

PRADIS

РАЗРАБОТКА ПГО НА C++

**ПРОГРАММНЫЙ КОМПЛЕКС ДЛЯ АВТОМАТИЗАЦИИ
МОДЕЛИРОВАНИЯ НЕСТАЦИОНАРНЫХ ПРОЦЕССОВ В
МЕХАНИЧЕСКИХ СИСТЕМАХ И СИСТЕМАХ ИНОЙ
ФИЗИЧЕСКОЙ ПРИРОДЫ**

ВЕРСИЯ 4.3

1. Содержание

1. Содержание.....	2
2. Введение.....	3
3. Объектная модель ПГО.....	4
3.1 Методы классов ПГО.....	6
3.2Порядок вызова методов.....	8
4. Принципы работы.....	9
5. Список используемых команд OpenCASCADE.....	10
5.1Команды создания топологии.....	10
5.2Команды преобразования в пространстве.....	12
5.3Создание объекта AIS_Shape.....	12
5.4Методы SolverContext.....	14
6. Системное окружение для разработки ПГО на C++.....	16
7. Процедура добавления плагин ПГО на C++ в PRADIS.....	18
8. Процесс создания новой ПГО на C++.....	20

2. Введение

Данный документ является вводным документом для пользователей комплекса Прадис, решивших создавать собственные программы графических образов (ПГО) на языке С++ для данного комплекса. Кроме языка С++, комплекс Прадис позволяет создавать ПГО на языке Фортран. Методы создания ПГО на Фортране изложены в документе «Разработка ПГО на Фортране». Выбор для разработки языка С++ дает разработчику неоспоримые преимущества по сравнению с разработкой на Фортране, так как предоставляет доступ ко всему богатому выбору функций графического пакета OpenCASCADE, но в то же время налагает гораздо более высокие требования к квалификации программиста и требует наличия гораздо большего количества знаний, особенно по работе с пакетом OpenCASCADE.

3. Объектная модель ПГО

Все разрабатываемые на C++ ПГО разрабатываются в виде отдельных классов и должны наследовать класс LVPS_GraphicModel и имплементировать имеющиеся там виртуальные методы:

<Конструктор>();

~<Деструктор>();

**virtual int Init(const std::vector<LVPS_Node>& nodeList,
const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,
Handle(AIS_InteractiveContext)& ais);** - инициализация параметров и номеров узлов.

virtual void Calculate(LVPS_XYZNode* nodes); - расчет по новому слою времени, геометрия, трансформация.

virtual void Display(); - отрисовка ПГО.

virtual void Refresh(); - обновление ПГО. Применение трансформации, перерисовка геометрии.

virtual void Reset(); - сброс ПГО. Удаление объекта из контекста.

virtual LVPS_GraphicModel* Clone(); - создание такого же объекта ПГО.

virtual inline QString GetType() const; - тип ПГО, т.е. имя ПГО в Прадис.

virtual inline QString GetModelClass() const; - класс ПГО, Mechanical – Механика.

virtual void SetVisibleLSK(bool); - установка видимости локальных осей объекта.

Так как в состав поставляемой с Прадис объектной библиотеки LVPS.lib входит большой набор вспомогательных классов, которые сами реализуют все эти методы кроме метода Init, то новые ПГО могут наследовать эти вспомогательные классы, и тогда им остается реализовать только метод Init. В состав библиотеки входят следующие вспомогательные классы:

LVPS_GM1Displacement

LVPS_GM1DisplacementExternalGeometry

LVPS_GM1Point

LVPS_GM1Point3Rotation

LVPS_GM1PointExternalGeometry

LVPS_GM1Quaternion

LVPS_GM1Rotation

LVPS_GM1RotationExternalGeometry

LVPS_GM2Displacement

LVPS_GM2Displacement1Rotation

LVPS_GM2Displacement1RotationExternalGeometry

LVPS_GM2DisplacementExternalGeometry

LVPS_GM2Point
LVPS_GM2PointExternalGeometry
LVPS_GM2PointSpringExternalGeometry
LVPS_GM2Quaternion

Эти классы реализуют различные стандартные графические преобразования а так же загрузку из графических файлов нужной геометрии. В общем случае ПГО может реализовывать сама все то, что делают вышеперечисленные классы, но лучше пользоваться ими, если задача позволяет это сделать.

3.1 Методы классов ПГО

Как было сказано выше, ПГО в общем случае должна реализовать следующие методы:

```
virtual int Init( const std::vector<LVPS_Node>& nodeList,  
const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,  
Handle(AIS_InteractiveContext)& ais);
```

Первым всегда выполняется метод Init, поэтому в нем должны выполняться все подготовительные действия.

```
virtual void Display();
```

Следующим выполняется метод Display. В нем графический объект отображается во вьювере.

```
virtual void Calculate(LVPS_XYZNode* nodes);
```

Далее, в процессе анимации используется метод Calculate. В этом методе происходит перерасчет пространственного положения и формы амортизатора, и объект перерисовывается методом OpenCASCADE - Redisplay.

```
virtual void Refresh();
```

Метод Refresh просто меняет пространственное положение объекта на основе пересчитанной трансформации.

```
virtual void Reset();
```

Метод Reset удаляет графический объект из вьювера.

```
virtual LVPS_GraphicModel* Clone();
```

Метод Clone создает копию данной ПГО.

```
virtual inline QString GetType() const;
```

Метод GetType возвращает тип ПГО, т.е. имя ПГО в Прадис.

```
virtual inline QString GetModelClass() const;
```

Метод GetModelClass возвращает класс ПГО, Mechanical – Механика.

```
virtual void SetVisibleLSK(bool);
```

Метод SetVisibleLSK устанавливает режим видимости локальных осей объекта.

Подробное описание и пример того, как реализуются эти методы можно прочитать в документе «Процесс создания новой ПГО на C++».

3.2 Порядок вызова методов

Вышеперечисленные методы вызываются в процессе работы Постпроцессора в следующем порядке:

- **Init**
- **Display**
- **Calculate**
- **Refresh**
- **Reset**

Остальные методы вызываются в произвольном порядке.

