Creating a CAM-system Add-in

Guide to creating a CAM-system Add-in: integrating CAD systems (as well as other geometry sources) with a CAM system and transferring geometry through the SGF API.

Contents

1. Introduction
    - Glossary
    - Requirements
2. Toolbar architecture
    - Component overview
        - Add-ins Wizard
        - Distribution files
        - Translator
        - Add-in XML file
        - CAMInfo.ini file
        - CheckLockToolbar utility
        - Toolbar
    - Interaction scenarios
        - Installing the Add-in
        - Uninstalling the Add-in
        - Import from a CAD system to a CAM system
        - Importing a CAD file into a CAM system
3. Importing geometry into CAM
    - CAD model structure
    - Import interfaces
    - Connecting STGeomFile.dll
    - Building the SGF file
    - SGF data structures
    - Working with an assembly
    - Working with a part
    - Curves and points
        - Point and analytic curves
            - 3D point
            - Coordinate system
            - Line segment
            - Circle
            - Circular arc
            - Ellipse
            - Elliptical arc

        - Conical spiral
        - NURBS curve
        - Composite curve
    - Derived curves
        - Curve on a surface
        - Trimmed curve
        - Inverted curve
        - Intersection curve
        - Blend boundary
    - Curve parametrization
- Face (trimmed surface)
- Surfaces
    - Analytic surfaces
        - Plane
        - Cylinder
        - Cone
        - Sphere
        - Torus
        - NURBS surface
        - Polygonal mesh
    - Derived surfaces
        - Surface of revolution
        - Extrusion surface
        - Transposed UV surface
        - Blend surface
        - Offset surface
    - Surface parametrization
- PMI (Product Manufacturing Information)
    - IPMIGeomThread
- Deprecated and unused methods

Introduction

This document describes the architecture and interaction protocols of CAM system Add-ins with CAD systems. Following these instructions, a developer can implement an Add-in for any CAD system that meets the minimum API requirements.

The document is not tied to a specific CAD system or a specific programming language. All examples of file formats and contracts are taken from real Add-ins (KOMPAS-3D, SolidWorks, etc.) and presented in a generalized form.

The main content of the document consists of two large chapters:

- Toolbar architecture — which components make up the Add-in (the Translator, the Toolbar, and the accompanying files), how they are connected, and in which scenarios they interact: installing and removing the Add-in, as well as the two paths for transferring geometry to the CAM system.
- Importing geometry into CAM — the core of the Add-in: how to use the API of the `STGeomFile.dll` library to build a file in the CAM system's internal format (SGF), transferring the model's topology and geometry into it — bodies, faces, surfaces, curves, and PMI.

Although the source of geometry in the examples is a CAD system, the geometry import itself is not tied to it. The CAD system can be replaced with any other source — for example, a neutral exchange format (STEP, IGES, Parasolid, etc.): only the way the source geometry is obtained changes, while writing it to the SGF stays the same. For this reason, the chapter "Importing geometry into CAM" can also be used as a standalone guide to converting to SGF from an arbitrary source.

Glossary

Term	Meaning
CAD system	The application into which the Toolbar is embedded. At a minimum it must allow third-party plugins to be registered, give them access to the currently open model, and provide a way to export geometry (through its own API or built-in export to STEP/Parasolid).
CAM system	The consumer of geometry. Loads the converted geometry from the CAD system.
Add-in	A complete package for integrating a CAD system with a CAM system. Consists of a Translator and a Toolbar.
Translator (`Translator`)	The EXE part of the Add-in. Performs installation/removal of the Add-in, and on request from the CAM system imports a CAD file into the CAM system's internal format.
Toolbar (`Toolbar`)	The part of the Add-in (usually a library) embedded in the CAD system as a button or a panel of buttons. When clicked, it converts and passes the current geometry to the CAM system.
Add-in Wizard (`AddinManager.dll`)	A CAM system UI tool for installing/removing Add-ins. It sends control commands to the Translator.
`HostApplication`	The string identifier of the CAD system (a tag of the same name in the Add-in XML file). The CAM system uses it to find the appropriate Add-in when geometry is passed from the Toolbar.
Add-in XML file (`<Translator>.xml`)	An XML file that describes the Add-in's metadata, supported formats, and commands.
`CAMInfo.ini`	A configuration file with certain CAM system parameters; it is created automatically by the Add-in Wizard.

Term	Meaning
CheckLockToolbar (`CheckLockToolbar.exe`)	A helper utility: checks whether a CAD system process is blocking installation/removal.
STGeomFile.dll	A library that provides access to the `ISTGeomFiler` interface — it contains methods for converting geometry into the CAM system's internal format (SGF).
SGF	Structured Geometry File. The CAM system's internal geometry format.
`ISTGeomFiler` (`sgf`)	The interface for managing an SGF file: open for writing (`StartFile`), close (`CloseFile`). Returned by the `CreateGeomFiler` function from STGeomFile.dll.
`ISTGeomReceiver` (`sgr`)	The main import interface: it receives geometry through a sequence of commands (surfaces, curves, faces, bodies, PMI). The same COM object as `ISTGeomFiler`.
ComboModel / ComboSolid	A pair of SGF objects for writing a closed body with stitching: the faces, edges, and vertices of a shell are assembled into a single solid (B-rep) without losing connectivity.
PMI	Product Manufacturing Information — manufacturing information annotations: dimensions, tolerances, threads, and so on.

Requirements

Before you begin developing an Add-in for the CAM system, make sure that the finished product will meet the following requirements:

1. CAD system licensing. The CAD-SDK files used by the Add-in must be permitted for distribution as part of the CAM system's distributive.
2. External install/uninstall. The Add-in must support installation and uninstallation without user interaction in the CAD system's UI — this is required for the Add-ins Wizard to work correctly.
3. Geometry export. The Add-in must be able to export geometry into one of the CAM system's internal formats.

Additionally — if the Add-in supports importing CAD files on request from the CAM system, one more requirement applies to it:

- Programmatic import of CAD files. The CAD system's API must allow a project file of that CAD system to be opened programmatically and converted — without interactive user involvement.

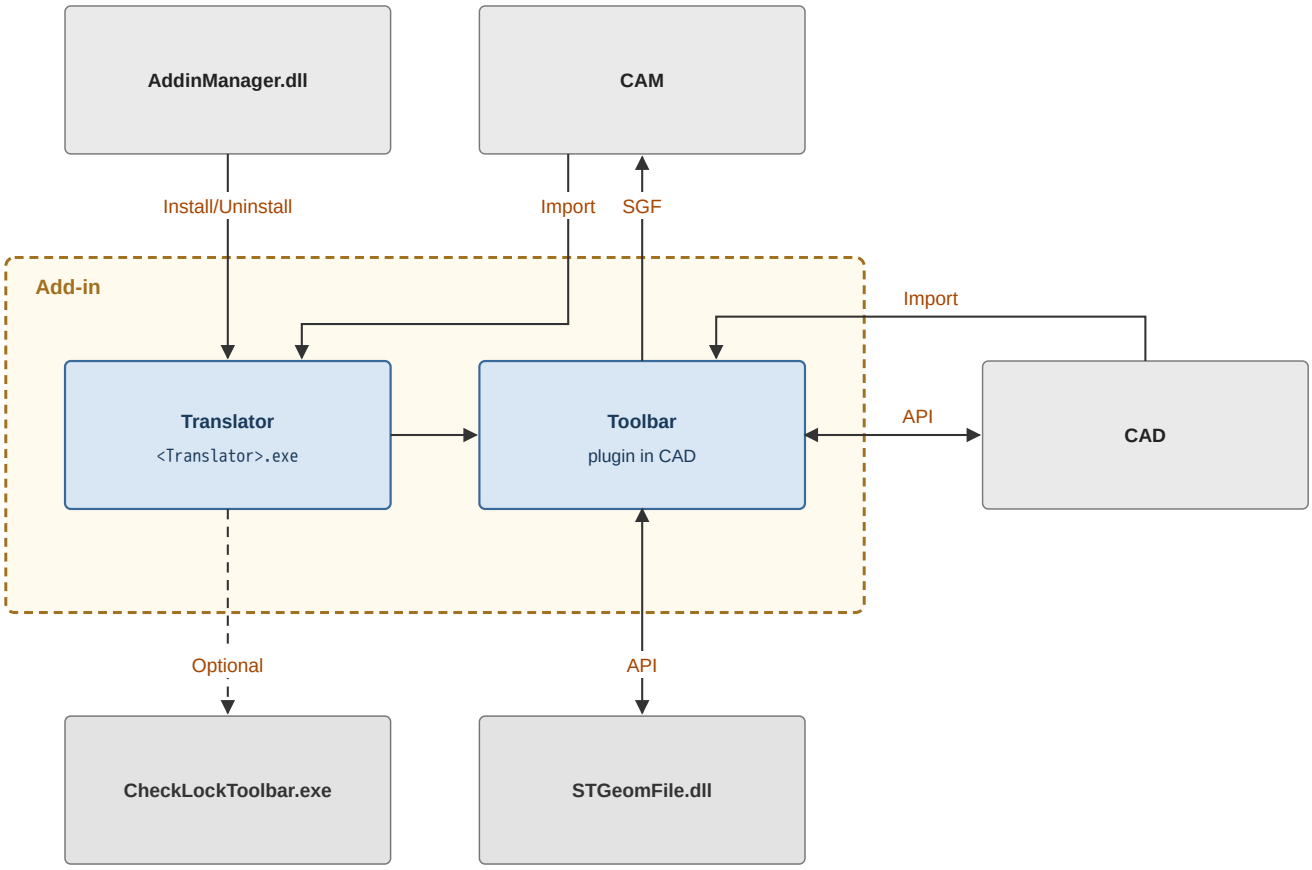
Toolbar architecture

The Add-in consists of two parts — the Translator and the Toolbar — which interact with three external participants: the Add-in Wizard, the CAD system, and the CAM system. This section describes their composition and their operating scenarios.

- Component description — which parts the Add-in consists of, which files are included in the distribution, and what each component is responsible for.
- Interaction scenarios — how the components interact with each other: installing and uninstalling the Add-in, as well as the two paths for transferring geometry into the CAM system.

Component description

An Add-in consists of two parts: the Translator (`<Translator>.exe`) and the Toolbar (a plugin in the CAD). They interact with three external participants — the Add-ins Wizard, the CAD system, and the CAM system.

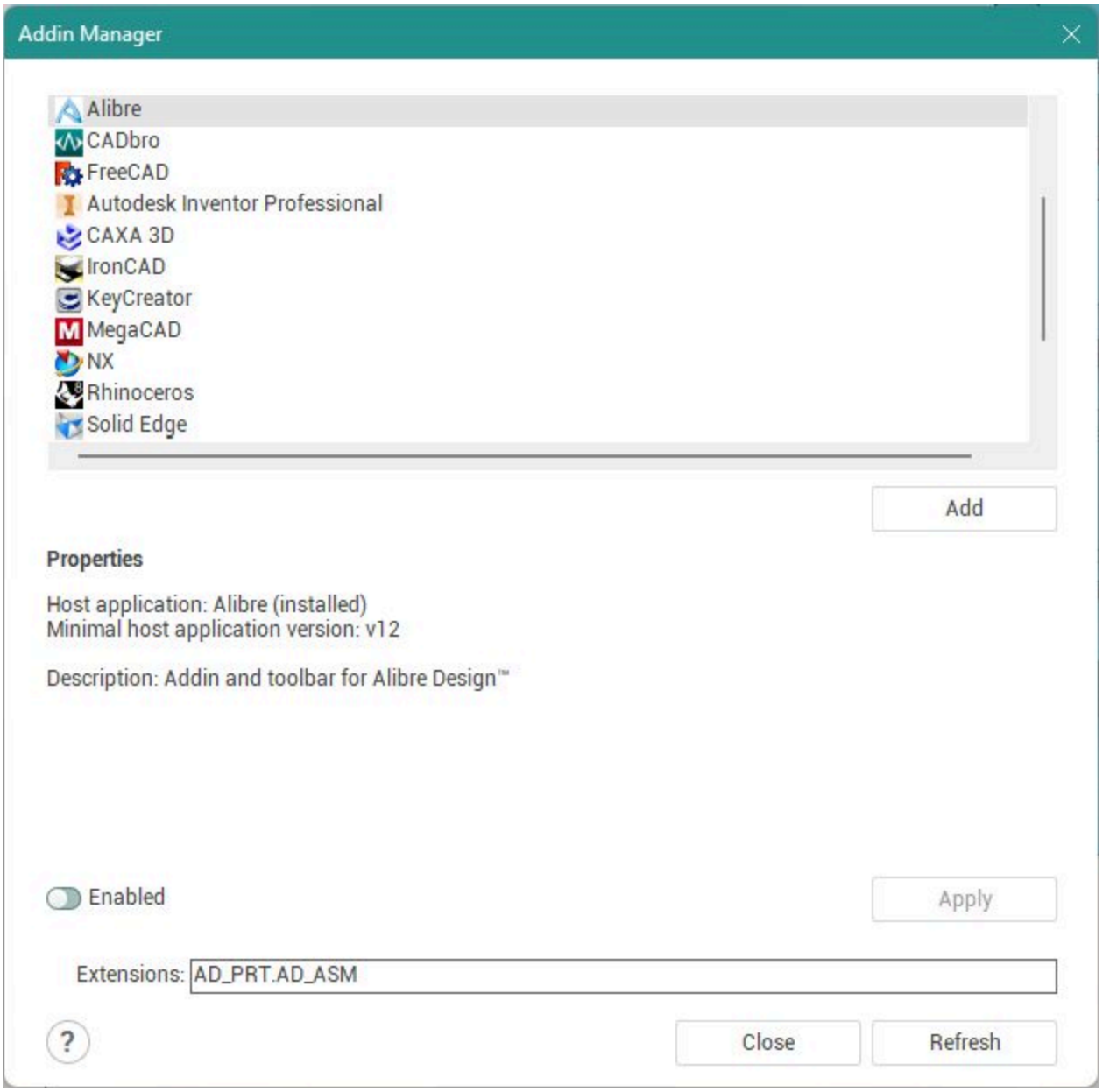


The Add-in components and its environment, as well as the distribution files, are described below:

- Add-ins Wizard — a CAM system UI tool (`AddinManager.dll`) for managing the Add-in lifecycle.
- Distribution files — where Add-ins are stored, the path template, and the mandatory folder contents.
- Translator — the single entry point `<Translator>.exe`: the command-line contract and the return codes.
- Add-in XML file — Add-in metadata, supported formats, and command names.
- CAMInfo.ini file — the current CAM system settings available to Add-ins.
- CheckLockToolbar utility — helper code (`LockProcManager`) that simplifies developing the Translator.
- Toolbar — a button in the CAD system that transfers geometry to the CAM system.

Add-in Wizard

A CAM system UI tool for managing the lifecycle of Add-ins. Implemented by the `AddinManager.dll` library that is part of the CAM system.



When you select an Add-in in the top list, all of its additional information is shown in the `Properties` panel. Usually this is the name and version of the application the Add-in works with, a description of the capabilities it provides, and other information.

The `Extensions` field appears on screen when the Add-in selected in the list operates on files with certain extensions (for example, import/export).

The `Enable` checkbox changes the installation setting of the selected Add-in.

The `Apply` button applies the value of the `Enable` checkbox to the selected Add-in.

The `Close` button closes the Add-in Wizard. If any changes were made, closing prompts the question: "Exit and save / Exit without saving / Cancel exit".

The `Refresh` button refreshes the list of Add-ins and all data about them. Any unsaved changes are lost in the process.

Distribution files

All Add-ins are stored in the CAM system's standard directory. This is the only place from which Add-ins are managed and interacted with — it also holds the files required for the Toolbar to work.

Path template.

```
%ProgramData%\<Company_name>\<Product_name>\<Product_version>\AddIns\
<CAD_system_name>\
```

For example, for the KOMPAS CAD system:

```
C:\ProgramData\<Company_name>\<Product_name>\<Product_version>\AddIns\Kompas\
```

The files of each Add-in are stored in a separate subfolder named after the CAD system. The Add-in wizard scans all `AddIns/` subfolders and offers them for installation.

