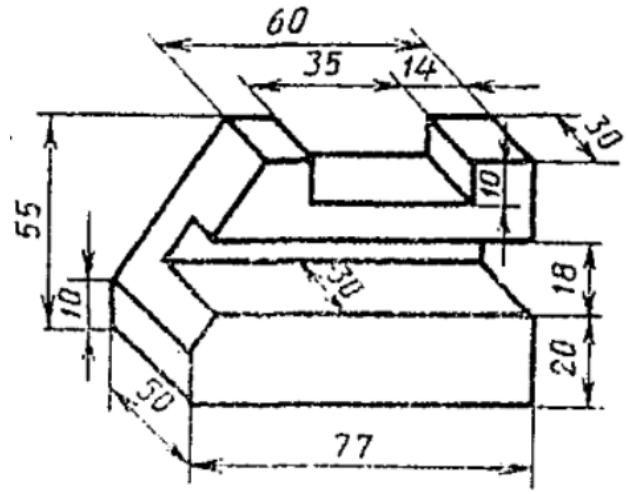
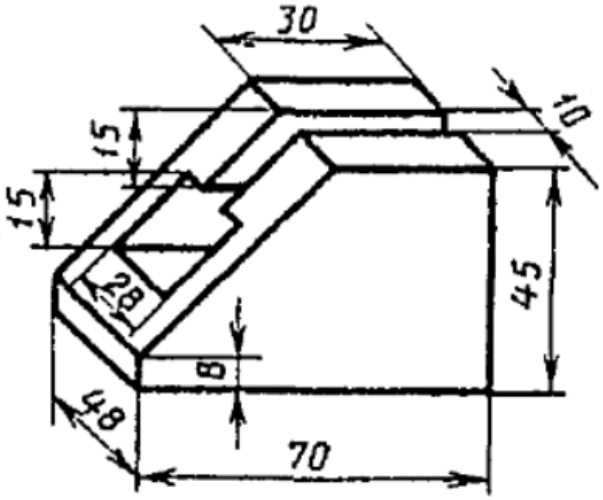
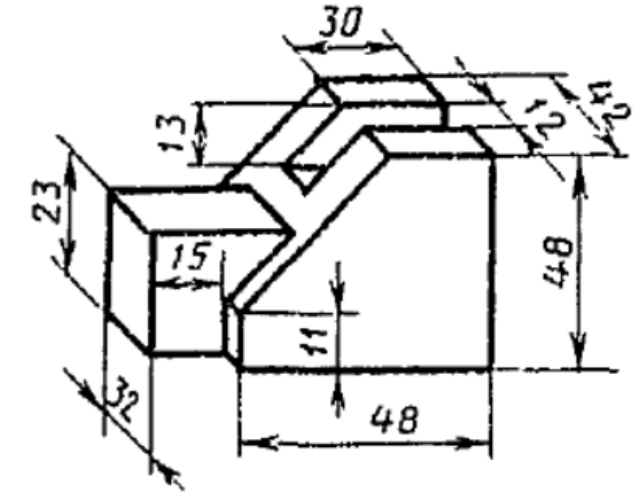
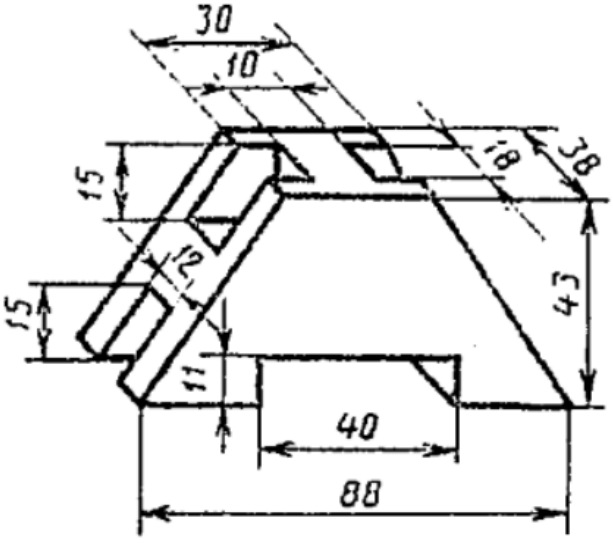
Приложение В
(обязательное)