4. Принципы работы

ПГО подключаются к Постпроцессору как динамические библиотеки по технологии плагинов. Одна динамическая библиотека может содержать в себе несколько различных ПГО. Каждая ПГО должна иметь в библиотеке соответствующую функцию, возвращающую объект, созданный из класса данной ПГО.

Сведения о каждой ПГО содержатся в файле репозитория ПГО (PGO_List.txt). ПГО вызываются из Постпроцессора по мере необходимости и им передаются из Постпроцессора необходимые для создания графических объектов параметры.

ПГО создает нужный графический образ пользуясь средствами пакета OpenCASCADE, и передает этот графический образ в 3D выювер. Графический образ создается только один раз в методе Init, и отображается методом Display. А потом он только перемещается или трансформируется методом Calculate.

5. Список используемых команд OpenCASCADE

Ниже приведен список наиболее часто используемых при создании ПГО функций OpenCASCADE (OCC). Следует заметить, что возможности OCC по работе с трехмерной графикой весьма велики и данный список не может претендовать на полноту, но служит только, как пособие для начинающих пользоваться функциями OCC. Для дальнейшего изучения предлагается изучить документацию, прилагаемую в дистрибутиве OCC.

5.1 Команды создания топологии

Порядок создания трехмерного образа следующий:

- Сначала создается геометрия объекта с помощью геометрической библиотеки **gp**;
- Потом на основе геометрии создается топология объекта с помощью библиотек **BRepBuilderAPI** и **BRepPrimAPI**;
- Потом создается топологический объект **AIS_Shape**, который отрисовывается во выювере;
- Потом к этому объект **AIS_Shape** применяется трансформация **gp_Trsf**.

Для создания графических объектов используются следующие классы:

gp_Ax2 – создание оси с направлением;
gp_Pnt – создание трехмерной точки;
gp_Circ – создание круга;
gp_Lin - создание прямой линии;
gp_Dir – создание направления;
gp_Elips – создание эллипса;
gp_Hypr – создание гиперболы;
gp_Parab – создание параболы;
gp_Pln – создание плана;
gp_Vec – создание вектора;

В OCC имеется большой набор классов и методов создания топологии трехмерных объектов:

BRepBuilderAPI_MakeEdge – создание различных типов кривых;
BRepBuilderAPI_MakeWire – создание ломаных линий, состоящих из различных кривых;
BRepBuilderAPI_MakeFace – создание поверхностей;
BRepPrimAPI_MakeCone – создание конуса;
BRepPrimAPI_MakeSphere – создание сферы;
BRepPrimAPI_MakePrism – создание призмы;
BRepPrimAPI_MakeCylinder – создание цилиндра;
GeomFill_Pipe - вытягивание контура по нормали или по кривой;
TopoDS_Compound – объединение различных топологических объектов в одну деталь.

Все создаваемые топологические примитивы записываются в единую структуру данных описываемую классом **TopoDS_Shape**.

Примеры использования вышеперечисленных классов:

Создание поверхности в форме круга

```
gp_Pnt A(0, 0, 0);
gp_Pnt B(0, 0, 1);
double radius = 10;
gp_Vec vec(A, B);
gp_Dir dir(vec);
gp_Ax2 ax2( A, dir );
gp_Circ circ(ax2, radius);
BRepBuilderAPI_MakeEdge edge(circ);
TopoDS_Wire Wire = BRepBuilderAPI_MakeWire(edge.Edge());
TopoDS_Shape s = BRepBuilderAPI_MakeFace(Wire);
```

Создание шара:

```
gp_Pnt A(10, 0, 5);
double diameter = 5;
BRepPrimAPI_MakeSphere sf(A, diameter/2.);
TopoDS_Shape s = sf.Shape();
```

Создание цилиндра:

```
gp_Pnt A(0,0,0);
gp_Pnt B(0,0,10);
double diameter = 7;
double length=sqrt(pow(B.X()-A.X(),2)+pow(B.Y()-A.Y(),2)+pow(B.Z()-A.Z(),2));
gp_Vec vec(A, B);
gp_Dir dir(vec);
gp_Ax2 ax2( A, dir );
BRepPrimAPI_MakeCylinder cyl(ax2, diameter/2., length);
TopoDS_Shape s = cyl.Shape();
```

Создание прямой линии:

```
gp_Pnt A(10, 15, 20);
gp_Pnt B(50, 100, 130);
BRepBuilderAPI_MakeEdge edge(A, B);
TopoDS_Shape s = edge.Edge();
```

Создание дуги:

```
gp_Pnt A(0,0,0);
gp_Pnt B(0,0,10);
double Ang1 = 0;
double Ang2 = 0.5;
double diameter = 10;
gp_Vec vec(A, B);
gp_Dir dir(vec);
gp_Ax2 ax2( A, dir );
gp_Circ circ(ax2, diameter/2.);
BRepBuilderAPI_MakeEdge edge(circ, Ang1, Ang2);
TopoDS_Shape s = edge.Edge();
```

5.2 Команды преобразования в пространстве

Графические объекты, созданные командами, перечисленными в пункте первом могут быть перемещаемы в пространстве и трансформируемы. Для этого используется класс **gp_Trsf**. Этот класс имеет такие методы, как:

SetRotation – задание вращения;

SetTranslation – задание перемещения;

SetScaleFactor – задание масштабного коэффициента.

Эти методы позволяют производить над трехмерным графическим объектом различные перемещения и трансформации. Кроме того, возможно прямое задание матрицы трансформации.