Example of an Add-in's file structure.

```
AddIns/
  <CAD>/
    <Translator>.exe
    <Translator>.bmp
    <Translator>.xml
    STGeomFile.dll
    CAMInfo.ini
    <ToolbarFiles>
```

Required folder contents. Three files with the same name and different extensions; without them the Add-in wizard will not display this add-in:

File	Purpose
`<Translator>.exe`	The Translator executable (see Translator).
`<Translator>.bmp`	A 16×16 icon shown in the Add-in wizard UI.
`<Translator>.xml`	The Add-in XML file (see Add-in XML file).

Accompanying files.

File	Purpose
`CAMInfo.ini`	Created automatically by the Add-in wizard. Contains the current CAM system settings (see CAMInfo.ini file).
Toolbar file (extension depends on the CAD)	The plugin file that converts the CAD system's geometry and embeds itself into its toolbar (see Toolbar).
CAD system UI resources	Button icons, fonts, panel configuration XML/CUIX files — the format depends on the CAD. Usually placed in a subfolder.

Any other files (if present) should be placed in a separate subfolder (for example, `ToolbarFiles/`) — this makes the structure easier to read and does not interfere with the Add-in wizard finding the required files.

Translator

A single entry point for commands from the Add-in Wizard and for the import command from the CAM system. A standalone process, an executable `.exe` file. The implementation language is arbitrary; most commonly Delphi, C++, or C# (.NET Framework).

Command-line contract. `<Translator>.exe` accepts one of four argument formats:

Arguments	Semantics
`IsInstalled`	Check whether installation is possible — namely, that the CAD system is present on the machine and all conditions for installation are met. On success, return 10. Otherwise the Add-in Wizard will block the ability to install.
`Install`	The Translator performs the operations needed to make a Toolbar with a button appear in the CAD system. On success, return 10. Otherwise the Add-in Wizard will show an error.
`UnInstall`	Remove the Toolbar from the CAD system. Same as the command above; as a result, the button should disappear from the CAD system. On success, return 10.
`$(InFile) $(OutFile)`	Import of a CAD file. The Translator launches the CAD system (or connects to an already running one), creates a new document and loads `InFile` into it, then imports all the geometry (usually by means of the Toolbar) and produces the `OutFile`. For details, see Importing a CAD file into a CAM system.

The command argument names (`IsInstalled`, `Install`, `UnInstall`, and the `<In> <Out>` format) are specified in the Add-in XML file through the tags `CommandLineIsAppInst`, `CmdLnInstall`, `CmdLnUnInstall`, `CommandLine`. The names may differ in each specific Add-in — the Add-in Wizard and the CAM system read them from the XML.

Return codes.

Code	Meaning
`0`	False — the operation was not performed (the CAM system will show an error window).
`10`	True — the operation completed successfully.

Add-in XML file

Contains the Add-in's metadata, a description of the supported formats, and the names of the commands with which the Add-in wizard calls `<Translator>.exe`.

Root structure.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Collection>
    <CADTranslators>
        <TranslatorName>Alibre</TranslatorName>
        <TranslatorComposition AddinConnected="-1" ToolbarConnected="-1"/>
        <IsExternal>-1</IsExternal>
        <WantAdmin>0</WantAdmin>
        <Enabled>0</Enabled>
        <HideWinow>-1</HideWinow>

<TranslatorPath>C:\ProgramData\Company_name\Product_name\Product_version\AddIns\Alib
re\AlibreTranslator.exe</TranslatorPath>
        <OutputExtension>IGS</OutputExtension>
        <InputExtensions>AD_PRT.AD_ASM</InputExtensions>
        <CommandLine>$(InFile) $(OutFile)</CommandLine>
        <CommandLineIsAppInst>IsInstalled</CommandLineIsAppInst>
        <CmdLnInstall>Install</CmdLnInstall>
        <CmdLnUnInstall>UnInstall</CmdLnUnInstall>
        <HostApplication>Alibre</HostApplication>
        <DeprecatedHostApplications>
            <Item>GeoMagic</Item>
        </DeprecatedHostApplications>
        <MinimalVersion>v12</MinimalVersion>
        <CADPlatform>xAny</CADPlatform>
        <Description>
            <Item LangCode="1033">Addin and toolbar for Alibre Design™</Item>
            <Item LangCode="1049">Дополнение и панель инструментов для
Alibre Design™</Item>
        </Description>
        <Copyright>Company name</Copyright>
        <HostCopyright>© Alibre, LLC</HostCopyright>
        <BuildInfo PackageID="" PackageVersion="" BuildNumber=""/>
    </CADTranslators>
</Collection>
```

Tag descriptions. Pay particular attention to `HostApplication` — its value is used by the Toolbar when transferring geometry to the CAM system.

Tag name	Description	Value
`TranslatorName`	The Add-in name shown in the Add-in wizard list.	—
`TranslatorComposition`	The `AddinConnected` and `ToolbarConnected` attributes indicate which import scenarios will be available.	`AddinConnected="-1"` — import from the CAM system is available (see Importing a CAD file into the CAM system); `ToolbarConnected="-1"` — import from the CAD system is available (see Importing from the CAD system into the CAM system).
`IsExternal`	Indicates that this is an external importer.	Must be equal to `-1`.
`WantAdmin`	Indicates whether the installer needs to be run with administrator rights.	`0` — False; `-1` — True.
`WantAdminForInstallation`	Indicates whether administrator rights are required for installation.	`0` — False; `-1` — True.
`Enabled`	Indicates the application's installation state.	Defaults to `0`. Set automatically.
`HideWinow`	Indicates whether the installer should be run in hidden mode.	`0` — False; `-1` — True.
`TranslatorPath`	Path to the Translator executable.	Empty by default. Set automatically.
`OutputExtension`	The file format into which the CAD system's geometry will be converted.	Specify the extension (for example, `SGF`, `IGS`, `STEP`).
`InputExtensions`	The file formats the Translator can import. When importing a file, the CAM system uses this	Usually the CAD system's native project extensions are specified here. Multiple extensions are

Tag name	Description	Value
	tag to find the appropriate Add-in.	separated by `.` (for example, `CDW.FRW.A3D`).
`CommandLine`	The command for import. Originally it was assumed the number of elements could vary, but in the end two were settled on.	Must be equal to `$(InFile) $(OutFile)`.
`CommandLineIsAppInst`	The command to check whether installation is possible.	Usually `IsInstalled`.
`CmdLnInstall`	The command to install the Add-in.	Usually `Install`.
`CmdLnUnInstall`	The command to uninstall the Add-in.	Usually `UnInstall`.
`HostApplication`	The CAD system's string identifier. Used by the Toolbar when transferring geometry to the CAM system — the CAM system uses it to find the registered Add-in.	Usually the value is duplicated from the `TranslatorName` tag. The key requirement is that it must be unique among the other Add-ins.
`DeprecatedHostApplications`	Old identifiers for backward compatibility with projects created in earlier versions of the CAM system.	A list of `<Item>` entries with the former `HostApplication` values.
`MinimalVersion`	The minimum CAD system version on which the Add-in can be installed.	A free-format string (for example, `v12`, `20+ x64`).
`CADPlatform`	Not used.	`xAny` by default.
`Description`	The description block shown in the Add-in wizard window. Split into sub-blocks with `LangCode` (locale, code page number) for display in different languages.	`<Item LangCode="1033">` — EN; `<Item LangCode="1049">` — RU.

Tag name	Description	Value
`Copyright`	Copyright of the Add-in owner.	—
`HostCopyright`	Copyright of the CAD system.	—
`BuildInfo`	Build metadata (attributes `PackageID`, `PackageVersion`, `BuildNumber`).	Optional.

CAMInfo.ini file

The `CAMInfo.ini` file contains the current settings of the CAM system that are used by Add-ins. The file is created automatically during a scan by the Add-ins Wizard in the directory of each Add-in. If necessary, the list can be extended with new parameters.

Example CAMInfo.ini file:

```
[CAM]
CAMName=<Product_name> <Product_version>
CompanyName=<Company_name>
CompanyURL=<company_website>
CAMPath=C:\Program Files\<Company_name>\<Product_name>\Bin64\<CAM>.exe
CAMLinearMeasure=lmMillimetre
CAMCfgPath=C:\Users\<User>\AppData\Roaming\<Company_name>\
<Product_name>\<Product_version>
LanguageID=1049
CompanyLocalizedName=<Company_name>
```

Parameter descriptions.

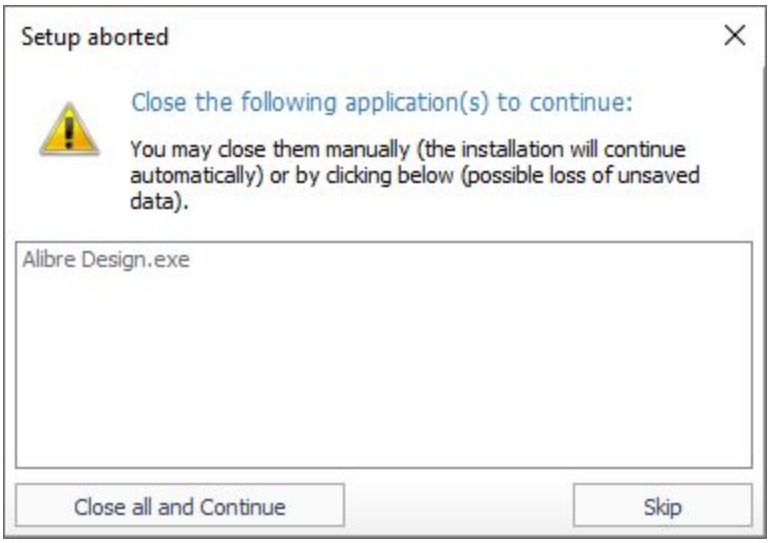
Parameter	Description
`CAMName`	The official name of the installed product.
`CompanyName`	The official company name. Since different companies are used for different markets, it is better to use `CompanyLocalizedName`.
`CompanyURL`	Link to the website.
`CAMPath`	Path to the CAM system's executable file.
`CAMLinearMeasure`	The current units of measurement. Possible values: `lmMillimetre`, `lmCentimetre`, `lmDecimetre`, `lmMetre`, `lmInch`, `lmFoot`.
`CAMCfgPath`	Path to the settings file.
`LanguageID`	The current encoding number used in the CAM system (LCID).
`CompanyLocalizedName`	The localized company name.

CheckLockToolbar utility

CheckLockToolbar is a repository of helper code that simplifies development of the Translator. The `LockProcManager` unit (implementations in both Delphi and C# are available) is added to the project; its functions are described below. The repository also builds the `CheckLockToolbar.exe` utility, which can be invoked as a separate process if adding the unit is undesirable.

`IsTerminateProc(ProcessNames: string): boolean`

Checks whether the specified processes are running in the system. If any are found, it shows a dialog:



This dialog lets you forcibly terminate the specified processes or ignore them.

The function can be used during installation/uninstallation of the Add-in, since access to the required files can sometimes be locked by the CAD system.

Parameters:

Parameter	Description
`ProcessNames`	Process names separated by spaces, for example `"Alibre Design.exe"`.

Result:

Value	Description
`True`	The specified processes are not active or were closed.

Value	Description
`False`	The specified processes could not be closed, or the user cancelled closing them.

`RunAs(Filename, Parameters: string; Admin: boolean; WinState: DWORD; Wait: boolean = true): boolean`

Runs an executable file with parameters.

Parameters:

Parameter	Description
`Filename`	Name of the executable file.
`Parameters`	List of parameters, separated by spaces.
`Admin`	Flag: run with administrator privileges.
`WinState`	Process startup parameters (for example, `SW_HIDE`, `SW_SHOW`).
`Wait`	Flag: wait for the process to finish.

Result:

Value	Description
`True`	The file was started successfully.
`False`	An error occurred while starting.

`RunRegsvr32(DllPath, Param: string): boolean`

Registers/unregisters a library using the `regsvr32` utility, essentially acting as a wrapper for it.

Parameters:

Parameter	Description
`DllPath`	Path to the library.

Parameter	Description
`Param`	Parameters for the `regsvr32` utility: `dllregisterserver` for registration, `dllunregisterserver` for unregistration.

Result:

Value	Description
`True`	The library was successfully registered/unregistered.
`False`	Error.

Toolbar

The Toolbar is a button in the CAD system that, when activated, transfers geometry to the CAM system. Depending on the structure of the CAD system, the Toolbar can have a different appearance and a different location — in a menu, in a shared panel, in a separate panel, and so on.

All the data required for the Toolbar to work is located in two files:

- **`CAMInfo.ini`** (see The CAMInfo.ini file) — data for filling in names, tooltips, as well as the path to the CAM system and the settings files;
- The add-in XML file (see The add-in XML file) — the value of the `HostApplication` tag, by which the CAM system links the transferred geometry to the correct importer.

Since in some cases the Toolbar application may reside in a different directory, when installing you must provide access to the folder with these files.

Additionally, you can use the CheckLockToolbar utility (see The CheckLockToolbar utility).

Algorithm on button press.

1. Save the model file in the CAD system's native format (hereinafter — the CAD system project file).
    - If the model file did not exist (this is a new project), you need to show a "Save As…" dialog. In most cases this happens automatically when the corresponding API method is called; if not, you must implement this part yourself.
    - If the user declines to save, you must abort execution.
2. Convert the file into one of the CAM system's internal formats. If possible, it is preferable to use STEP or Parasolid; otherwise, build SGF (see Geometry import into the CAM system).
3. Launch the CAM system with command-line parameters (see the example call below).

Example of a CAM system call.

```
void LaunchCAM(string fileName, string originalFileName)
{
    if (File.Exists(fCAMPath))
    {
        string par =
            " -EXTIMPORT -HOST \"" + fHost + "\"" +
            " -ORIGINALFILEPATH \"" + originalFileName + "\"" +
            " -DELETEIMPORTFILE \"" + fileName + "\"";