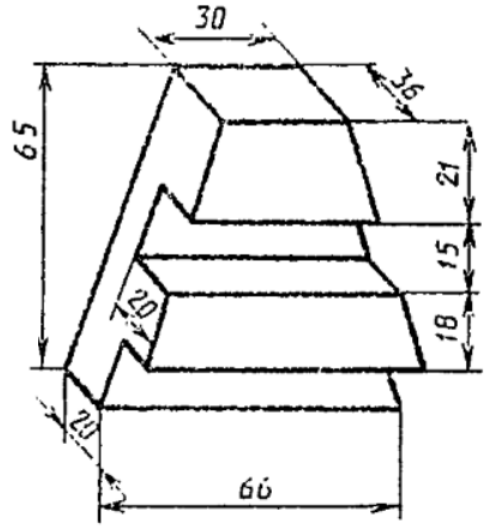
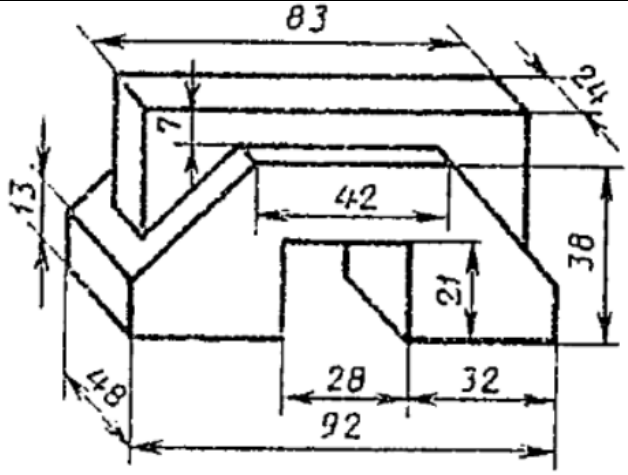
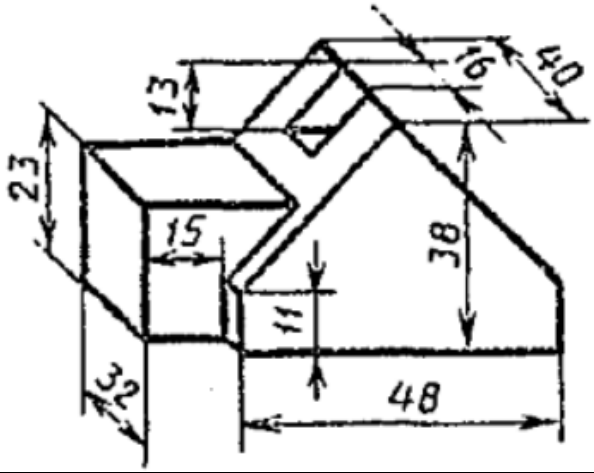
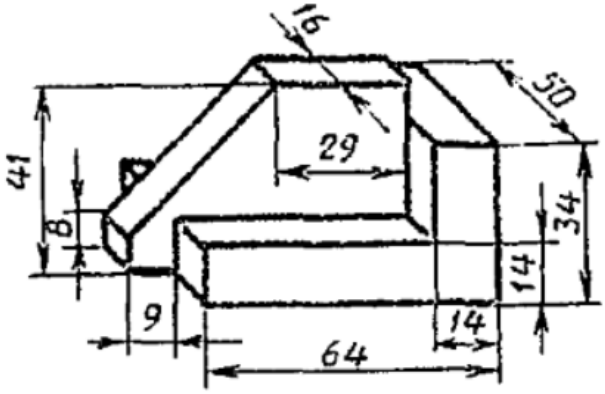
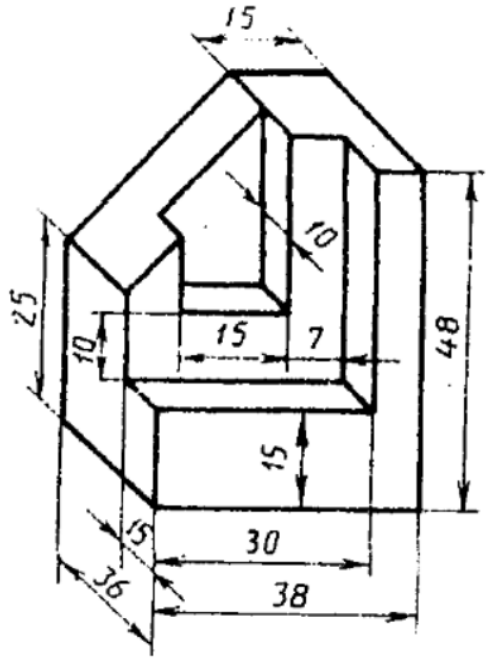
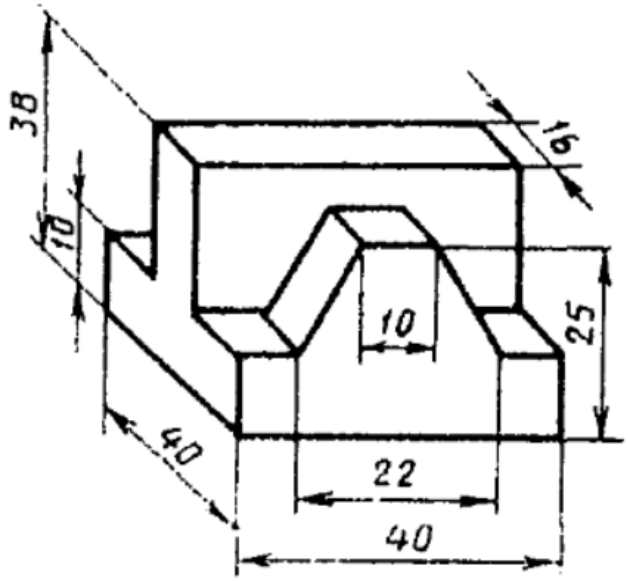
Варианты заданий для разработки программы построения трехмерной модели

Варианты заданий приведены из учебного пособия [6]. Штриховые линии строить не требуется.

Таблица В.1 – Варианты заданий для разработки программы построения трехмерной модели

Вариант 1	Вариант 2
	
Вариант 3	**Вариант 4**
	
Вариант 5	**Вариант 6**
	

Вариант 7	Вариант 8
	
Вариант 9	Вариант 10
	
Вариант 11	Вариант 12
	

Вариант 13	Вариант 14
	
Вариант 15	Вариант 16
	
Вариант 17	Вариант 18
	

Вариант 19	Вариант 20