Пример использования класса gp_Trsf:

```
gp_Trsf aTrsf;

gp_Vec Vector(gp_Pnt(0,0,0),gp_Pnt(0,0,5));
gp_Vec v2(B.X()-A.X(),B.Y()-A.Y(),B.Z()-A.Z());
gp_Pnt Point(0,0,0);

double phi = acos(Vector*v2/Vector.Magnitude()/v2.Magnitude ());
if(fabs(phi)>=1e-8 && !Vector.IsParallel (v2,gp::Resolution()))
{
    gp_Vec vec = Vector^v2;
    gp_Ax1 ax1(Point, vec);
    aTrsf.SetRotation(ax1, phi);
}
gp_Trsf trsf;

gp_Vec n1(A.X()-Point.X (),A.Y()-Point.Y (),A.Z()-Point.Z ());
trsf.SetTranslation(n1);
aTrsf=trsf*aTrsf;
aTrsf.SetScaleFactor(5.);
```

5.3 Создание объекта AIS_Shape

Для отображения графических объектов в 3D вьювере используется специальный объект **AIS_Shape**, который используется совместно с другим классом - **AIS_InteractiveContext**, являющимся механизмом отображения графических объектов. Для каждого законченного графического объекта создается отдельный объект **AIS_Shape**, который затем заносится в объект **AIS_InteractiveContext**, который существует один для всех графических объектов и является средой их существования.

Класс **AIS_Shape** создается на основе объекта **TopoDS_Shape** и имеет следующие полезные методы:

SetMaterial – задание материала;

SetColor – задание цвета;

SetTransparency – задание прозрачности;

Redisplay – перерисовка объекта;

SetDisplayMode – задание режима рисования;

Класс **AIS_InteractiveContext** создается один на все приложение и в ПГО он создаваться и изменяться не должен и приходит в виде входного параметра. Класс **AIS_InteractiveContext** имеет следующие полезные методы:

SetDisplayMode – задание режима рисования графического объекта;

Display – отображение графического объекта;

Redisplay – перерисовка уже отображенного графического объекта;

ResetLocation – изменение положения объекта;

SetLocation – перемещение графического объекта с использованием объекта gp_Trsf;

Remove – удаление графического объекта;

SetMaterial – задание материала;

SetColor – задание цвета;

SetTransparency – задание прозрачности;

Пример совместного использования классов AIS_Shape и AIS_InteractiveContext:

Создание объекта AIS_Shape и задание режима его отображения.

```
Handle(AIS_Shape) myAISShape = new AIS_Shape (Shape);  
myAISShape->SetMaterial(Graphic3d_NOM_PLASTIC);  
myAISContext->SetDisplayMode(myAISShape, 1, Standard_False);
```

.....

Отображение объекта AIS_Shape и задание его положения в пространстве.

```
myAISContext->Display(myAISShape,1,1,Standard_False,Standard_False);  
myAISContext->SetLocation(myAISShape,aTrsf);
```

.....

Перерисовка объекта AIS_Shape на основе новой топологии.

```
Handle(AIS_Shape)::DownCast(myAISShape)->Set(Shape);  
myAISContext->Redisplay(myAISShape,Standard_False);
```

.....

Изменение пространственного положения объекта AIS_Shape.

```
myAISContext->ResetLocation(myAISShape);  
myAISContext->SetLocation(myAISShape,aTrsf);
```

.....

Удаление объекта AIS_Shape.

```
myAISContext->Remove(myAISShape);
```

5.4 Методы SolverContext

Для доступа к входным параметрам ПГО используются методы объекта **SolverContext** и некоторые другие классы, которые работают совместно с объектом класса **SolverContext**. Объект **SolverContext** создается один на все приложение и в ПГО он создаваться и изменяться не должен и приходит в виде входного параметра. Подробное описание работы с **SolverContext** см. В документе «Работа с DAT файлом».

Все статические входные параметры ПГО берутся из **SolverContext** и приходят в ПГО как параметр в методе **Init** в виде списка с ключами **Q3Dict<QString>& parameterList**. Каждый вид параметров имеет соответствующее ключевое имя к которому прибавляется его порядковый номер. Существуют следующие ключевые имена:

Par – параметры модели, к которой относится данная ПГО;

PARIMD – параметры ПГО;

PARLR2 – параметры слоя;

Пример получения входных параметров ПГО:

```
double par[9];
for(int a=0;a<6;a++)
{
    QString param=QString().sprintf("Par%d",a);
    QString Dr=*(parameterList[param]);
    par[a]=LVPS_Utility::ToDouble(Dr);
}

for(int a=0;a<2;a++)
{
    QString param=QString().sprintf("PARIMD%d",a);
    QString Dr=*(parameterList[param]);
    par[a + 6]=LVPS_Utility::ToDouble(Dr);
}
```

Значения на узлах приходят как входной параметр метода **Calculate** - **LVPS_XYZNode* nodes**, а описания самих узлов приходят как параметр метода **Init** - **std::vector<LVPS_Node>& nodeList**. Этот вектор содержит объекты класса **LVPS_Node**, который содержит ID узла, который используется в данной ПГО. По этому ID можно получить значение конкретного узла из вектора **nodes** в методе **Calculate**.

Пример получения значений узлов:

```
int LVPS_AMORT::Init(const std::vector<LVPS_Node>& nodeList,
    const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,
    Handle(AIS_InteractiveContext)& ais)
{
    LVPS_Node NodeX1= nodeList[0];
    LVPS_Node NodeY1= nodeList[1];
}
```

```

        LVPS_Node NodeZ1= nodeList[2];
        LVPS_Node NodeX2= nodeList[3];
        LVPS_Node NodeY2= nodeList[4];
        LVPS_Node NodeZ2= nodeList[5];
        .....
void LVPS_AMORT::Calculate(LVPS_XYZNode* nodes)
{
    double leng=sqrt(pow(nodes[NodeX2.ID].S+B.X()-nodes[NodeX1.ID].S-
    A.X(),2)+pow(nodes[NodeY2.ID].S+B.Y()-nodes[NodeY1.ID].S-
    A.Y(),2)+pow(nodes[NodeZ2.ID].S+B.Z()-nodes[NodeZ1.ID].S-A.Z(),2));
    .....

```

Все дополнительные параметры берутся из соответствующих контекстов, работающих на основе объекта **SolverContext**.