        RunAs(fCAMPath, par, false, SW_SHOW, false);
    }
}
```

This example uses the following variables:

Variable	Description
`fCAMPath`	The full path to the CAM system executable. Read from the `CAMInfo.ini` file (the `CAMPath` parameter).
`fHost`	The value of the `HostApplication` tag from the add-in XML file. This value is unlikely to change, so it can be hard-coded in the source — but you need to keep that in mind.
`originalFileName`	The CAD system project file.
`fileName`	The converted file.

The `RunAs` function is from the `LockProcManager` unit (see The CheckLockToolbar utility).

Interaction scenarios

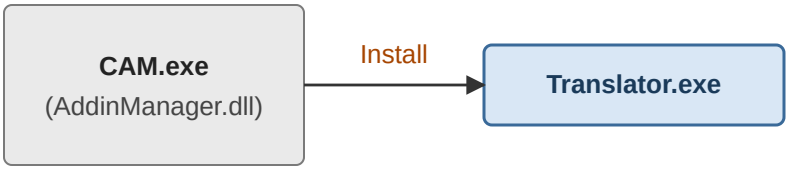
The Add-in components are involved in four scenarios, which fall into two groups:

- Add-in life cycle — installation and uninstallation; these are managed by the Add-in Manager (`AddinManager.dll`), which calls the Translator.
- Transferring geometry to the CAM system — by pressing the Toolbar button in an open CAD model, and importing a CAD file on request from the CAM system.

Each scenario involves its own set of components and connections (shown in the diagrams below). The CAD system does not directly "know" about the CAM system — the connection is provided by the Toolbar and the Translator (`<Translator>.exe`).

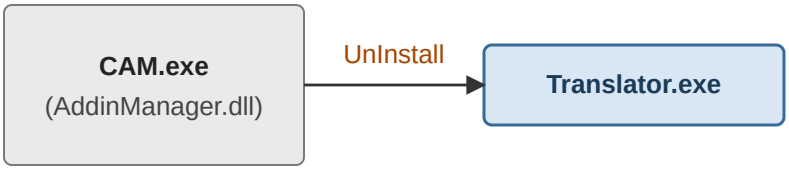
- Installing the Add-in — the Add-in Manager calls the Translator to embed the Toolbar into the CAD system.
- Uninstalling the Add-in — the Add-in Manager calls the Translator to remove the Toolbar from the CAD system.
- Import from the CAD system into the CAM system — the user exports the open model by pressing the Toolbar button.
- Import of a CAD file into the CAM system — the CAM system imports a CAD file by calling the Translator on an already loaded model.

Installing the Add-in



1. The user opens the Add-in wizard in the CAM system.
2. The Add-in wizard scans the standard folder and shows a list of the Add-ins found.
3. The user selects the desired Add-in.
4. The Add-in wizard calls `<Translator>.exe IsInstalled`. If the return code is not `10` (False), the `Apply` button is disabled.
5. The user selects the `Enable` checkbox and clicks the `Apply` button.
6. The Add-in wizard calls `<Translator>.exe Install`.
7. The Translator installs the Toolbar into the CAD system. If the return code is not `10` (False), an error message is shown to the user.

Uninstalling the Add-in



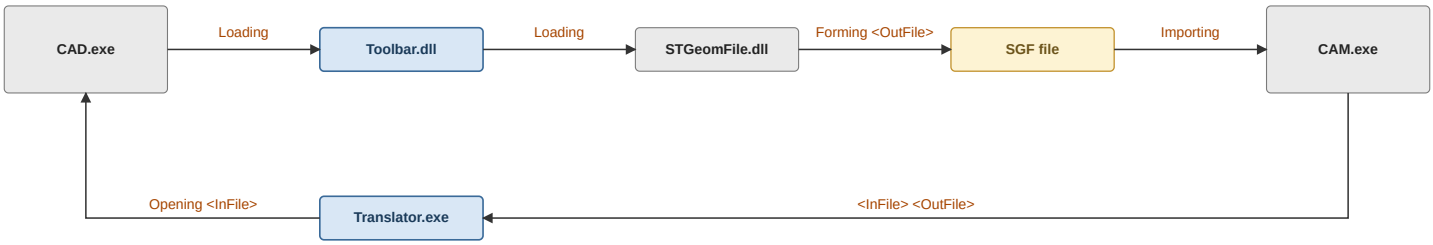
1. The user opens the Add-ins Wizard in the CAM system.
2. The Add-ins Wizard scans the standard folder and shows a list of the Add-ins found.
3. The user selects an installed Add-in. Installed Add-ins have the `Enable` checkbox checked.
4. The user clears the `Enable` checkbox and clicks the `Apply` button.
5. The Add-ins Wizard calls `<Translator>.exe UnInstall`.
6. The Translator removes the Toolbar from the CAD system. If the return code is not equal to `10` (False), an error message is shown to the user.

Import from a CAD system into a CAM system



1. The user works in the CAD system, creating or opening a model.
2. The user clicks the Toolbar button (which was added when the Add-in was installed).
3. The Toolbar saves the file in the CAD system's native format (for a new file — the "Save As…" dialog; if it is cancelled, the process is aborted).
4. The Toolbar converts the geometry into one of the CAM system's internal formats (into SGF for direct import, or into any other supported format) as a temporary file.
5. The Toolbar launches the CAM system with command-line parameters (see Toolbar).
6. The CAM system opens, imports the passed file, and deletes the temporary file.

Import a CAD file into a CAM system



1. In the CAM system, the user imports a CAD file.
2. By the file extension, the CAM system finds the appropriate Add-in, and if there are several, it shows the user a dialog to choose one.
3. The CAM system calls `<Translator>.exe <InFile> <OutFile>` (see Translator).
4. The translator launches an instance of the CAD system (in hidden mode where possible, so as not to disturb the user's session).
5. The translator creates a new CAD-system document, loads `InFile` into it, and launches the Toolbar for import, passing it `OutFile` as the output file.
6. The Toolbar converts the geometry into one of the CAM system's internal formats (into SGF for direct import, or into any other supported format) and saves the result to `OutFile`.
7. The translator closes the document and the open instance of the CAD system.
8. The CAM system loads the geometry from `OutFile` and deletes it.

Geometry import into the CAM system

This chapter describes the core of the Add-in — transferring the model geometry from the CAD system to the CAM system. The geometry available through the CAD system API is converted into a file in the CAM system's internal format, which the CAM system then loads.

The base format for such a transfer is SGF (Structured Geometry File). It stores the topology (faces, edges, vertices) and the geometry (surfaces, curves, points) as a hierarchy of linked entities.

The file itself is built by the STGeomFile.dll library: it ships together with the CAM system and provides an API for building solids step by step — first the geometry primitives (surfaces, curves) are described, then the topology (faces and edges) is built on top of them, and finally the result is saved to a file.

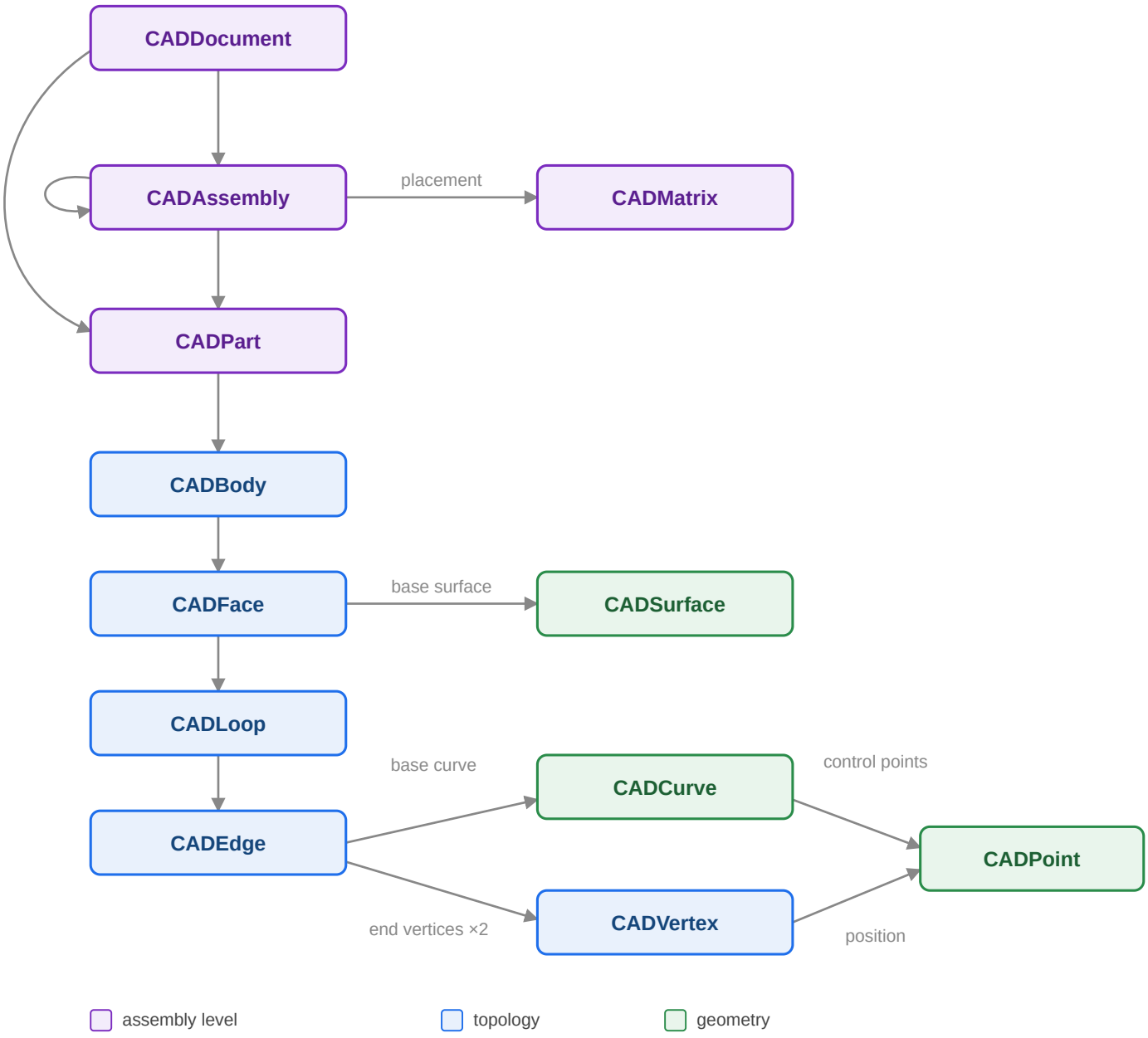
The chapter is organized around the principle of "first *what* we import, then *how*". First, an abstract CAD model structure is introduced — a hierarchy of objects (`CADDocument`, `CADPart`, `CADBody`, ⋯) that in the subsequent examples we will transfer into SGF. Then come the two import interfaces (`ISTGeomFiler` and `ISTGeomReceiver`), how to connect the library in order to obtain them, how to build the file itself, and the basic SGF data structures — the types on which all the signatures are written. After that comes working with assemblies: traversing the tree of components and their placement, and working with a part: saving its solids. Then comes geometry import by type: curves and points, faces, surfaces; PMI annotations; deprecated and unused methods.

> Example language. All code examples in this chapter are given in C# — for consistency and clarity. The API itself is not tied to a language: it can be accessed from any language with COM support (`ISTGeomFiler` / `ISTGeomReceiver`).

CAD model structure

> This is a notional (fictitious) structure. The `CADxxx` classes described below are *not* part of the CAM-system API and do not relate to any particular CAD system. This is a simplified data model of a "hypothetical" CAD, introduced solely so that the code examples show *where* the source data comes from and *how* it maps to SGF calls. In a real Add-in, the types of your CAD system (or its API) will take their place, while the conversion logic stays the same.

A CAD model is a hierarchy of related objects: from the document as a whole down to individual points. Importing into SGF is essentially a traversal of this hierarchy, translating each object into the corresponding `ISTGeomReceiver` call.



Model entities (top-down — from the assembly level to geometric primitives):

Entity	What it is	Imported via
CADDocument	The model root — the entire CAD document. The `IsDetail` flag determines the content: a part (`CADPart`) or an assembly (`CADAssembly`).	model tree
CADAssembly	Assembly: a component tree. For each component, the `IsDetail` flag selects a part (`CADPart`) or a nested assembly (`CADAssembly`); the position is defined by a matrix (`CADMatrix`).	Blocks / instances
CADMatrix	Placement matrix: the position and orientation of an assembly component in the parent's coordinate system.	placement matrix
CADPart	Part: one or more bodies plus attributes (name, color).	parts and bodies
CADBody	Body: a shell of faces describing the topology (faces, edges, vertices).	ComboSolid
CADFace	Face: a region of the underlying surface bounded by loops.	trimmed surface
CADSurface	The underlying surface of a face: plane, cylinder, cone, sphere, torus, NURBS, mesh, etc.	surfaces
CADLoop	Face loop — a closed sequence of edges. The outer loop defines the boundary, the inner ones define holes.	face loops
CADEdge	Edge: a segment of the underlying curve (`CADCurve`)	trimmed curve / edge

Entity	What it is	Imported via
	between two vertices (`CADVertex`).	
CADVertex	Vertex — the end point of an edge; its position is defined by a point (`CADPoint`).	body (ComboSolid)
CADCurve	The underlying curve of an edge: line segment, arc, ellipse, NURBS, intersection curve, etc.	curves
CADPoint	A point in 3D (X/Y/Z coordinates) — the base primitive: the position of vertices (`CADVertex`), and the control and support points of curves and surfaces.	as a parameter of other calls

Relationships between entities:

- CADDocument → CADPart | CADAssembly — by the `IsDetail` flag: a document is either a single part or an assembly;
- CADAssembly → CADPart | CADAssembly — by the `IsDetail` flag, an assembly component is a part or a nested assembly (hence the tree recursion);
- CADAssembly → CADMatrix — an assembly stores the placement matrix of a component;
- CADPart → CADBody — a part consists of bodies;
- CADBody → CADFace — a body is bounded by a set of faces;
- CADFace → CADSurface (1) + CADLoop (1..*) — a face has one underlying surface and one or more loops;
- CADLoop → CADEdge — a loop consists of edges;
- CADEdge → CADCurve (1) + CADVertex (2) — an edge relies on one underlying curve and is bounded by two vertices;
- CADVertex → CADPoint — a vertex is defined by a position point;
- CADCurve / CADSurface → CADPoint — points serve as control points (for NURBS) and support points.

In the following chapters, the examples are laid out as `SaveXxx` methods that receive such a `CADxxx` object and translate it into SGF.

Import interfaces

Geometry import relies on two interfaces of the same COM object of the STGeomFile.dll library (for how to obtain it, see Connecting STGeomFile.dll): `ISTGeomFiler` (`sgf`) manages the file itself, while `ISTGeomReceiver` (`sgr`) writes geometry into it.

ISTGeomFiler

A pointer to the `ISTGeomFiler` interface is returned by the exported `CreateGeomFiler` function (see Connecting STGeomFile.dll); this interface manages the file:

Method	Description
`StartFile(FileName: string)`	Begin building the SGF file (open it for writing). A nested `StartFile` call while a file is already open is not allowed.
`CloseFile()`	Close the SGF file. It is at this call that the file is physically built (written to disk).
`OpenFile(FileName: string; Receiver: ISTGeomReceiver)`	Open the SGF file for reading; commands are replayed into the specified `ISTGeomReceiver`. This method is used by the CAM system.

The workflow for working with the file is described in the chapter Building the SGF file.

ISTGeomReceiver

The main import interface: it receives geometry as a sequence of commands. All methods return a `boolean` — the success status (omitted in the signatures below). Objects are created under a unique string identifier `ID` — subsequent calls reference them by it. The parameter types (`STFloat`, `TST3DPoint`, `TST2DPoint`, and others) are described in Data structures.

An object's local coordinate system is usually defined by a triple of points: `vT` — the position (origin), `vZ` — the Z axis, `vX` — the X axis. If the Y axis is not passed among the parameters, it is computed internally as the cross product `vZ × vX`, which automatically yields a right-handed coordinate system. In this case `vZ` and `vX` must be mutually perpendicular, otherwise the axis orientation is ambiguous.

Some methods come in pairs: a method with the `Start…` prefix opens a compound object, within which nested methods are called (for example, adding points, edges, or knots), and at the end the object must be closed with the matching `Close…`-prefixed method. For example: `StartNurbsCurve` → `SetNurbsCurveControlPoint` / `SetNurbsCurveKnot` / ⋯ → `CloseNurbsCurve`; `StartFace` → `StartFaceLoop` → `AddFaceEdge2d` → `CloseFaceLoop` → `CloseFace`. Such pairs may be nested. If an

opened `Start…` is not closed with the matching `Close…`, at best the object will be built incorrectly, and at worst the SGF file structure will be corrupted and reading impossible.

> The referenced object comes before the dependent one. Regardless of the order of presentation, when writing to SGF each object must be created before the one that references it: a curve before an edge, a surface before a face, faces before a body. In the examples this is ensured by recursion: the top-level method (`SaveClosedShell`, `SaveTrimmedFace`, ⋯) first saves the child objects and then assembles the compound object on top of them.

Connecting STGeomFile.dll

`STGeomFile.dll` provides access to the `ISTGeomFiler` interface. The library ships as part of the Add-in (see Distribution files). You will also need two TLB files — they describe the interfaces (`ISTGeomFiler`, `ISTGeomReceiver`) and the enumerations, which gives typed access to the API:

- `STGeomApiTypes.tlb`;
- `STTypes.tlb`.

The library exports the function **`CreateGeomFiler`**, which returns a reference to the `ISTGeomFiler` interface.

Example of loading the library.