Пример получения значений рабочего массива из ModelContext:

Для получения значений массивов для конкретной модели надо вначале установить ModelContext на нужную модель по ее номеру используя метод SetModelNumber.

```

ModelContext MC(SolverCont);
MC.SetModelNumber(2);
double W14 = MC.GetWRK()[14];
double W15 = MC.GetWRK()[15];
double W8 = MC.GetWRK()[8];

```

Пример получения значений старого и нового вектора состояния ModelContext:

```

ModelContext MC(SolverCont);
MC.SetModelNumber(2);
double NEW1 = MC.GetNew()[1];
double OLD3 = MC.GetOld()[3];

```

6. Системное окружение для разработки ПГО на C++

Для разработки ПГО на C++ необходимо, что бы в системе были установлены следующие программные средства:

OpenCASCADE 5.2 или выше.

Qt 4

PRADIS Postprocessor

MS Visual Studio 6.0

Так же в проект разрабатываемой ПГО должны быть включены библиотеки **AReader.lib** и **LVPS.lib**, входящие в комплект PRADIS. Кроме того, должны использоваться следующие include файлы, так же входящие в комплект PRADIS:

- Для работы с библиотекой AReader надо включить директорию AReader;
- Для работы с библиотекой LVPS нужно включить директорию LVPS.

Проект разрабатываемой DLL библиотеки должен включать в себя следующие настройки:

Для компиляции: /nologo /MTd /W3 /Gm /GX /ZI /Od /I "c:\dinama\include\LVPS" /I "c:\dinama\include\AReader" /I "\$(QTDIR)\include" /I "\$(QTDIR)\include\QtGui" /I "\$(QTDIR)\include\Qt3Support" /I "Qwt/include" /I "\$(QTDIR)\include\QtXml" /I "\$(QTDIR)\include\QtOpenGL" /I "\$(QTDIR)\include\QtCore" /I "\$(QTDIR)\include" /I "\$(QTDIR)\include\ActiveQt" /I "tmp\moc\release_shared" /I "." /I "\$(QTDIR)\mkspecs\win32-msvc" /I "\$(CASROOT)\inc" /D "_DEBUG" /D "WIN32" /D "_WINDOWS" /D "_MBCS" /D "WNT" /D "CSFDB" /D "QT_DLL" /D "QT3_SUPPORT" /FR"tmp\obj\release_shared" /Fp"win32\obj\IESample.pch" /YX /Fo"win32\objd/" /Fd"win32\objd/" /FD /GZ /c

Для сборки: "c:\dinama\lib\LVPS.lib" "c:\dinama\lib\AReader.lib" "\$(QTDIR)\lib\qtmain.lib" "\$(QTDIR)\lib\QtCore4.lib" "\$(QTDIR)\lib\Qt3Support4.lib" "\$(QTDIR)\lib\QtOpenGL4.lib" "\$(QTDIR)\lib\QtXml4.lib" "\$(QTDIR)\lib\Qt3Supportd4.lib" "\$(QTDIR)\lib\QtGui4.lib" kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbccp32.lib qtmain.lib TKernel.lib TKMath.lib TKService.lib TKV3d.lib TKBrep.lib TKIGES.lib PTKernel.lib TKSTL.lib TKVRML.lib TKSTEP.lib TKShapeSchema.lib TKG3d.lib TKG2d.lib TKXSBase.lib TKPShape.lib TKShHealing.lib TKTopAlgo.lib TKBool.lib TKBO.lib TKFillet.lib TKOffset.lib TKPrim.lib TKGeomBase.lib TKGeomAlgo.lib TKMeshVS.lib TKFeat.lib TKCAF.lib /nologo /dll /incremental:yes /pdb:"Debug\PGO_NAME.pdb" /debug /machine:I386 /out:"bin\PGO_NAME.dll" /implib:"Debug\PGO_NAME.lib" /pdbtype:sept /libpath:"\$(QTDIR)\lib" /libpath:"\$(CASROOT)\win32\lib"

Название ПГО **PGO_NAME** должно быть соответственно изменено на нужное.

В пути **c:\dinama** должен быть соответственно проставлен диск, на котором установлен Прадис.

Проект должен иметь доступ к следующим библиотекам PRADIS:

LVPS.lib

AReader.lib

Проект должен иметь доступ к следующим библиотекам Qt:

LVPS.lib
AReader.lib
qtmain.lib
QtCore4.lib
Qt3Support4.lib
Qt3Supportd4.lib
QtGui4.lib
qtmain.lib

Проект должен иметь доступ к следующим библиотекам OpenCASCADE:

TKernel.lib
TKMath.lib
TKService.lib
TKV3d.lib
TKBrep.lib
TKIGES.lib
PTKernel.lib
TKSTL.lib
TKVRML.lib
TKSTEP.lib
TKShapeSchema.lib
TKG3d.lib
TKG2d.lib
TKXSBase.lib
TKPShape.lib
TKShHealing.lib
TKTopAlgo.lib
TKBool.lib
TKBO.lib
TKFillet.lib
TKOffset.lib
TKPrim.lib
TKGeomBase.lib
TKGeomAlgo.lib
TKMeshVS.lib
TKFeat.lib
TKCAF.lib

7. Процедура добавления плагин ПГО на С++ в PRADIS

Для добавления новой ПГО в PRADIS необходимо произвести следующие действия:

1. Создать DLL библиотеку, содержащую новую ПГО.
2. Поместить созданную DLL библиотеку в каталог **%DINSYS%\dinama\post\plugins\gip**.
3. Поместить информацию о ПГО в файл репозитория ПГО (**PGO_List.txt**). Подробно эта процедура описана в документе «Разработка ПГО на Фортране».
4. Создать текст пустой ПГО, содержащий только заголовок и описание ПГО согласно требованиям, предъявляемым к этому системой Прадис, и не содержащей никаких вычислений. Формат описания ПГО на Фортране содержится в документах «Динамическое встраивание в решатель PRADIS библиотек моделей элементов, ПГО, ПРВП» и «ВКЛЮЧЕНИЕ ПРОГРАММ РЕАЛИЗАЦИИ ГРАФИЧЕСКИХ ОБРАЗОВ В БИБЛИОТЕКИ КОМПЛЕКСА».