```
[DllImport("kernel32.dll")]
public static extern IntPtr LoadLibrary(string dllToLoad);
[DllImport("kernel32.dll")]
public static extern IntPtr GetProcAddress(IntPtr hModule, string procedureName);
[DllImport("kernel32.dll")]
public static extern bool FreeLibrary(IntPtr hModule);

// Signature of the exported CreateGeomFiler function
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
private delegate ISTGeomFiler TCreateGeomFiler();

private IntPtr            hDLL;
protected ISTGeomFiler    sgf;   // building the file: StartFile / CloseFile (see
"Building the SGF file")
protected ISTGeomReceiver sgr;   // writing geometry: methods in the geometry
sections and beyond

public bool ConnectToGeomFiler(string DLLPath)
{
    hDLL = IntPtr.Zero;
    sgf  = null;
    sgr  = null;

    // releases already acquired resources and signals failure
    bool Fail()
    {
        sgr = null;
        sgf = null;
        if (hDLL != IntPtr.Zero) { FreeLibrary(hDLL); hDLL = IntPtr.Zero; }
        return false;
    }
```

```
    try
    {
        if (!File.Exists(DLLPath))
            return Fail();

        hDLL = LoadLibrary(DLLPath);
        if (hDLL == IntPtr.Zero)
            return Fail();

        IntPtr pCreateGeomFiler = GetProcAddress(hDLL, "CreateGeomFiler");
        if (pCreateGeomFiler == IntPtr.Zero)
            return Fail();

        var CreateGeomFiler =
(TCreateGeomFiler)Marshal.GetDelegateForFunctionPointer(
            pCreateGeomFiler, typeof(TCreateGeomFiler));

        sgf = CreateGeomFiler();
        if (sgf == null)
            return Fail();

        // the same object exposes the geometry-writing interface (in C#,
QueryInterface is a type cast)
        sgr = sgf as ISTGeomReceiver;
        if (sgr == null)
            return Fail();

        return true;
    }
    catch
    {
        return Fail();
    }
}
```

Thus, `sgf` and `sgr` are the same COM object (see Import interfaces).

Building the SGF file

This is the entry point of geometry import — the moment when control is handed to the Add-in from outside. There are two possible scenarios (see Interaction scenarios):

- clicking a Toolbar button in the CAD system — the user exports the open model (Import from a CAD system into a CAM system);
- a call from a translator or from the CAM system when importing a CAD file — the Add-in is launched on an already loaded model (Import a CAD file into a CAM system).

In both cases the input is the same: the current model of the CAD system (what to save) and the path to the output file (where). From there the Add-in builds the SGF file following a single scheme:

1. connect to the import library (see Connecting STGeomFile.dll);
2. traverse the CAD model top-down and write its geometry through `sgr` (see the sections Curves and points–PMI);
3. release the interfaces and unload the library (`FreeLibrary`).

Global import settings

Right after `StartFile`, before geometry is written, the global import settings are specified. They apply to the whole file and are set once through the `ISTGeomReceiver` interface (`sgr`, see Import interfaces):

Method	Description
`SetModelUnits(Units: TST_LinearMeasure; MultScale: STFloat)`	Set the model units of measurement (`lm_Millimetre`, `lm_Centimetre`, `lm_Decimetre`, `lm_Metre`, `lm_Inch`, `lm_Foot`). `MultScale` is a scale factor. The defaults are `lm_Millimetre` and `MultScale = 1`.
`SetArcToler(Value: STFloat)`	Set the arc approximation tolerance. The default is `0.001` (in model units).
`SetImportOption(OptionType: TSTImportOption; OptionValue: boolean)`	Enable/disable import option `OptionType` (value `OptionValue`). The `TSTImportOption` values and their purpose are in the table below.

The `TSTImportOption` options control the import of face loops — primarily for inverted (reversed-normal) and cyclic (parameter-periodic) surfaces. Usually there is no need to change them: the default values are suitable for most cases, and the options are touched only for specific problems with orientation or with reconstructing loops. Possible values:

TSTImportOption	Purpose	Default
ioCreateRectLoops	Reconstruct the outer loop for a cyclic face if it is not given explicitly: a rectangular (outer) loop is added over the parametric region of the surface.	off
ioInverse3dLoopsForInversedFaces	Reverse the traversal direction of 3D loops (loops made of spatial edges) for faces with an inverted surface — so that the "face body on the left" rule is preserved.	on
ioInverse2dLoopsForInversedFaces	The same, but for 2D loops specified in the surface UV parameters.	off

SGF header metadata

The SGF file starts with a header: a signature, the format version and a JSON metadata block. The header as a whole is built by the library at `CloseFile`; the Add-in only extends the JSON block with information about the source — which file, which CAD system and which version of the Add-in produced the SGF. The CAM system shows this information to the user — for example, the name of the source model. The `originalFileName` and `hostName` keys are mandatory; the rest are filled in when possible.

The metadata is written through `sgr.Header.MetaWriter` (the `ISGFMetaWriter` interface) strictly between `StartFile` and `CloseFile` — outside this window `sgr.Header` returns `null`. The root JSON object is owned by the library: pairs are added right away, without opening the root, and every `BeginObject`/`BeginArray` is closed by its paired `EndObject`/`EndArray` (just like the geometric `Start…`/`Close…`). The methods return a `boolean` — the success status:

Method	Description
`AddStrPair(AKey, AValue: string)`	Add a "key — string value" pair. Similarly `AddIntPair` (`integer`), `AddFltPair` (`STFloat`), `AddBolPair` (`boolean`).
`BeginObject(ObjectID: string)` / `EndObject()`	Open / close a nested object with the given key.

Method	Description
`BeginArray(ArrayID: string)` / `EndArray()`	Open / close a nested array with the given key.
`AddStrValue(AValue: string)`	Add a value to the open array. Similarly `AddIntValue`, `AddFltValue`, `AddBolValue`.

The standard set of keys is given in the table. The names in the table are the keys: the CAM system reads the values exactly by these names, so write them exactly as specified, without renaming. Custom keys may be added — the format version does not change because of that, and unknown keys are ignored on reading.

Key	Contents	Mandatory
`originalFileName`	The full path to the CAD system project file (see Toolbar).	mandatory
`hostName`	The value of the `HostApplication` tag — the string identifier of the CAD system by which the CAM system finds the Add-in (see Add-in XML file and Glossary).	mandatory
`sourceApp`	A nested object: `name` — the name of the CAD system executable, `processFileName` — its full path, `version` — the CAD system version.	optional
`toolbarLibrary`	A nested object: `name`, `path`, `version` — the name, path and version of the Add-in module.	optional

The `_generated` key is reserved: this section (`timeUTC` — the file creation time, `library` — the import library version, `processFileName` — the path of the process that produced the file) is appended by the library itself at `CloseFile`.

A writing example (the method is called from the import entry-point example) and the resulting JSON block:

```
// Header metadata: the source file, the CAD system, the Add-in module
void WriteHeaderMeta(CADDocument doc)
```

```
{
    var mw = sgr.Header.MetaWriter;
    mw.AddStrPair("originalFileName", doc.FileName);
    mw.AddStrPair("hostName", fHost); // the HostApplication tag from the Add-in
XML file
    mw.BeginObject("sourceApp");
        mw.AddStrPair("name", "mycad.exe");
        mw.AddStrPair("processFileName", cadExePath);
        mw.AddStrPair("version", cadVersion);
    mw.EndObject();
    mw.BeginObject("toolbarLibrary");
        mw.AddStrPair("name", "MyCADToolbar.dll");
        mw.AddStrPair("path", toolbarPath);
        mw.AddStrPair("version", toolbarVersion);
    mw.EndObject();
}
```

```
{
  "originalFileName": "C:\\Models\\part.mcad",
  "hostName": "MyCAD",
  "sourceApp":      { "name": "mycad.exe", "processFileName":
"C:\\MyCAD\\mycad.exe", "version": "3.2.1" },
  "toolbarLibrary": { "name": "MyCADToolbar.dll", "path":
"C:\\MyCAD\\Addins\\MyCADToolbar.dll", "version": "1.4.0.12" },
  "_generated":     { "timeUTC": "2026-07-15T10:30:00Z", "library": "STGeomFile
1.0.0.0", "processFileName": "C:\\MyCAD\\mycad.exe" }
}
```

Example: the import entry point

```
// The import entry point. Input: the current model of the CAD system and the path
to the output SGF file.
bool ExportToSGF(CADDocument doc, string outFilePath)
{
    if (!ConnectToGeomFiler(GetLibraryPath()))
        return false;

    bool ok = false;
    try
    {
        sgf.StartFile(outFilePath);
        try
        {
            // header metadata (see "SGF header metadata")
```

```
            WriteHeaderMeta(doc);

            // sgr.SetModelUnits(TST_LinearMeasure.lm_Metre, 1.0);
            // sgr.SetArcToler(0.005);
            // sgr.SetImportOption(TSTImportOption.ioCreateRectLoops, true);

            // UnitMatrix3D — the identity matrix (root without offset)
            if (!doc.IsDetail)
                SaveAssembly(doc.Assembly, UnitMatrix3D);
            else
                SavePart(doc.Part, UnitMatrix3D);
        }
        finally
        {
            sgf.CloseFile();
        }
        ok = true;
    }
    catch
    {
        ok = false;
    }
    finally
    {
        sgr = null;
        sgf = null;
        if (hDLL != IntPtr.Zero) { FreeLibrary(hDLL); hDLL = IntPtr.Zero; }
    }
    return ok;
}
```

SGF data structures

The base types and structures are declared in `STTypes.idl`. All the method signatures in the following sections are written in terms of them.

Type	Definition	Purpose
`STFloat`	`double`	Floating-point number: coordinates, radii, angles, parameters, weights.
`TST3DPoint`	`record (X, Y, Z: double)`	A point or vector in 3D. When used as a vector (`vZ`, `vX`, `Axis`, `ExtrusionVector`) it specifies a direction.
`TST2DPoint`	`record (X, Y: double)`	A point in 2D — in the parametric space of a surface (when building 2D loop curves).
`TST3DMatrix`	`record (vX, vY, vZ, vT: TST3DPoint; A, B, C, D: double)`	A transformation matrix / coordinate system: the basis vectors `vX`, `vY`, `vZ`, the origin `vT`, and the coefficients `A`–`D`.

The signatures also use primitives: `string` — a text identifier (`ID`) or a file path, `integer` — an integer, `boolean` — a flag.

CAD → SGF conversions in the examples. The CAD-system types (`CADPoint`, `CADMatrix`, ⋯) do not match the SGF types, so the examples use small conversion functions: `ToPoint` converts a CAD point to `TST3DPoint`, and `ToMatrix` converts a CAD matrix (coordinate system) to `TST3DMatrix`. This is a component-wise copy of the fields; the implementation is trivial and is not given in the examples — only the calls are shown (with explanatory comments).

Working with an assembly

An assembly (`CADAssembly`) is a node of the model tree with a placement matrix (`CADMatrix`). The `IsDetail` flag defines the node's contents: when `IsDetail = true`, it holds a part (`CADPart`) with geometry; when `IsDetail = false`, it holds nested assemblies. Only **`CADAssembly`** has a matrix, so any placed component — even a single part (a screw, a washer) — is wrapped in its own `CADAssembly` (which carries its position), and the geometry is stored inside, in a `CADPart`. The same `CADPart` can be reused by many assemblies (patterns, repeated fasteners). Importing an assembly is a recursive top-down traversal of the tree that accumulates the transform.

For example, this is how the "Roller in bracket" node looks in the CAD system's model tree:

```
Roller in bracket — CADAssembly
├─ Bracket — CADAssembly → CADPart
├─ Axle — CADAssembly → CADPart
├─ Roller — CADAssembly
│  ├─ Bushing — CADAssembly → CADPart
│  └─ Cage — CADAssembly → CADPart
├─ Screw M6 (1) — CADAssembly → CADPart
├─ Screw M6 (2) — CADAssembly → CADPart
├─ Screw M6 (3) — CADAssembly → CADPart
├─ Screw M6 (4) — CADAssembly → CADPart
├─ Washer 6 (1) — CADAssembly → CADPart
├─ Washer 6 (2) — CADAssembly → CADPart
├─ Washer 6 (3) — CADAssembly → CADPart
└─ Washer 6 (4) — CADAssembly → CADPart
```

Each tree node is a `CADAssembly` with its own matrix; the actual geometry appears only in the leaf `CADPart` nodes inside them. The "Roller" is a nested subassembly (`IsDetail = false`). "Screw M6" and "Washer 6" are present in four instances — these are four separate `CADAssembly` nodes (four matrices) that reference the same `CADPart` geometry.

> In the examples below, an assembly node (`CADAssembly`) has the following fields: `Name` — the name (a folder in the tree), `Transform` — the placement matrix (`CADMatrix`), `IsDetail` — the contents flag, and the contents themselves: when `IsDetail = true` — the part `Part` (`CADPart`), when `IsDetail = false` — the list of child nodes `Children`. In Variant 2, leaf nodes also use `ComponentId` — the identifier of the source component, shared by all identical parts (it is used to detect repeats). The exact set of fields depends on your CAD system.

What needs to be reproduced in SGF. Importing an assembly solves three tasks: reproduce the tree structure (folders), place each component using its matrix, and write the geometry of the parts (`ComboModel` / `ComboSolid`, see Working with a part). The tree structure is built by groups, and repeated

components can be reused through blocks. Below are both approaches; they are equivalent and are usually combined.

Group. Each assembly node is shown in the CAM system's model tree as a folder. The pair of methods `StartGroupEntity` / `CloseGroupEntity` opens and closes such a folder: all objects written between the calls belong to it. The calls are nested — exactly the way the nodes of the CAD assembly tree are nested.

Method	Description
`StartGroupEntity(EntityName: string)`	Begin a group — a folder in the model tree with the name `EntityName`. All objects built before the paired `CloseGroupEntity` go into this folder. The calls can be nested: nested folders are created inside a folder. It cannot be called inside `StartComboModel`, because that builds a separate closed stitched shell.
`CloseGroupEntity()`	Close the current group (folder).

Transformation matrix. A component's position is set before its geometry is saved. The matrix is set with the `SetCurrentTransform2` method, and all subsequent surfaces — until the next matrix change — are multiplied by it. During assembly traversal, the node's accumulated world transform (the product of the matrices from the root) is passed into the matrix, so each solid is given its final position.

The matrix is called before opening a solid — before `StartGroupEntity` or `StartComboModel`. It cannot be changed inside `StartComboModel`: the body is stitched in its own coordinates and placed as a whole according to the matrix set before it was opened (see Working with a part).

Method	Description
`SetCurrentTransform2(vT, vX, vY, vZ: TST3DPoint)`	Switches the transformation matrix for all subsequent objects; it is defined by a position (`vT` — the origin) and explicitly specified axes `vX`, `vY`, `vZ`. It must be called before saving a solid (before `StartGroupEntity` or `StartComboModel`) — all of its surfaces will be multiplied by this matrix. When saving assemblies, the final matrix is set for each solid. It cannot be called inside `StartComboModel`.

Method	Description
`SetCurrentTransform(vT, vZ, vX: TST3DPoint)`	Deprecated (but still used in some cases). The same as `SetCurrentTransform2`, but the coordinate system is defined by a position (`vT`), the Z axis (`vZ`), and the X axis (`vX`); the Y axis is computed as the cross product `vZ × vX` — which is ambiguous if `vZ` and `vX` are not strictly perpendicular. Where possible, use `SetCurrentTransform2`.

Variant 1 — standard traversal (groups). The simplest approach: each assembly node is wrapped in `StartGroupEntity` / `CloseGroupEntity`, and the part's geometry is written inside. The placement matrix is accumulated along the path from the root: each node's world transform = the parent's transform × the node's local matrix.

```
// Recursive traversal of the assembly. parentTransform is the parent's
accumulated transform.
void SaveAssembly(CADAssembly asm, TST3DMatrix parentTransform)
{
    // node's world transform = parent's transform × the assembly's local matrix
    TST3DMatrix world = Multiply(parentTransform, ToMatrix(asm.Transform));

    sgr.StartGroupEntity(asm.Name);
    try
    {
        if (asm.IsDetail)
            SavePart(asm.Part, world);                // leaf: a part with geometry
        else
            foreach (CADAssembly child in asm.Children)
                SaveAssembly(child, world);           // subassembly: descend into
child nodes
    }
    finally
    {
        sgr.CloseGroupEntity();
    }
}
```

This variant is suitable for most cases. Downside: if the same component appears in the assembly many times (for example, four identical bolts), its geometry is written to the file four times. Variant 2 is intended for such cases.

Variant 2 — blocks (Blocks). A block is a single model with topology (`ComboModel`) or without, written to SGF once under a unique name and inserted into the tree any number of times through `InsertBlock`. The geometry is stored once; each instance only specifies a placement matrix. For components that appear more than once, this reduces the file size and speeds up loading.

- subassembly (`IsDetail = false`) — an ordinary `StartGroupEntity` / `CloseGroupEntity` folder (as in Variant 1) with recursive traversal of the child nodes;
- part (`IsDetail = true`) — saved as a block: at the first occurrence of the component its geometry is written into a block (`StartBlock` / `CloseBlock`) and immediately placed with `InsertBlock`; at subsequent occurrences only `InsertBlock` is called. Already written blocks are tracked by the `savedBlocks` set (in the STEP importer — the `fShellSaved` flag); pre-counting the repeats is not needed.

`InsertBlock` itself creates a node in the model tree (its `NewBlockName` argument is the name of that node), so there is no need to wrap the node in an additional `StartGroupEntity`.

Block methods.

Method	Description
`StartBlock(BlockName: string)`	Begin a block — register, under the name `BlockName`, a single model with topology (`ComboModel`) or without, for later reuse. The geometry inside the block is written by the same methods as for a single part (see Working with a part). The block is placed in a buffer.
`CloseBlock()`	Finish building the block (it remains in the buffer).
`InsertBlock(BlockID, NewBlockName: string)`	Copy a block from the buffer into the model tree. `BlockID` is the block name from `StartBlock`; `NewBlockName` is the name of the node (group) being created in the tree. The placement matrix is set by a preceding call to `SetCurrentTransform2`.

```
// savedBlocks — the ComponentIds of blocks already written (an importer field;
analogous to fShellSaved in STEP).
readonly HashSet<string> savedBlocks = new HashSet<string>();

void SaveAssemblyWithBlocks(CADAssembly asm, TST3DMatrix parentTransform)
{
```

```
    TST3DMatrix world = Multiply(parentTransform, ToMatrix(asm.Transform));

    if (!asm.IsDetail)
    {
        sgr.StartGroupEntity(asm.Name);
        try
        {
            foreach (CADAssembly child in asm.Children)
                SaveAssemblyWithBlocks(child, world);
        }
        finally
        {
            sgr.CloseGroupEntity();
        }
        return;
    }

    // leaf (part): at the first occurrence of the component create a block,
afterwards just insert it
    if (savedBlocks.Add(asm.ComponentId))
    {
        sgr.StartBlock(asm.ComponentId);
        try
        {
            SavePart(asm.Part, UnitMatrix3D);
        }   // geometry in the part's local coordinates
        finally
        {
            sgr.CloseBlock();
        }
    }
    // place the instance by its matrix (including at the first occurrence)
    sgr.SetCurrentTransform2(world.vT, world.vX, world.vY, world.vZ);
    sgr.InsertBlock(asm.ComponentId, asm.Name);    // InsertBlock itself creates the
tree node
}
```

> Both variants are equivalent and can be combined in a single importer. Variant 1 is simpler to implement and debug; Variant 2 gives a smaller SGF file for assemblies with repeated parts and loads faster. In the example above, parts are formed as blocks (repeated geometry is written once and reused), while subassemblies and assembly nodes remain group folders.

Working with a part

A part (`CADPart`) consists of bodies (`CADBody`) and may also contain individual curves, points, and PMI. A body is a shell of faces. A closed body (a solid) is saved separately via `ComboSolid`, which provides stitching: the faces, vertices, and edges of the shell are written inside the `ComboModel` → `ComboSolid` pair, and on load the CAM system assembles them into a single solid. Open shells, as well as individual curves and points, do not go into `ComboSolid` — they are added to the model tree directly via `AddEntity` (see below).

Why stitch the model. The full topology of the body comes from the CAD model — surfaces, edges, and vertices together with the relationships between them. `ComboSolid` transfers these relationships to the SGF without loss: the CAM receives the same connected topology (B-rep) with well-defined "inside/outside" and face adjacency that existed in the CAD, and it does not need to find and guess the joints again.

Faces are added to the body after they have been created via `StartFace` (Face (trimmed surface)); the relationships along shared edges and vertices define the stitching.

Method	Description
`StartComboModel(GeomFile: string)`	Begin a model with topology (it will be "stitched"). `GeomFile` is the unique model name.
`StartComboSolid(ID: string)`	Begin a body inside the `ComboModel`.
`AddComboSolidVertex(p: TST3DPoint; VertexID: string)`	Add a body vertex `p` under the identifier `VertexID` (edges refer to it).
`AddComboSolidEdge(EdgeID, CurveID, StartVertexID, EndVertexID, LeftFaceID, RightFaceID: string)`	A body edge between vertices `StartVertexID`⋯ `EndVertexID`, separating the left (`LeftFaceID`) and right (`RightFaceID`) faces — it is exactly this relationship that defines the stitching. `CurveID` is usually better passed empty: a topological edge does not really need its own geometry — it is already localized by the joint of the left and right surfaces. If `CurveID` is passed, during stitching the engine will try to insert the geometry of the specified curve, and that curve does not always account for the fact that the edge may be split into several during stitching.

Method	Description
`AddComboSolidFace(FaceID, FaceName: string)`	Add a previously created (via `StartFace`) face.
`CloseComboSolid()`	Close the body.
`CloseComboModel()`	Close the model.

Unique identifiers and associativity. For topological elements — faces, edges, and vertices — it is critical to assign unique identifiers. Elements refer to each other by them (an edge refers to its two faces and two vertices, and so on), and this provides associativity: after stitching and subsequent recomputations, the CAM can unambiguously match an operation to a specific face rather than "losing" it. Therefore, the identifiers must be unique and stable.

The unique identifiers themselves are provided by the CAD system: every topological element in its model has its own persistent identifier that is preserved between rebuilds. The Translator does not need to generate identifiers itself — it is enough to take the element's native identifier from the CAD system's API and pass it into the SGF. This guarantees both uniqueness and stability: on re-importing a modified model, the unchanged elements will keep the same identifiers, and the operation bindings will not break.

Example: saving a part. The `SavePart` method, called while traversing the assembly (Working with an assembly), receives the part and its world matrix, sets the current transform, and saves its contents — bodies, individual curves, points, and PMI. It is convenient to wrap the part itself in a model tree folder (`StartGroupEntity`, Working with an assembly) — here this is an organizational folder, not stitching.

```
void SavePart(CADPart part, TST3DMatrix transform)
{
    sgr.SetCurrentTransform2(transform.vT, transform.vX,
transform.vY, transform.vZ);

    sgr.StartGroupEntity(part.Name);
    try
    {
        SaveBodies(part.Bodies);   // closed bodies — via ComboSolid, open ones —
via AddEntity
        SaveCurves(part.Curves);   // curves, see "Curves and points"
        SavePoints(part.Points);   // points, see "Curves and points"
        SavePMI(part.PMIs);        // PMI (threads, etc.), see "PMI"
    }
    finally
    {
        sgr.CloseGroupEntity();
    }
```

```
}

void SaveBodies(CADBody[] bodies)
{
    foreach (CADBody body in bodies)
    {
        if (body.IsClosed)
            SaveClosedShell(body);   // closed body → ComboSolid (stitching)
        else
            SaveOpenShell(body);     // open shell → AddEntity
    }
}

// An open shell is not a body: faces are added to the tree directly via AddEntity,
outside of ComboModel.
void SaveOpenShell(CADBody body)
{
    sgr.StartGroupEntity(body.Name);   // shell folder in the model tree
    try
    {
        foreach (CADFace face in body.Faces)
            if (SaveTrimmedFace(face))      // creates a face under face.Id
(see "Face")
                sgr.AddEntity(face.Id, face.Name);
    }
    finally
    {
        sgr.CloseGroupEntity();
    }
}

// A closed shell → one ComboSolid inside a ComboModel; the ComboSolid itself
provides the stitching.
void SaveClosedShell(CADBody body)
{
    sgr.StartComboModel(body.Id);
    try
    {
        sgr.StartComboSolid(body.Id);
        try
        {
            // 1) shell faces: save each one and add it to the body
            foreach (CADFace face in body.Faces)
                if (SaveTrimmedFace(face))
                    sgr.AddComboSolidFace(face.Id, face.Name);
```

```
            // 2) body vertices (edges refer to them by Id); convert coordinates
to TST3DPoint
            foreach (CADVertex vertex in body.Vertices)
                sgr.AddComboSolidVertex(ToPoint(vertex.Point), vertex.Id);

            // 3) body edges: which faces (left/right) meet at a shared edge — that
is the stitching
            foreach (CADEdge edge in body.Edges)
                sgr.AddComboSolidEdge(edge.Id, "",
                                      edge.Start.Id, edge.End.Id,
                                      edge.LeftFaceId, edge.RightFaceId);
        }
        finally
        {
            sgr.CloseComboSolid();
        }
    }
    finally
    {
        sgr.CloseComboModel();
    }
}
```

> `body.Vertices` and `body.Edges` are the body's topology: vertices with coordinates (`Point`) and identifiers (`Id`); each edge (`CADEdge`) stores its end vertices (`Start`, `End`) and the identifiers of the adjacent faces (`LeftFaceId`, `RightFaceId`). Vertices are added before edges — an edge refers to them by `Id`. An empty string is passed for `CurveID`: as in the STEP importer, the edge geometry is carried by the faces.

Open shells, curves, and points. Not everything in a part is a closed body. An open shell (`body.IsClosed == false`), individual 3D curves, and points do not go into `ComboSolid`: putting an open shell into a `ComboSolid` is pointless — there is nothing to stitch, no closed body will result — and it does not work with curves and points at all. Such objects are created outside `StartComboModel` and added to the model tree directly with a call to `AddEntity(ID, EntityName)` (Working with an assembly) — under the same `ID` the object was created with (see `SaveOpenShell` / `SaveCurves` / `SavePoints` in the example above). Placement, if needed, is defined by the `SetCurrentTransform2` matrix before `AddEntity`.

Method	Description
`AddEntity(ID: string; EntityName: string)`	Add a previously created entity to the model tree by its `ID` (a face, curve, or point) under the name `EntityName`. Its local coordinates are multiplied by the matrix set via `SetCurrentTransform2`. The

Method	Description
	method can also be called inside a `ComboModel` — the entity will go into the tree as a standalone object, outside the stitching (faces are added into the body's stitching by `AddComboSolidFace`).

Curves and points

Methods for building curve geometry: analytic primitives (point, line segment, circle, arc, ellipse, helix, NURBS, composite curves), derived curves (on-surface, trimmed, inverted, intersection) and their parametrization.

Color. The current color is set with the `SetCurrentColor(Value: integer)` call. This is a state: the specified color applies to all objects created after it — surfaces, curves and points (added through `AddEntity` or `AddComboSolidFace`) — and persists until it is changed by the next call. Therefore `SetCurrentColor` is called before each object that needs its own color. If `SetCurrentColor` was not called, the color is not overridden and the object inherits the color of the parent tree node. The value format is a packed integer `Blue shl 16 or Green shl 8 or Red` (8 bits per channel).

Example: saving curves and points.

```
void SaveCurves(CADCurve[] curves)
{
    foreach (CADCurve curve in curves)
        if (SaveCurve(curve))
        {
            sgr.SetCurrentColor(curve.Color);  // the color affects the next
AddEntity (and onward, until changed)
            sgr.AddEntity(curve.Id, curve.Name);
        }
}

void SavePoints(CADPoint[] points)
{
    foreach (CADPoint point in points)
    {
        sgr.CreatePoint(point.Id, ToPoint(point));   // convert the CAD point
coordinates to TST3DPoint
        sgr.AddEntity(point.Id, point.Name);         // SetCurrentColor was not
called — the point inherits the current/parent color
    }
}
```

The curve-building methods are grouped into three sections:

- Point and analytic curves — point, coordinate system, line segment, circle, arc, ellipse, helix, NURBS curve, composite curves.
- Derived curves — on-surface curve, trimmed, inverted, intersection curve and fillet boundary.
- Curve parametrization — changing the range and reparametrizing an already created curve.

Point and analytic curves

Curves are built in a local XY plane: `vT` is the center, `vZ` is the normal (the axis out of the plane), `vX` is the X axis of the local coordinate system, and the Y axis is computed as the cross product of Z and X (so it does not need to be passed). Building a curve starts from the `vX` direction (this is the start point of its parameterization) and proceeds counterclockwise around `vZ`; angles are measured in the same direction. The created objects can either be added to the geometry tree via `AddEntity` or used as building blocks for other objects — for example, as loop curves for trimming surfaces.

Point and analytic curve primitives:

- 3D point — `CreatePoint`.
- Coordinate system — `CreateCoordinateSystem`.
- Line segment — `CreateLineSeg`.
- Circle — `CreateCircle`.
- Circular arc — `CreateArc`.
- Ellipse — `CreateEllipse`.
- Elliptical arc — `CreateEllipseArc`.
- Conical spiral — `CreateConicalSpiral`.
- NURBS curve — `StartNurbsCurve` ⋯ `CloseNurbsCurve`.
- Composite curve — planar (2D) and spatial (3D).

Example of saving an analytic curve

`SaveCurve` is a dispatcher: it receives a CAD system curve and, depending on its type, calls the appropriate handler (`SaveLine`, `SaveCircle`, ⋯). Each handler writes the geometry to the SGF with the appropriate call from Curves and points and returns a success flag. Below are the dispatcher itself and one of the handlers (`SaveLine`); the rest are structured the same way.

```
// Saves a curve of any type; returns a success flag (Id is already set in CADCurve)
bool SaveCurve(CADCurve curve)
{
    switch (curve.Type)
    {
        case "Line":    return SaveLine(curve.Id, curve.Param);
        case "Circle":  return SaveCircle(curve.Id, curve.Param);      // handlers
for the other
        case "Arc":     return SaveArc(curve.Id, curve.Param);         // types are
structured the same way
        case "Ellipse": return SaveEllipse(curve.Id, curve.Param);
        case "Nurbs":   return SaveNurbsCurve(curve.Id, curve.Param);
        // … other curve types (see "Curves and points")
        default:        return false;                  // type not supported
```