Например, текст пустой ПГО AMORT на Фортране выглядел бы следующим образом:

```
C IMAGE AMORT:EXT=6, PAR=3, WRK=1
C
C HELP Графический образ амортизатора.
C HELP НАЗВАНИЕ:      Графический образ амортизатора.
C
C
C HELP СТЕПЕНИ СВОБОДЫ:
C HELP  1 - поступательная точки A в направлении оси OX;
C HELP  2 - поступательная точки A в направлении оси OY;
C HELP  3 - поступательная точки A в направлении оси OZ;
C HELP  4 - поступательная точки B в направлении оси OX;
C HELP  5 - поступательная точки B в направлении оси OY;
C HELP  6 - поступательная точки B в направлении оси OZ.
C
C HELP ПАРАМЕТРЫ:
C HELP  1 - диаметр амортизатора;
C HELP  2 - отношение хода сжатия к начальной длине амортизатора;
C HELP  3 - отношение хода растяжения к начальной длине амортизатора.
C
C
      include 'init.inc'

      SUBROUTINE AMORT (
,          NAMEX,
,          I,
,          X_, V_, A_,
,          INNER, EXT,
,          PARX, WRKX,
```

```
,      PAR, WRK,  
,      PARLR2 )
```

!DEC\$ ATTRIBUTES DLLEXPORT:: AMORT

```
include 'common.inc'
```

```
C      Формальные параметры  
      CHARACTER*8 NAMEX  
      REAL * 8 I (1)  
      REAL * 8      X_(6), V_(6), A_(6)  
      REAL * 8 INNER(1),PARX(1),WRKX(1),PAR(1),WRK(1),PARLR2(1)  
      INTEGER * 4 EXTC  
  
C      RETURN  
      END
```

5. Добавить эту ПГО в решатель с помощью утилиты ARM, дав команду:

ARM + <имя ПГО>
(Например: **ARM + AMORT**)

Работа с утилитой ARM описана в документе «Динамическое встраивание в решатель PRADIS библиотек моделей элементов, ПГО, ПРВП».

6. После выполнения всех вышеперечисленных процедур, новая ПГО готова к использованию.

8. Процесс создания новой ПГО на C++

Предполагается, что читатель этого документа знаком с основами программирования в среде OpenCASCADE. Для более подробного ознакомления с OpenCASCADE следует изучить документацию, прилагаемую к OpenCASCADE.

Для того, что бы создать новую ПГО на C++, нужно прежде всего организовать соответствующее окружение.

После этого надо создать в VC 6.0 пустой проект динамической библиотеки, подключить к нему все нужные библиотеки и include файлы, включить в проект файлы name.cpp и name.h содержащие текст ПГО, как это показано ниже.

Одна DLL библиотека может содержать как одну, так и несколько ПГО. Каждая ПГО описывается как отдельный класс и для каждой ПГО в DLL библиотеке должна быть предусмотрена функция ее вызова, как это показано в ниже приведенном примере для ПГО AMORT:

```
#ifdef Q_WS_WIN
#define MY_EXPORT __declspec(dllexport)
#else
#define MY_EXPORT
#endif

extern "C" MY_EXPORT LVPS_GraphicModel* AMORT()
{
    LVPS_GraphicModel* model = new LVPS_AMORT();
    return model;
}
```

Далее мы рассмотрим пример разработки конкретной реально используемой ПГО с названием AMORT. Сначала приведем полные тексты ее кода, содержащегося в двух файлах LVPS_AMORT.cpp и LVPS_AMORT.h:

LVPS_AMORT.cpp

```
#include <LVPS_AMORT.h>
#include <LVPS_XYZNode.hxx>
#include <LVPS_Utility.hxx>
#include <Geom_TrimmedCurve.hxx>
#include <TopoDS_Edge.hxx>
#include <BRepBuilderAPI_MakeWire.hxx>
#include <GC_MakeArcOfCircle.hxx>
#include <BRepBuilderAPI_MakeEdge.hxx>
#include <TopoDS_Wire.hxx>
#include <gp_Circ.hxx>
#include <gp_Pln.hxx>
#include <TopoDS_Face.hxx>
#include <BRepBuilderAPI_MakeFace.hxx>
#include <BRepOffsetAPI_MakePipe.hxx>
```

```

#include <Geom_CartesianPoint.hxx>
#include <AIS_Point.hxx>
#include <BRepPrimAPI_MakeCylinder.hxx>
#include <TopoDS_Compound.hxx>

#ifdef Q_WS_WIN
#define MY_EXPORT __declspec(dllexport)
#else
#define MY_EXPORT
#endif

extern "C" MY_EXPORT LVPS_GraphicModel* AMORT()
{
    LVPS_GraphicModel* model = new LVPS_AMORT();
    return model;
}

LVPS_AMORT::LVPS_AMORT()
{
    myAISShape.Nullify();
};

LVPS_AMORT::~LVPS_AMORT()
{
};

int LVPS_AMORT::Init(const std::vector<LVPS_Node>& nodeList,
    const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,
    Handle(AIS_InteractiveContext)& ais)
{
    NodeX1= nodeList[0];
    NodeY1= nodeList[1];
    NodeZ1= nodeList[2];
    NodeX2= nodeList[3];
    NodeY2= nodeList[4];
    NodeZ2= nodeList[5];
    myAISContext = ais;

    double par[9];
    for(int a=0;a<6;a++)
    {
        QString param=QString().sprintf("Par%d",a);
        QString Dr=*(parameterList[param]);
        par[a]=LVPS_Utility::ToDouble(Dr);
    }

    for(int a=0;a<2;a++)
    {
        QString param=QString().sprintf("PARIMD%d",a);
        QString Dr=*(parameterList[param]);
    }
}