```
    }
}

// One of the handlers — saving a line segment
bool SaveLine(string id, CADLineParam param)
{
    // convert the CAD point coordinates to TST3DPoint
    return sgr.CreateLineSeg(id, ToPoint(param.Start), ToPoint(param.End));
}
```

3D point

`CreatePoint(ID: string; P: TST3DPoint)` — create a 3D point.

- `ID` — identifier of the entity being created;
- `P` — coordinates of the point.

Coordinate system

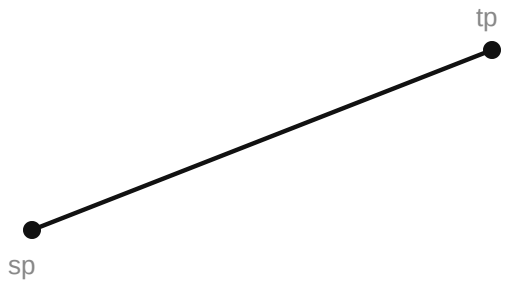
`CreateCoordinateSystem(CSID: string; vT, vZ, vX: TST3DPoint)` — create a coordinate system. It is defined in absolute coordinates.

- `CSID` — coordinate system identifier;
- `vT` — position (origin);
- `vZ` — Z axis;
- `vX` — X axis.

Line segment

`CreateLineSeg(ID: string; sp, tp: TST3DPoint)` — create a line segment from two points.

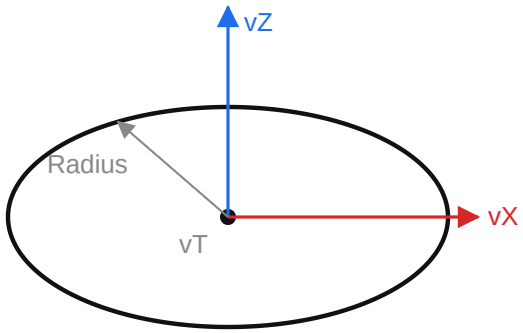
- `ID` — identifier of the entity;
- `sp` — start point;
- `tp` — end point.



Circle

`CreateCircle(ID: string; Radius: STFloat; vT, vZ, vX: TST3DPoint)` — create a circle from a point and a radius. Construction starts from `vX` and proceeds counterclockwise.

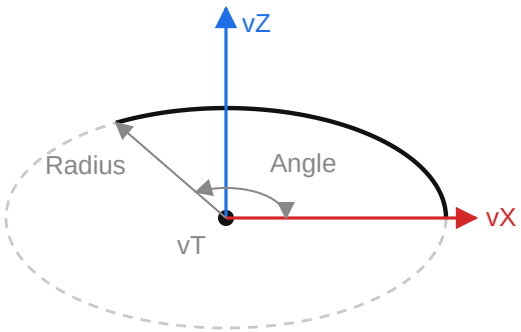
- `ID` — entity identifier;
- `Radius` — radius;
- `vT` — center point;
- `vZ` — Z axis;
- `vX` — X axis.



Circular arc

`CreateArc(ID: string; Radius, Angle: STFloat; vT, vZ, vX: TST3DPoint)` — create a circular arc. The circle is defined by a point and a radius. Construction starts from `vX` counterclockwise and sweeps through the angle `Angle`.

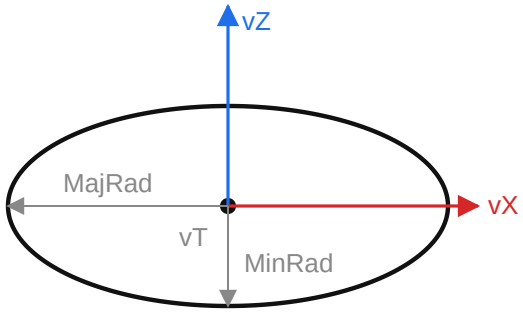
- `ID` — entity identifier;
- `Radius` — radius;
- `Angle` — arc sweep angle (measured from `vX`);
- `vT` — center point;
- `vZ` — Z axis;
- `vX` — X axis.



Ellipse

`CreateEllipse(ID: string; MajRad, MinRad: STFloat; vT, vZ, vX: TST3DPoint)` — create an ellipse.

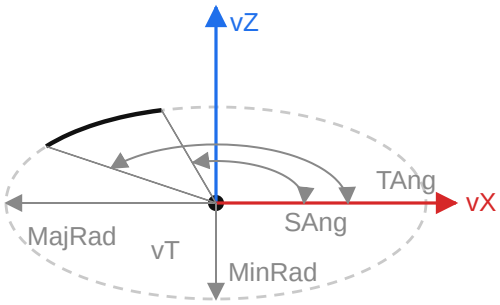
- `ID` — entity identifier;
- `MajRad` — major semi-axis (along the X axis);
- `MinRad` — minor semi-axis (along the Y axis);
- `vT` — center point;
- `vZ` — Z axis;
- `vX` — X axis.



Elliptical arc

`CreateEllipseArc(ID: string; MajRad, MinRad, SAng, TAng: STFloat; vT, vZ, vX: TST3DPoint)` — create an elliptical arc from two angles. All angles are measured from `vX` counterclockwise.

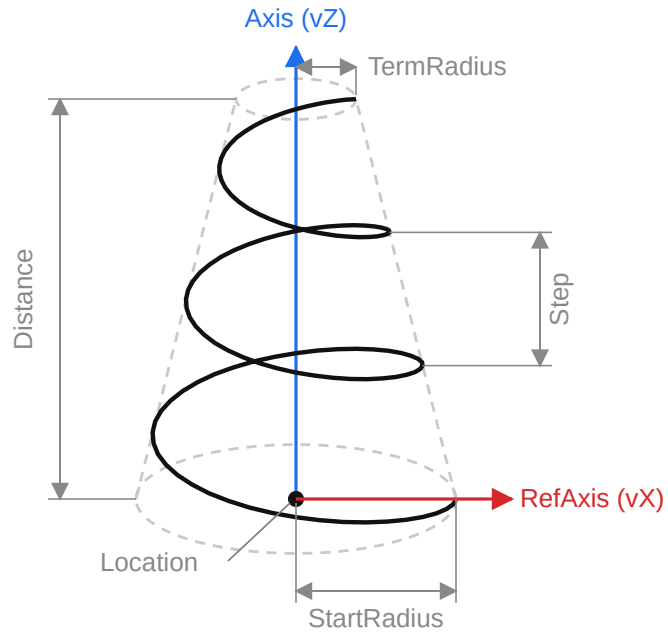
- `ID` — entity identifier;
- `MajRad` — major semi-axis (along the X axis);
- `MinRad` — minor semi-axis (along the Y axis);
- `SAng` — start angle of the arc;
- `TAng` — end angle of the arc;
- `vT` — center point;
- `vZ` — Z axis;
- `vX` — X axis.



Conical spiral

`CreateConicalSpiral(ID: string; Location, Axis, RefAxis: TST3DPoint; StartRadius, TermRadius, Step, Distance: STFloat; CCW: boolean)` — create a conical spiral around the `Axis` axis.

- `ID` — the entity identifier;
- `Location` — the center of the base (position, analogous to `vT`);
- `Axis` — the spiral axis (analogous to `vZ`);
- `RefAxis` — the reference direction (analogous to `vX`);
- `StartRadius` — the radius at the start;
- `TermRadius` — the radius at the end;
- `Step` — the step (rise per turn);
- `Distance` — the length along the axis;
- `CCW` — the direction of rotation (True = counterclockwise).



NURBS curve

A NURBS curve is formed through a sequence of calls: open (`StartNurbsCurve`), set the control points, weights, and knots, then close (`CloseNurbsCurve`).

`StartNurbsCurve(ID: string; Degree: integer; IsPeriodic, IsRational, IsClosed: boolean; KnotCount, CPCount: integer)` — begin forming a NURBS curve.

- `ID` — entity identifier;
- `Degree` — degree of the curve;
- `IsPeriodic` — periodicity flag;
- `IsRational` — rationality flag;
- `IsClosed` — closedness flag;
- `KnotCount` — number of knots;
- `CPCount` — number of control points.

`SetNurbsCurveControlPoint(Index: integer; P: TST3DPoint)` — set a control point by index.

- `Index` — index of the control point;
- `P` — coordinates of the point.

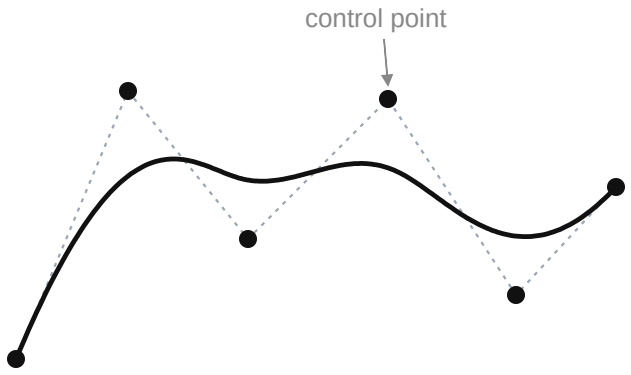
`SetNurbsCurveWeight(Index: integer; w: STFloat)` — set the weight of a control point.

- `Index` — index of the control point;
- `w` — weight.

`SetNurbsCurveKnot(Index: integer; knot: STFloat)` — set a knot.

- `Index` — index of the knot;
- `knot` — knot value.

`CloseNurbsCurve()` — finish forming the NURBS curve.



Example of importing a NURBS curve.

```
// Parameters of a CAD system NURBS curve
class CADNurbsParam
{
    public int         Degree;        // degree of the curve
    public bool        Periodic;      // periodicity
    public bool        Rational;      // rationality
    public bool        Closed;        // closedness
    public double[]    Knots;         // full knot vector
    public CADPoint[]  ControlPoints; // control points (CAD vertices)
    public double[]    Weights;       // weights (one per control point)
}

bool SaveNurbsCurve(string id, CADNurbsParam param)
{
    sgr.StartNurbsCurve(id, param.Degree, param.Periodic,
param.Rational, param.Closed,
                        param.Knots.Length, param.ControlPoints.Length);
    try
    {
        // knot vector
        for (int i = 0; i < param.Knots.Length; i++)
            sgr.SetNurbsCurveKnot(i, param.Knots[i]);

        // control points: weight + coordinates (cast to TST3DPoint)
        for (int i = 0; i < param.ControlPoints.Length; i++)
        {
            double w = param.Weights[i];
            TST3DPoint p = ToPoint(param.ControlPoints[i]);

            // for a rational curve, control points are given in homogeneous form
(× weight)
            if (param.Rational)
            {
                p.X *= w;
                p.Y *= w;
                p.Z *= w;
            }

            sgr.SetNurbsCurveWeight(i, w);
            sgr.SetNurbsCurveControlPoint(i, p);
        }
    }
    finally
    {
        sgr.CloseNurbsCurve();
    }
```

```
        return true;
    }
```

Composite curve

A composite curve is assembled through a sequence of calls: it is opened from a start point, extended with segments, and finished with a closing call. It can be planar (in a local plane) or spatial.

Planar (2D). Built in a local plane and extended with line segments, arcs, and Bezier curves.

`StartCurve2d(ID: string; p: TST2DPoint; vT, vZ, vX: TST3DPoint)` — start a 2D curve from point `p`.

- `ID` — entity identifier;
- `p` — start point (in plane coordinates);
- `vT` — position of the plane;
- `vZ` — normal;
- `vX` — X axis.

`CutTo2d(p: TST2DPoint)` — extend with a line segment to point `p`.

- `p` — end point of the segment.

`ArcTo2d(pc, p: TST2DPoint; rad: STFloat)` — extend with an arc to point `p`.

- `pc` — center of the arc;
- `p` — end point;
- `rad` — radius of the arc.

`BezierTo2d(p1, p2, p3: TST2DPoint)` — extend with a 3rd-order Bezier curve.

- `p1`, `p2`, `p3` — control points of the Bezier curve.

`CloseCurve2d(Close: boolean)` — finish the curve.

- `Close` — when `true`, the curve is closed.



Spatial (3D). Built in space and extended with line segments.

`StartCurve3d(ID: string; p: TST3DPoint)` — start a 3D curve from point `p`.

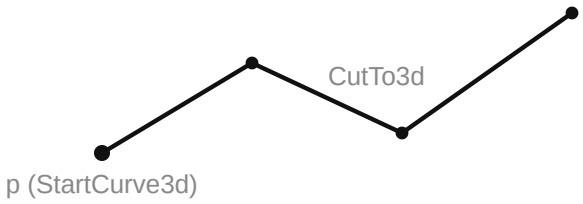
- `ID` — entity identifier;
- `p` — start point.

`CutTo3d(p: TST3DPoint)` — extend with a line segment to point `p`.

- `p` — end point of the segment.

`CloseCurve3d(IsClosed: boolean)` — finish the curve.

- `IsClosed` — when `true`, the curve is closed.



Derived curves

Derived curves are built on top of objects that already exist: the source curve (and, for a curve on a surface, the surface as well) is specified by its identifier, and the result receives a new `ID`. Such curves can be trimmed, inverted, reparameterized, or built on a surface. Like analytical curves, the result can be added to the tree via `AddEntity` or used as a building block (for example, a loop curve for trimming surfaces). When creating derived objects, the source objects are not changed, but if the same `ID` is used, the new object will overwrite the old one.

Types of derived curves:

- Curve on surface — `CreateCurveOnSurface`.
- Trimmed curve — `CreateTrimmedCurve` / `CreateTrimmedCurve2`.
- Inverted curve — `CreateInversedCurve`.
- Intersection curve — `StartIntersectionCurve` ⋯ `CloseIntersectionCurve`.
- Blend boundary — `CreateBlendBound`.

Example of saving a derived curve

To save a derived curve, you must first save all of its supporting objects and only then save the curve itself.

```
bool SaveTrimmedEdge(string edgeId, CADEdge edge)
{
    // first save the supporting curve
    if (!SaveCurve(edge.Curve))
        return false;

    // then save the trimmed curve itself, based on the supporting curve (by two
boundary points;
    // the CAD vertex coordinates are converted to TST3DPoint)
    return sgr.CreateTrimmedCurve2(edgeId, edge.Curve.Id,
                                   ToPoint(edge.Start.Point),
ToPoint(edge.End.Point));
}
```

Curve on a surface

`CreateCurveOnSurface(ID: string; CurveID, SurfaceID: string)` — create a 3D curve lying on a surface. The source curve is defined in the parametric space of the surface and projected onto it.

- `ID` — identifier of the entity being created;
- `CurveID` — identifier of the source curve;
- `SurfaceID` — identifier of the carrier surface.

Trimmed curve

A portion of the source curve between two boundaries. The boundaries are specified in one of two ways — by parameter values or by points:

`CreateTrimmedCurve(ID: string; SourceCurveID: string; t1, t2: STFloat)` — trim by parameter values.

- `ID` — identifier of the entity being created;
- `SourceCurveID` — identifier of the source curve;
- `t1`, `t2` — the start and end parameter values.

`CreateTrimmedCurve2(ID: string; SourceCurveID: string; p1, p2: TST3DPoint)` — trim by two points.

- `ID` — identifier of the entity being created;
- `SourceCurveID` — identifier of the source curve;
- `p1`, `p2` — the start and end points.

Inverted curve

`CreateInversedCurve(ID: string; SourceCurveID: string)` — create a curve with reversed (inverted) parameterization: the traversal direction of the source curve is reversed.

- `ID` — identifier of the entity to create;
- `SourceCurveID` — identifier of the source curve.

Intersection curve

An intersection curve is a spatial curve that lies simultaneously on two surfaces, along the line where they intersect. Both surfaces must be created beforehand and are passed by their identifiers.

`StartIntersectionCurve(ID: string; FirstSurfID, SecondSurfID: string; L1, L2: TSTLimitType; BaseParam, BaseScale: STFloat)` — begin building the intersection curve of two surfaces.

- `ID` — identifier of the entity being created;
- `FirstSurfID` — identifier of the first surface;
- `SecondSurfID` — identifier of the second surface;
- `L1` — type of the start boundary (see `TSTLimitType`);
- `L2` — type of the end boundary (see `TSTLimitType`);
- `BaseParam` — parameter of the first point lying on the intersection curve;
- `BaseScale` — parameterization scale.

`AddIntersectionCurvePoint(Position: TST3DPoint)` — add an intersection point.

- `Position` — coordinates of the point.

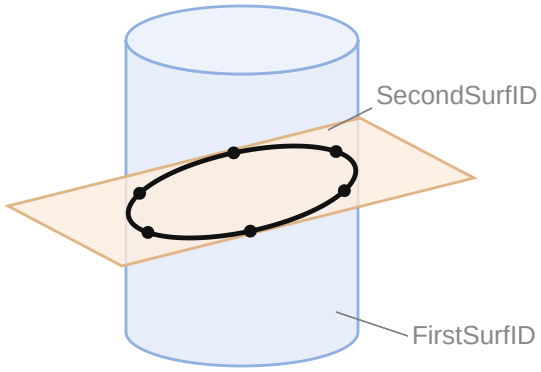
`CloseIntersectionCurve()` — finish building the intersection curve.

The `TSTLimitType` enumeration specifies the boundary type at the ends of the intersection curve (`L1` — start, `L2` — end) — that is, exactly how the curve is bounded on each side:

Value	Boundary type	Description
`H`	help vector	The end is defined by an auxiliary ("help") direction vector.
`T`	terminator	The curve terminates at an explicitly specified terminator point; for such a boundary, a boundary point is added to the curve with the `AddIntersectionCurvePoint` method.
`L`	arbitrary boundary	An arbitrary limit on the end of the curve.

Value	Boundary type	Description
`B`	spine boundary	The end is bounded by the edge of the spine region.
`U`	not defined	The boundary is not defined (no limit).

The same boundary types are also used when creating a blended surface (`CreateBlendedSurface`, see Blended surface).



Example of importing an intersection curve. It is assumed that both surfaces are already saved, and the curve points and boundary parameters have been obtained from CAD. The curve is written as a sequence of points; for boundaries of type `T` (terminators), explicit boundary points are additionally added.

```
// CAD system intersection curve: two supporting surfaces + points
class CADIntersectionCurve
{
    public string       Id;         // curve identifier in CAD
    public string       Surf1Id;    // identifier of the first surface
(already saved)
    public string       Surf2Id;    // identifier of the second surface
(already saved)
    public TSTLimitType StartLim;   // type of the start boundary
    public TSTLimitType EndLim;     // type of the end boundary
    public double       BaseParam;  // parameter of the first point on the curve
    public double       BaseScale;  // parameterization scale
    public CADPoint[]   Points;     // points of the intersection curve
    public CADPoint     StartPoint; // boundary point (for a boundary of type T)
    public CADPoint     EndPoint;   // boundary point (for a boundary of type T)
}

bool SaveIntersectionCurve(CADIntersectionCurve curve)
{
```

```
    sgr.StartIntersectionCurve(curve.Id, curve.Surf1Id, curve.Surf2Id,
                               curve.StartLim, curve.EndLim,
curve.BaseParam, curve.BaseScale);
    try
    {
        // convert CAD point coordinates to TST3DPoint
        if (curve.StartLim == TSTLimitType.T)
            sgr.AddIntersectionCurvePoint(ToPoint(curve.StartPoint));

        foreach (CADPoint v in curve.Points)
            sgr.AddIntersectionCurvePoint(ToPoint(v));

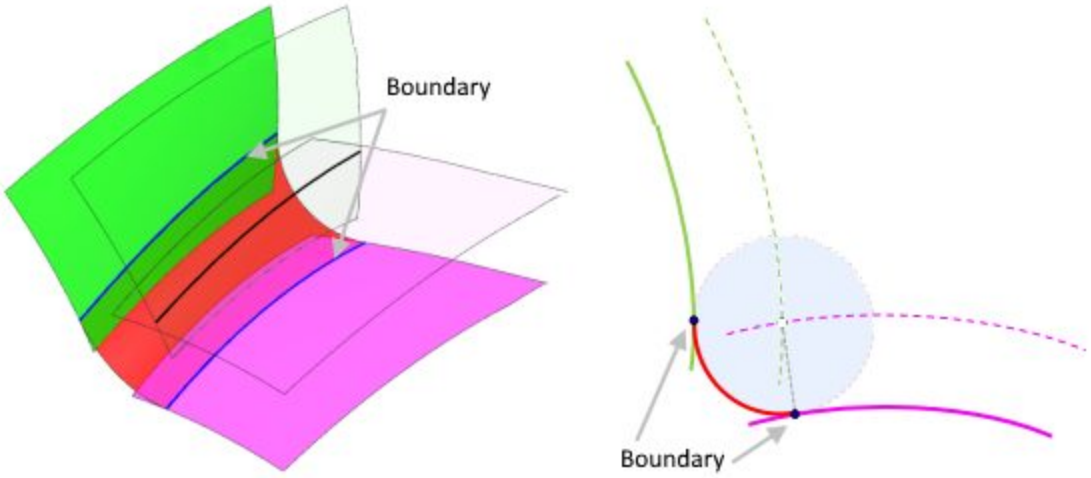
        if (curve.EndLim == TSTLimitType.T)
            sgr.AddIntersectionCurvePoint(ToPoint(curve.EndPoint));
    }
    finally
    {
        sgr.CloseIntersectionCurve();
    }
    return true;
}
```

Blend boundary

In the Parasolid XT format this is an auxiliary (construction) surface `blend_bound`, defined implicitly so that it intersects the corresponding supporting blend surface along the sought boundary line and is orthogonal both to the blend and to that surface along it. It has no parametrization of its own and in practice is represented by an intersection curve, so in our model it is created as a curve. The supporting surface to which the boundary corresponds is selected by the `Boundary` index — it is `surface[1 – Boundary]` from the pair of supporting surfaces of the `Blend`.

`CreateBlendBound(ID, Blend: string; Boundary: integer)` — blend boundary: the curve along which the blend surface (see Blend surface) is tangent to one of its supporting surfaces.

- `ID` — identifier of the entity being created;
- `Blend` — identifier of the blend surface (`ID` from `CreateBlendedSurface`, see Blend surface);
- `Boundary` — index of the supporting surface of the blend (`0` or `1`).



Curve parametrization

These methods change the parametrization of an already created curve (analytic or derived) by its identifier; no new object is created in the process.

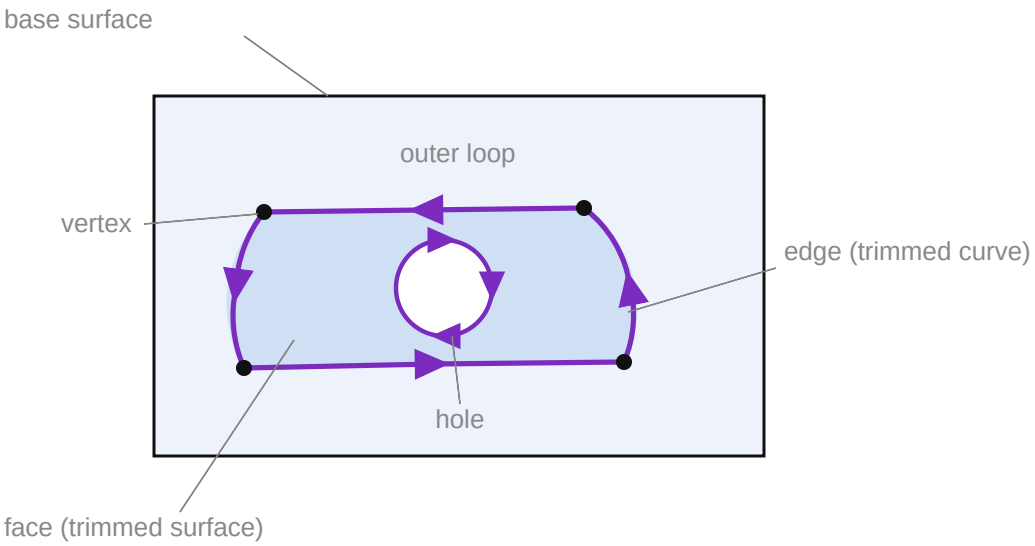
Method	Description
`SetCurveParametrization(CurveID: string; TMin, TMax: STFloat)`	Set new bounds for the parameter range of an existing curve. `CurveID` — curve identifier; `TMin`, `TMax` — new minimum and maximum parameter values.
`SetCurveDomain(ID: string; SrcTmin, SrcTmax, PrxTmin, PrxTmax: STFloat)`	Reparametrize the curve: define the correspondence between the old (`Src*`) and new (`Prx*`) parameter ranges. `ID` — curve identifier; `SrcTmin`, `SrcTmax` — old range bounds; `PrxTmin`, `PrxTmax` — new range bounds.

Face (trimmed surface)

A bounded region can be cut out of a surface — a face (trimmed surface) — and added to the geometry tree as a standalone object, or as part of a shell. A face consists of a *base surface* and one or more *contours (loops)* that bound the region being cut out. In this way a face links two kinds of geometry: the underlying surface (built with the methods in the Surfaces chapter) and the contour edge curves together with their vertices (the Curves and points chapter).

How it works.

1. Base surface. A face is built on top of a previously saved surface (any of the Surfaces). It is opened by calling `StartFace(ID, SurfaceID, NormalOrientation)`, where `SurfaceID` is the identifier of that surface, and `NormalOrientation` specifies whether the face normal coincides with the surface normal.
2. Contour (loop). This is a closed sequence of edges running along the boundary of the region being cut out; it is opened with `StartFaceLoop` and closed with `CloseFaceLoop`. There can be several contours: the first is typically the outer boundary, and the rest are holes inside the face.
3. Loop traversal direction. The region being kept (the face body) must lie to the left of the traversal direction (when looking against the face normal). Therefore the outer contour is traversed *counterclockwise*, and hole contours *clockwise*. The direction is set both by the order in which edges are added and by the `Orientation` flag of each edge (whether the traversal direction matches the direction of the supporting curve).
4. Edge = trimmed curve between two vertices. The ends of an edge — the *vertices* — are created as 3D points by calling `CreatePoint(ID, P)` (`AddFaceEdge3d` references them by identifier). The geometry of the edge itself is a trimmed curve: a supporting 3D curve (a line segment, arc, NURBS, intersection curve, etc., already saved in the SGF) is taken and trimmed between the endpoints of the edge by calling `CreateTrimmedCurve2(ID, SourceCurveID, p1, p2)` (for a straight edge you can use `CreateLineSeg` directly). If the edge is directed against the supporting curve, the curve is either reversed beforehand with `CreateInversedCurve` or `Orientation = false` is passed. The completed edge is added to the current loop: `AddFaceEdge3d` for a spatial curve, `AddFaceEdge2d` for a curve specified in the surface's UV parameters (see `CreateCurveOnSurface`).
5. Completion and adding to the tree. The face is closed with `CloseFace` (which returns a flag indicating whether "closure" succeeded). For the trimmed surface to appear in the geometry tree as a standalone object, it is registered by calling `AddEntity(ID, EntityName)` — where `ID` matches the face identifier from `StartFace`.



arrows: loop direction (face interior on the left)

`StartFace(ID, SurfaceID: string; NormalOrientation: boolean)` — begin a face based on a previously saved surface.

- `ID` — identifier of the face to create;
- `SurfaceID` — identifier of the supporting (base) surface;
- `NormalOrientation` — whether the face normal coincides with the base surface normal (`true` — coincides, `false` — opposite).

`StartFaceLoop()` — begin a face contour (loop).

`AddFaceEdge2d(CurveID: string; Orientation: boolean; svID, tvID: string)` — add an edge to the current contour using a curve specified in the surface's UV parameters (see Curve on surface).

- `CurveID` — identifier of the edge's supporting curve;
- `Orientation` — whether the contour traversal direction matches the curve direction (`false` — the curve is traversed in the opposite direction);
- `svID` — identifier of the edge's start vertex;
- `tvID` — identifier of the edge's end vertex.

`AddFaceEdge3d(CurveID: string; Orientation: boolean; svID, tvID: string)` — same as `AddFaceEdge2d`, but the edge is specified by a spatial (3D) curve; `svID` and `tvID` reference vertices — 3D points created by calling `CreatePoint`.

- `CurveID` — identifier of the edge's supporting 3D curve;
- `Orientation` — whether the contour traversal direction matches the curve direction;
- `svID` — identifier of the edge's start vertex;
- `tvID` — identifier of the edge's end vertex.

`CloseFaceLoop()` — finish the current contour.

`CloseFace()` — finish the face (returns a flag indicating whether closure succeeded).

Face color. The face color is set the same way as for other objects — by calling `SetCurrentColor` before `StartFace`; for how the current color behaves (state, inheritance, value format) see Curves and points.

Example: importing a CAD-model face into SGF. The `SaveTrimmedFace` method receives a `CADFace`, saves its underlying surface, then opens the face on top of it and traverses its contours.

```
bool SaveTrimmedFace(CADFace face)
{
    if (!SaveSurface(face.Surface))
        return false;

    // the face color is set before StartFace — by analogy with curves (see "Curves
and points")
    sgr.SetCurrentColor(face.Color);

    sgr.StartFace(face.Id, face.Surface.Id, face.SameSense);
    try
    {
        for (int i = 0; i < face.Loops.Length; i++)
        {
            CADLoop loop = face.Loops[i];
            sgr.StartFaceLoop();
            try
            {
                // the edge order sets the traversal direction:
                // outer contour — counterclockwise, holes — clockwise
                for (int j = 0; j < loop.Edges.Length; j++)
                {
                    // composite edge identifier: face.loop.edge
                    string edgeId = $"{face.Id}.{i}.{j}";
                    SaveFaceEdge(edgeId, loop.Edges[j]);
                }
            }
            finally
            {
                sgr.CloseFaceLoop();
            }
        }
    }
    finally
    {
        sgr.CloseFace();
```

```
    }
    return true;
}

// Saving a single contour edge
bool SaveFaceEdge(string edgeId, CADEdge edge)
{
    if (!SaveCurve(edge.Curve))
        return false;

    // edge geometry — the supporting curve, trimmed between the vertices
(coordinates converted to TST3DPoint)
    sgr.CreateTrimmedCurve2(edgeId, edge.Curve.Id,
                            ToPoint(edge.Start.Point), ToPoint(edge.End.Point));

    // add the edge to the loop (3D); SameSense — whether the traversal matches the
curve direction
    return sgr.AddFaceEdge3d(edgeId, edge.SameSense, edge.Start.Id, edge.End.Id);
}
```

The helper methods `SaveSurface(CADSurface)` and `SaveCurve(CADCurve)` select the call based on the geometry type (plane, cylinder, NURBS, line segment, arc, etc. — see Surfaces–Curves and points), write it to the SGF, and return a success flag; the identifier is taken from the object itself (the `Id` field).

Surfaces

A surface is the underlying geometry of a face: from the full (often infinite) surface, a face carves out a bounded region using its contours (see Face (trimmed surface)). This chapter collects the methods for building surfaces: analytic primitives (plane, cylinder, cone, sphere, torus), NURBS surface and polygonal mesh, derived surfaces (of revolution, extrusion, fillet, offset, and others), and their parametrization.

The surface-building methods are grouped into three sections:

- Analytic surfaces — plane, cylinder, cone, sphere, torus, NURBS surface, and polygonal mesh.
- Derived surfaces — surfaces of revolution, extrusion, transposed UV, fillet surface, and offset.
- Surface parametrization — changing the parametric domain of an already created surface (`SetSurfaceUVDomain`).

Analytic surfaces

Surfaces are created under a unique `ID`. For analytic primitives (plane, cylinder, cone, sphere, torus) the placement is specified in the same way by a triple: `Location` — position (`vT`), `Axis` — the Z axis (`vZ`), `RefAxis` — the X axis (`vX`); the Y axis is computed as the cross product `Axis × RefAxis`. Besides the analytic primitives, this group also includes the NURBS surface and the polygonal mesh (Mesh).

Some analytic surfaces are mathematically infinite — a plane is unbounded, and a cylinder and a cone are unbounded along their axis. For that reason they must be bounded when created: for a plane the region is given by a rectangle `xMin…xMax` × `yMin…yMax`, for a cylinder and a cone by axial bounds `Hmin…Hmax` (and, if needed, an angular sector `Amin…Amax`). If the CAD system does not provide explicit bounds, it is reasonable to bound the surface by the extents of the whole model. (A sphere and a torus are closed and do not need any additional bounding.)

Analytic surface primitives:

- Plane — `CreatePlaneP`.
- Cylinder — `CreateCylinderP`.
- Cone — `CreateConeP`.
- Sphere — `CreateSphereP`.
- Torus — `CreateTorusP`.
- NURBS surface — `StartNurbsSurface` ⋯ `CloseNurbsSurface`.
- Polygonal mesh — `StartMesh` ⋯ `CloseMesh`.

Example of saving an analytic surface

`SaveSurface` is a dispatcher: it receives a CAD-system surface and, depending on its type, calls the corresponding handler (`SavePlane`, `SaveCylinder`, ⋯). Each handler writes the geometry to SGF with the appropriate call from Surfaces and returns a success flag. Below are the dispatcher itself and one of the handlers (`SavePlane`); the others are organized in the same way.

```
// Saves a surface of any type; returns a success flag (Id is already set
in CADSurface)
bool SaveSurface(CADSurface surf)
{
    switch (surf.Type)
    {
        case "Plane":    return SavePlane(surf.Id, surf.Param);
        case "Cylinder": return SaveCylinder(surf.Id, surf.Param);    // handlers
for the other
        case "Cone":     return SaveCone(surf.Id, surf.Param);        // types are
organized the same way
        case "Sphere":   return SaveSphere(surf.Id, surf.Param);
```

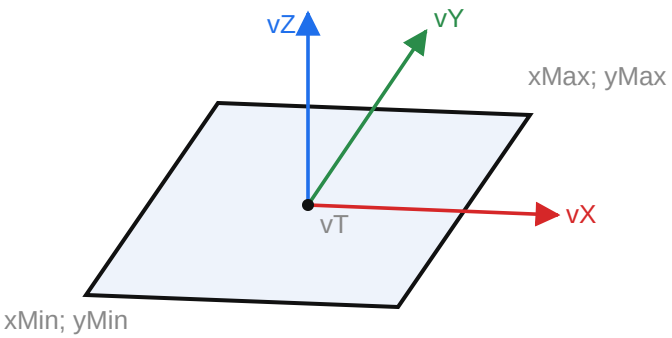
```
        case "Torus":    return SaveTorus(surf.Id, surf.Param);
        case "Nurbs":    return SaveNurbsSurface(surf.Id, surf.Param);
        case "Mesh":     return SaveMesh(surf.Id, surf.Param);
        // … other surface types (see "Surfaces")
        default:         return false;                  // type not supported
    }
}