```

```

        par[a + 6]=LVPS_Utility::ToDouble(Dr);
    }

    A=gp_Pnt(par[0],par[1],par[2]);
    B=gp_Pnt(par[3],par[4],par[5]);
    diameter=par[6];
    length=sqrt(pow(B.X()-A.X(),2)+pow(B.Y()-A.Y(),2)+pow(B.Z()-A.Z(),2));
    minlength = length * par[7];
    maxlength = length + length * par[8];

    Shape = Amort(diameter, length, minlength, maxlength);
    myAISShape = new AIS_Shape (Shape);
    myAISShape->SetMaterial(Graphic3d_NOM_PLASTIC);
    SetColor(myAISShape);

    myAISContext->SetDisplayMode(myAISShape, 1, Standard_False);

    return 0;
};

void LVPS_AMORT::Display()
{
    myAISContext->Display(myAISShape,1,1,Standard_False,Standard_False);
    myAISContext->SetLocation(myAISShape,aTrsf);
};

void LVPS_AMORT::Calculate(LVPS_XYZNode* nodes)
{
    double leng=sqrt(pow(nodes[NodeX2.ID].S+B.X()-nodes[NodeX1.ID].S-
A.X(),2)+pow(nodes[NodeY2.ID].S+B.Y()-nodes[NodeY1.ID].S-
A.Y(),2)+pow(nodes[NodeZ2.ID].S+B.Z()-nodes[NodeZ1.ID].S-A.Z(),2));
    Shape = Amort(diameter, leng, minlength, maxlength, nodes);

    if(myAISShape.IsNull())
    {
        myAISShape = new AIS_Shape (Shape);
    }else
    {
        Handle(AIS_Shape)::DownCast(myAISShape)->Set(Shape);
        myAISContext->Redisplay(myAISShape, Standard_False);
    }
};

void LVPS_AMORT::Refresh()
{
    myAISContext->ResetLocation(myAISShape);
    myAISContext->SetLocation(myAISShape,aTrsf);
};

void LVPS_AMORT::Reset()
{

```

```

        myAISContext->Remove(myAISShape);
};

LVPS_GraphicModel* LVPS_AMORT::Clone()
{
    return (LVPS_GraphicModel*)(new LVPS_AMORT());
};

TopoDS_Shape LVPS_AMORT::Amort(const Standard_Real diameter1 ,
                                const Standard_Real length, const Standard_Real minlength, const
Standard_Real maxlength, LVPS_XYZNode* nodes)
{
    TopoDS_Compound comp;
    BRep_Builder builder;
    builder.MakeCompound( comp );

    BRepPrimAPI_MakeCylinder cyl1(diameter1/2., minlength);
    builder.Add( comp, cyl1.Shape() );

    BRepPrimAPI_MakeCylinder cyl2(diameter1/10., length);

    builder.Add( comp, cyl2.Shape() );

    S = comp;

    if(nodes==NULL)
    {
        gp_Vec Vector(gp_Pnt(0,0,0),gp_Pnt(0,0,5));
        gp_Vec v2(B.X()-A.X(),B.Y()-A.Y(),B.Z()-A.Z());
        gp_Pnt Point(0,0,0);

        double phi = acos(Vector*v2/Vector.Magnitude()/v2.Magnitude ());
        if(fabs(phi)>=1e-8 && !Vector.IsParallel (v2,gp::Resolution()))
        {
            gp_Vec vec = Vector^v2;
            gp_Ax1 ax1(Point, vec);
            aTrsf.SetRotation(ax1, phi);
        }
        gp_Trsf trsf;

        gp_Vec n1(A.X()-Point.X (),A.Y()-Point.Y (),A.Z()-Point.Z ());
        trsf.SetTranslation(n1);
        aTrsf=trsf*aTrsf;
    }
    else
    {
        gp_Vec Vector(gp_Pnt(0,0,0),gp_Pnt(0,0,5));
        gp_Vec v2(nodes[NodeX2.ID].S+B.X()-nodes[NodeX1.ID].S-A.X(),
                    nodes[NodeY2.ID].S+B.Y()-nodes[NodeY1.ID].S-A.Y(),
                    nodes[NodeZ2.ID].S+B.Z()-nodes[NodeZ1.ID].S-A.Z());
        gp_Pnt Point(0,0,0);
    }
}

```

```

        double phi = acos(Vector*v2/Vector.Magnitude()/v2.Magnitude ());
        if(fabs(phi)>=1e-8 && !Vector.IsParallel (v2,gp::Resolution()))
        {
            gp_Vec vec = Vector^v2;
            gp_Ax1 ax1(Point, vec);
            aTrsf.SetRotation(ax1, phi);
        }
        gp_Trsf trsf;

        gp_Vec n1(nodes[NodeX1.ID].S+A.X()-Point.X (),
            nodes[NodeY1.ID].S+A.Y()-Point.Y
            (),nodes[NodeZ1.ID].S+A.Z()-Point.Z ());
        trsf.SetTranslation(n1);
        aTrsf=trsf*aTrsf;
    }

    return S;
}

```

LVPS_AMORT.h

```

#ifndef LVPS_AMORT_H
#define LVPS_AMORT_H

#include "LVPS_GraphicModel.hxx"
#include "LVPS_Node.hxx"
#include <TopoDS_Face.hxx>
#include <TopoDS_Wire.hxx>
#include "LVPS.h"

class LVPS_AMORT:public LVPS_GraphicModel
{
public:
    LVPS_AMORT();
    ~LVPS_AMORT();

    virtual inline QString GetType() const
    {
        return "AMORT";
    };
    virtual inline QString GetModelClass() const
    {
        return "Mechanical";
    };
    virtual int Init(
        const std::vector<LVPS_Node>& nodeList,
        const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,
        Handle(AIS_InteractiveContext)& ais);

```



```

virtual void Calculate(LVPS_XYZNode* nodes);
virtual void Display();
virtual void Refresh();
virtual void Reset();
virtual LVPS_GraphicModel* Clone();
virtual void SetVisibleLSK(bool){};
TopoDS_Shape Amort(const Standard_Real diameter1 ,
                    const Standard_Real length, const Standard_Real minlength,
                    const Standard_Real maxlength, LVPS_XYZNode* nodes=NULL);

protected:
    TopoDS_Shape Shape;
    Handle(AIS_Shape) myAISShape,myshape;
    Handle(AIS_InteractiveContext) myAISContext;
    LVPS_Node NodeX1, NodeY1, NodeZ1,NodeX2, NodeY2,NodeZ2;
    double diameter,quantity,length, maxlength, minlength;
    gp_Trsf aTrsf;
    gp_Pnt A,B;
    TopoDS_Face F;
    TopoDS_Wire W;
    TopoDS_Shape S;
};
#endif

```

Каждая ПГО должна содержать определенный набор реализованных обязательных методов:

```

virtual int Init( const std::vector<LVPS_Node>& nodeList,
                  const Q3Dict<QString>& parameterList, const LVPS_Animator* animator,
                  Handle(AIS_InteractiveContext)& ais);

virtual void Calculate(LVPS_XYZNode* nodes);
virtual void Display();
virtual void Refresh();
virtual void Reset();
virtual LVPS_GraphicModel* Clone();
virtual inline QString GetType() const;
virtual inline QString GetModelClass() const;

```

И должна наследоваться от класса LVPS_GraphicModel или от другого более общего класса, входящего в состав библиотеки LVPS и наследующего в свою очередь класс LVPS_GraphicModel. В таком случае сама ПГО должна реализовать только методы Init и Clone, так как остальные методы уже реализованы в наследуемых классах.

ПГО AMORT, которую мы рассматриваем в качестве примера, сама реализовывает все необходимые методы, так как она наследуется непосредственно от класса LVPS_GraphicModel.

Первым всегда выполняется метод Init, поэтому в нем должны выполняться все подготовительные действия. Вначале из входных параметров инициализируются переменные узлов:

```

NodeX1= nodeList[0];
NodeY1= nodeList[1];
NodeZ1= nodeList[2];
NodeX2= nodeList[3];
NodeY2= nodeList[4];
NodeZ2= nodeList[5];

```

Затем заполняется поле для AIS_InteractiveContext:

```

myAISContext = ais;

```

После этого запоминаются все необходимые входные параметры ПГО:

```

double par[9];
for(int a=0;a<6;a++)
{
    QString param=QString().sprintf("Par%d",a);
    QString Dr=*(parameterList[param]);
    par[a]=LVPS_Utility::ToDouble(Dr);
}

for(int a=0;a<2;a++)
{
    QString param=QString().sprintf("PARIMD%d",a);
    QString Dr=*(parameterList[param]);
    par[a + 6]=LVPS_Utility::ToDouble(Dr);
}

```

Далее, из значений входных параметров создаются две трехмерные точки и переменная, содержащая диаметр амортизатора:

```

A=gp_Pnt(par[0],par[1],par[2]);
B=gp_Pnt(par[3],par[4],par[5]);
diameter=par[6];

```

Вычисляется длина амортизатора и длины составляющих его частей:

```

length=sqrt(pow(B.X()-A.X(),2)+pow(B.Y()-A.Y(),2)+pow(B.Z()-A.Z(),2));
minlength = length * par[7];
maxlength = length + length * par[8];

```

Создается объект ToroDS_Shape содержащий геометрию амортизатора (работу функции Amort мы рассмотрим позже):

```

Shape = Amort(diameter, length, minlength, maxlength);

```

Создается AIS_InteractiveObject (AIS_Shape):

```

myAISShape = new AIS_Shape (Shape);

```

Назначается материал Пластик:

```
myAISShape->SetMaterial(Graphic3d_NOM_PLASTIC);
```

Назначается цвет из оператора задания Layer:

```
SetColor(myAISShape);
```

Объекту назначается режим отображения:

```
myAISContext->SetDisplayMode(myAISShape, 1, Standard_False);
```

Следующим выполняется метод Display. В нем графический объект отображается во вьювере:

```
myAISContext->Display(myAISShape,1,1,Standard_False,Standard_False);
```

и устанавливается его положение в пространстве:

```
myAISContext->SetLocation(myAISShape,aTrsf);
```

Далее, в процессе анимации используется метод Calculate. В этом методе происходит перерасчет пространственного положения и формы амортизатора, и объект перерисовывается методом **Redisplay(myAISShape, Standard_False)**.

```
double leng=sqrt(pow(nodes[NodeX2.ID].S+B.X()-nodes[NodeX1.ID].S-  
A.X(),2)+pow(nodes[NodeY2.ID].S+B.Y()-nodes[NodeY1.ID].S-  
A.Y(),2)+pow(nodes[NodeZ2.ID].S+B.Z()-nodes[NodeZ1.ID].S-A.Z(),2));  
Shape = Amort(diameter, leng, minlength, maxlength, nodes);
```

```
if(myAISShape.IsNull())  
{  
    myAISShape = new AIS_Shape (Shape);  
}else  
{  
    Handle(AIS_Shape)::DownCast(myAISShape)->Set(Shape);  
    myAISContext->Redisplay(myAISShape, Standard_False);  
}
```

Следует заметить, что во всех методах отображения, режим **updateviewer** должен быть установлен в значение **Standard_False**, что означает, что вьювер при этом не перерисовывается. Перерисовка будет сделана позднее в самом постпроцессоре один раз для всех ПГО, что существенно ускоряет процесс анимации.

Метод Refresh просто меняет пространственное положение объекта на основе пересчитанной трансформации **aTrsf**.

```
myAISContext->ResetLocation(myAISShape);  
myAISContext->SetLocation(myAISShape,aTrsf);
```

Теперь рассмотрим функцию Amort, в которой происходит расчет геометрии амортизатора.

В начале создается объект **TopoDS_Compound**, который позволяет создавать TopoDS_Shape со сложной конструкцией, состоящей из любого количества различных, не связанных друг с другом, графических объектов. Так же создается объект **BRep_Builder**, который позволяет работать с объектом **TopoDS_Compound**.

```
TopoDS_Compound comp;  
BRep_Builder builder;  
builder.MakeCompound( comp );
```

Далее создаем первый цилиндр, из которого будет состоять образ нашего амортизатора:

```
BRepPrimAPI_MakeCylinder cyl1(diameter1/2., minlength);
```

И помещаем его в **TopoDS_Compound**:

```
builder.Add( comp, cyl1.Shape() );
```

То же самое делаем и со вторым цилиндром:

```
BRepPrimAPI_MakeCylinder cyl2(diameter1/10., length);  
builder.Add( comp, cyl2.Shape() );
```

```
S = comp;
```

Далее, в зависимости от того, рисуется ли объект в самом начале (**nodes==NULL**), или же он рисуется в процессе анимации, производится вычисление его пространственного положения:

```
if(nodes==NULL)  
{  
    gp_Vec Vector(gp_Pnt(0,0,0),gp_Pnt(0,0,5));  
    gp_Vec v2(B.X()-A.X(),B.Y()-A.Y(),B.Z()-A.Z());  
    gp_Pnt Point(0,0,0);  
  
    double phi = acos(Vector*v2/Vector.Magnitude()/v2.Magnitude ());  
    if(fabs(phi)>=1e-8 && !Vector.IsParallel (v2,gp::Resolution()))  
    {  
        gp_Vec vec = Vector^v2;  
        gp_Ax1 ax1(Point, vec);  
        aTrsf.SetRotation(ax1, phi);  
    }  
    gp_Trsf trsf;  
  
    gp_Vec n1(A.X()-Point.X (),A.Y()-Point.Y (),A.Z()-Point.Z ());  
    trsf.SetTranslation(n1);  
    aTrsf=trsf*aTrsf;  
  
}  
else  
{  
    gp_Vec Vector(gp_Pnt(0,0,0),gp_Pnt(0,0,5));
```

```

gp_Vec v2(nodes[NodeX2.ID].S+B.X()-nodes[NodeX1.ID].S-A.X(),
          nodes[NodeY2.ID].S+B.Y()-nodes[NodeY1.ID].S-A.Y(),
          nodes[NodeZ2.ID].S+B.Z()-nodes[NodeZ1.ID].S-A.Z());
gp_Pnt Point(0,0,0);

double phi = acos(Vector*v2/Vector.Magnitude()/v2.Magnitude ());
if(fabs(phi)>=1e-8 && !Vector.IsParallel (v2,gp::Resolution()))
{
    gp_Vec vec = Vector^v2;
    gp_Ax1 ax1(Point, vec);
    aTrsf.SetRotation(ax1, phi);
}
gp_Trsf trsf;

gp_Vec n1(nodes[NodeX1.ID].S+A.X()-Point.X (),
          nodes[NodeY1.ID].S+A.Y()-
          Point.Y(),nodes[NodeZ1.ID].S+A.Z()-Point.Z ());
trsf.SetTranslation(n1);
aTrsf=trsf*aTrsf;
}

```

рассчитанная форма объекта возвращается в объекте S,

return S;

А рассчитанная трансформация сохраняется в объекте **aTrsf**, который потом используется при перерисовке амортизатора в методе Refresh.

Метод GetType возвращает строку, содержащую имя ПГО:

```

virtual inline QString GetType() const
{
    return "AMORT";
};

```

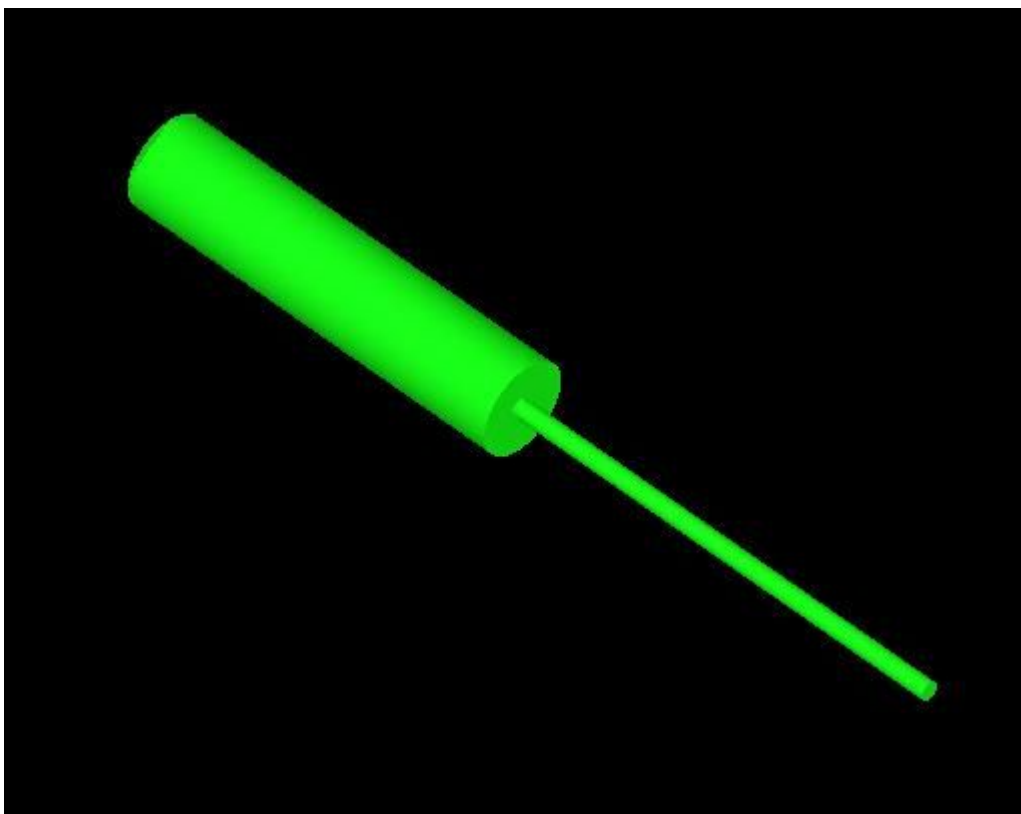
Метод GetModelClass возвращает строку, содержащую имя класса. Для механических ПГО оно всегда состоит из слова "**Mechanical**":

```

virtual inline QString GetModelClass() const
{
    return "Mechanical";
};

```

Получающийся образ амортизатора имеет следующий вид:



Получившаяся в результате компиляции этой ПГО DLL библиотека должна быть помещена в каталог **%DINSYS%\dinama\post**, а сведения о ней должны быть занесены в файл репозитория ПГО (PGO_List.txt).