// One of the handlers — saving a plane
bool SavePlane(string id, CADPlaneParam param)
{
    // convert the CAD direction points to TST3DPoint (Xmin/Ymin/Xmax/Ymax
are numbers)
    return sgr.CreatePlaneP(id, ToPoint(param.Location), ToPoint(param.Axis),
ToPoint(param.RefAxis),
                            param.Xmin, param.Ymin, param.Xmax, param.Ymax);
}
```

Plane

`CreatePlaneP(ID: string; Location, Axis, RefAxis: TST3DPoint; xMin, yMin, xMax, yMax: STFloat)` — create a plane bounded by a rectangular region.

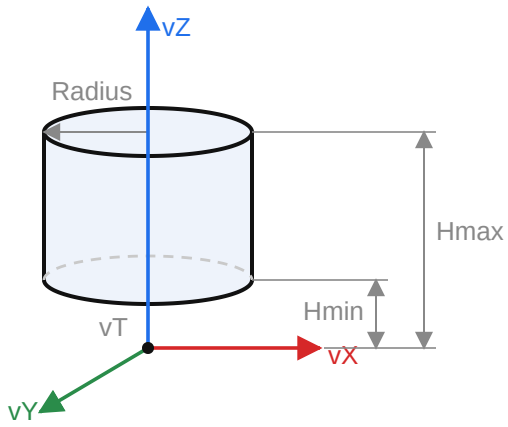
- `ID` — the entity identifier;
- `Location` — position (`vT`);
- `Axis` — the Z axis (`vZ`);
- `RefAxis` — the X axis (`vX`);
- `xMin`, `yMin` — the minimum X and Y of the bounding rectangle;
- `xMax`, `yMax` — the maximum X and Y of the bounding rectangle.



Cylinder

`CreateCylinderP(ID: string; Location, Axis, RefAxis: TST3DPoint; Radius, Hmin, Hmax, Amin, Amax: STFloat)` — create a cylindrical surface. The axis runs along the `Axis` vector.

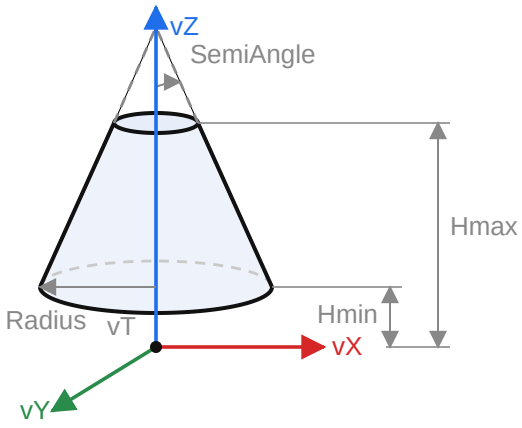
- `ID` — entity identifier;
- `Location` — position (`vT`);
- `Axis` — Z axis (`vZ`);
- `RefAxis` — X axis (`vX`);
- `Radius` — radius;
- `Hmin`, `Hmax` — height bounds (along the axis);
- `Amin`, `Amax` — angular sector bounds (a full cylinder runs from `0` to `2π`).



Cone

`CreateConeP(ID: string; Location, Axis, RefAxis: TST3DPoint; Radius, SemiAngle, Hmin, Hmax, Amin, Amax: STFloat)` — create a conical surface.

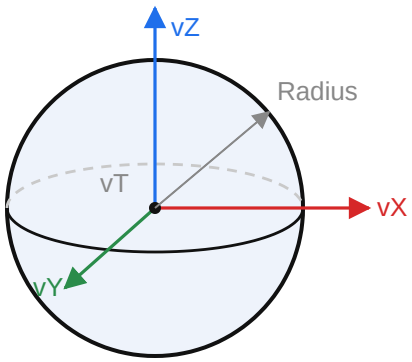
- `ID` — entity identifier;
- `Location` — position (`vT`);
- `Axis` — Z axis (`vZ`);
- `RefAxis` — X axis (`vX`);
- `Radius` — radius (in the `Location` plane);
- `SemiAngle` — half-angle at the cone apex;
- `Hmin`, `Hmax` — height bounds (along the axis);
- `Amin`, `Amax` — angular sector bounds (full cone — from `0` to `2π`).



Sphere

`CreateSphereP(ID: string; Location, Axis, RefAxis: TST3DPoint; Radius: STFloat)` — create a spherical surface.

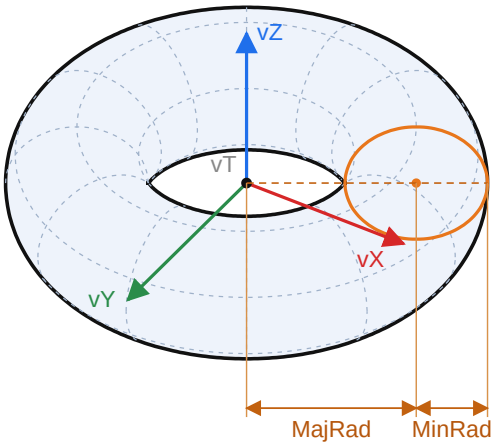
- `ID` — the entity identifier;
- `Location` — the center (`vT`);
- `Axis` — the Z axis (`vZ`);
- `RefAxis` — the X axis (`vX`);
- `Radius` — the radius.



Torus

`CreateTorusP(ID: string; Location, Axis, RefAxis: TST3DPoint; MajRad, MinRad: STFloat)` — create a toroidal surface.

- `ID` — identifier of the entity;
- `Location` — center (`vT`);
- `Axis` — Z axis (`vZ`), the axis of the torus;
- `RefAxis` — X axis (`vX`);
- `MajRad` — major radius (from the axis to the center of the tube);
- `MinRad` — minor radius (radius of the tube).



NURBS surface

A NURBS surface is formed by a sequence of calls: open (`StartNurbsSurface`), set the knots along U and V, the control points and weights in an `i×j` grid, close (`CloseNurbsSurface` or `CloseNurbsSurfaceB`).

`StartNurbsSurface(ID: string; K1, K2, UDegree, VDegree: integer; UClosed, VClosed, UPeriodic, VPeriodic, Rational: boolean)` — begin forming a NURBS surface.

- `ID` — entity identifier;
- `K1` — index of the number of control points along U: `K1 = UKnotCount − UDegree − 2`;
- `K2` — index of the number of control points along V: `K2 = VKnotCount − VDegree − 2`;
- `UDegree`, `VDegree` — degrees along U and V;
- `UClosed`, `VClosed` — closedness flags along U and V;
- `UPeriodic`, `VPeriodic` — periodicity flags along U and V;
- `Rational` — rationality flag.

`SetNurbsSurfaceUKnot(i: integer; Value: STFloat)` — set a U knot.

- `i` — knot number;
- `Value` — knot value.

`SetNurbsSurfaceVKnot(i: integer; Value: STFloat)` — set a V knot.

- `i` — knot number;
- `Value` — knot value.

`SetNurbsSurfaceControlPoint(i, j: integer; P: TST3DPoint)` — set a control point in the grid.

- `i`, `j` — position of the control point in the grid;
- `P` — point coordinates.

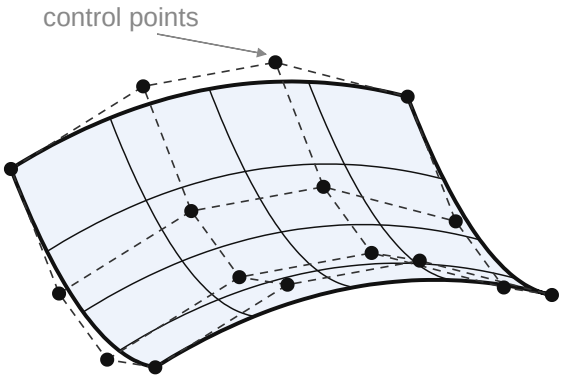
`SetNurbsSurfaceWeight(i, j: integer; Value: STFloat)` — set the weight of a control point.

- `i`, `j` — position of the control point in the grid;
- `Value` — weight.

`CloseNurbsSurface()` — finish forming the surface.

`CloseNurbsSurfaceB(UMin, UMax, VMin, VMax: STFloat)` — finish forming with explicit parameter ranges.

- `UMin`, `UMax` — bounds of the U parameter;
- `VMin`, `VMax` — bounds of the V parameter.



Example of importing a NURBS surface.

```
// NURBS surface parameters from the CAD system
class CADNurbsSurfParam
{
    public int         UDegree, VDegree;      // degrees along U and V
    public bool        UClosed, VClosed;      // closedness along U and V
    public bool        UPeriodic, VPeriodic;  // periodicity along U and V
    public bool        Rational;              // rationality
    public double[]    UKnots, VKnots;        // full knot vectors along U and V
    public CADPoint[,] ControlPoints;         // control point grid
    public double[,]   Weights;               // weights (one per control point)
}

bool SaveNurbsSurface(string id, CADNurbsSurfParam param)
{
    // control point count indices: K = KnotCount - Degree - 2
    int k1 = param.UKnots.Length - param.UDegree - 2;
    int k2 = param.VKnots.Length - param.VDegree - 2;

    sgr.StartNurbsSurface(id, k1, k2, param.UDegree, param.VDegree,
                          param.UClosed, param.VClosed, param.UPeriodic,
param.VPeriodic, param.Rational);
    try
    {
        // knot vectors along U and V
        for (int i = 0; i < param.UKnots.Length; i++)
            sgr.SetNurbsSurfaceUKnot(i, param.UKnots[i]);
        for (int i = 0; i < param.VKnots.Length; i++)
            sgr.SetNurbsSurfaceVKnot(i, param.VKnots[i]);

        // control points and weights in the (k1+1) × (k2+1) grid
        for (int i = 0; i <= k1; i++)
```

```
            for (int j = 0; j <= k2; j++)
            {
                double w = param.Weights[i, j];
                TST3DPoint p = ToPoint(param.ControlPoints[i, j]);

                // for a rational surface — in homogeneous form (× weight)
                if (param.Rational)
                {
                    p.X *= w;
                    p.Y *= w;
                    p.Z *= w;
                }

                sgr.SetNurbsSurfaceWeight(i, j, w);
                sgr.SetNurbsSurfaceControlPoint(i, j, p);
            }
    }
    finally
    {
        sgr.CloseNurbsSurface();
    }
    return true;
}
```

Polygonal mesh

A polygonal mesh is formed by a set of triangles, each consisting of three vertices.

`StartMesh(ID: string)` — begin building the mesh.

- `ID` — entity identifier.

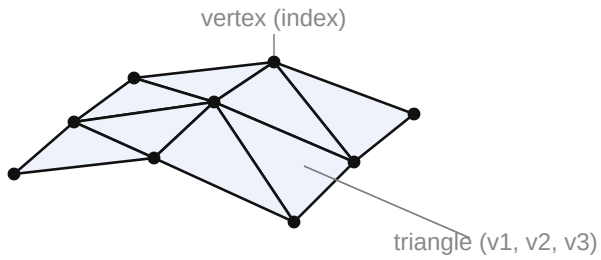
`AddMeshVertex(index: integer; p: TST3DPoint)` — add a vertex.

- `index` — vertex index (used in `AddMeshTriangle`);
- `p` — vertex coordinates.

`AddMeshTriangle(v1, v2, v3: integer)` — add a triangle.

- `v1`, `v2`, `v3` — the indices of the three vertices.

`CloseMesh()` — finish building the mesh.



Example of importing a polygonal mesh. Each vertex is added once — by its index (repeated indices from different triangles are skipped), then triangles are added by these indices.

```
// Parameters of the CAD system's polygonal mesh
class CADMeshParam
{
    public CADPoint[] Vertices;  // vertices (index = position in the array)
    public int[][]    Triangles; // triangles: triples of vertex indices
}

bool SaveMesh(string id, CADMeshParam meshParam)
{
    sgr.StartMesh(id);
    try
    {
        var added = new HashSet<int>();
        foreach (int[] tri in meshParam.Triangles)
        {
            // add each vertex once — by its index (convert to TST3DPoint)
```

```
            foreach (int vIndex in tri)
                if (added.Add(vIndex))   // convert the vertex to TST3DPoint
                    sgr.AddMeshVertex(vIndex, ToPoint(meshParam.Vertices[vIndex]));

            sgr.AddMeshTriangle(tri[0], tri[1], tri[2]);
        }
    }
    finally
    {
        sgr.CloseMesh();
    }
    return true;
}
```

Derived surfaces

Derived surfaces are built on the basis of objects that have already been created — curves (generatrices, spines) and other surfaces: the source objects are specified by their identifiers, and the result receives a new `ID`.

Types of derived surfaces:

- Surface of revolution — `CreateRevSurface120`.
- Extrusion surface — `CreateTabSurface122`.
- Transposed UV surface — `CreateTransposedUVSurface`.
- Blended surface — `CreateBlendedSurface`.
- Offset surface — `CreateOffsetSurf`.

Example of saving a derived surface

To save a derived surface, you must first save all the supporting objects, and only then the surface itself.

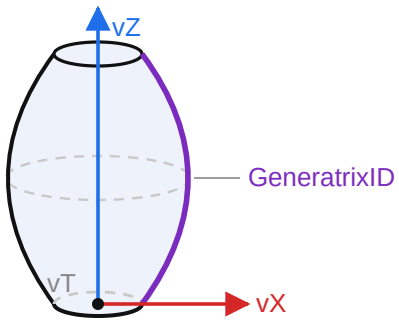
```
bool SaveOffsetSurface(string id, CADOffsetParam param)
{
    if (!SaveSurface(param.Surface))
        return false;

    return sgr.CreateOffsetSurf(id, param.Surface.Id, param.Offset);
}
```

Surface of revolution

`CreateRevSurface120(ID: string; Location, Axis: TST3DPoint; GeneratrixID: string; sa, ta: STFloat)` — a surface of revolution of a generatrix curve around an axis. It is built starting from the X axis, counterclockwise around `Axis`: from the start angle `sa` to the end angle `ta`.

- `ID` — identifier of the entity to create;
- `Location` — a point on the axis of revolution (`vT`);
- `Axis` — direction of the axis of revolution (`vZ`);
- `GeneratrixID` — identifier of the generatrix curve;
- `sa` — start angle (usually `0`);
- `ta` — end angle (full revolution — `2π`).



Example of saving a surface of revolution.

```
// Surface of revolution parameters from the CAD system
class CADRevolutionParam
{
    public CADPoint Location;    // point on the axis of revolution (vT)
    public CADPoint Axis;        // direction of the axis of revolution (vZ)
    public CADCurve Curve;       // generatrix curve
    public double   StartAngle;  // start angle (sa)
    public double   EndAngle;    // end angle (ta); full revolution — 2π
}

bool SaveRevolution(string id, CADRevolutionParam param)
{
    if (!SaveCurve(param.Curve))
        return false;

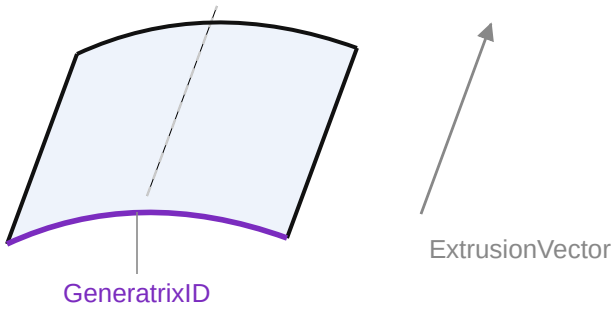
    return sgr.CreateRevSurface120(id, ToPoint(param.Location), ToPoint(param.Axis),
                                   param.Curve.Id, param.StartAngle,
```

```
param.EndAngle);
}
```

Extrusion surface

`CreateTabSurface122(ID: string; GeneratrixID: string; ExtrusionVector: TST3DPoint)` — a ruled extrusion surface: the generatrix curve is swept along a vector.

- `ID` — identifier of the entity to create;
- `GeneratrixID` — identifier of the generatrix curve;
- `ExtrusionVector` — extrusion vector (direction and length).



Example of saving an extrusion surface.

```
// Extrusion surface parameters from the CAD system
class CADExtrusionParam
{
    public CADCurve Curve;            // generatrix curve
    public CADPoint ExtrusionVector;  // extrusion vector (direction and length)
}

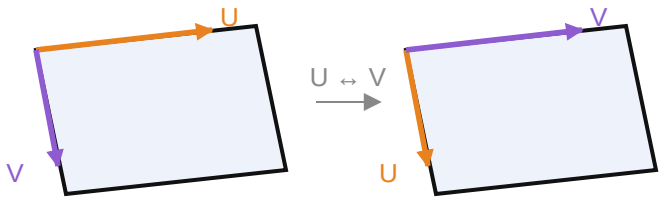
bool SaveExtrusion(string id, CADExtrusionParam param)
{
    if (!SaveCurve(param.Curve))
        return false;

    return sgr.CreateTabSurface122(id, param.Curve.Id,
ToPoint(param.ExtrusionVector));
}
```

Transposed UV surface

`CreateTransposedUVSurface(ID: string; SourceSurface: string)` — a surface with the U and V parameters swapped.

- `ID` — the identifier of the entity being created;
- `SourceSurface` — the identifier of the source surface.



Blend surface

An exact rolling-ball blend surface between two surfaces. Corresponds to the `blended_edge` surface of the Parasolid XT format and usually arises when blending an edge between two faces.

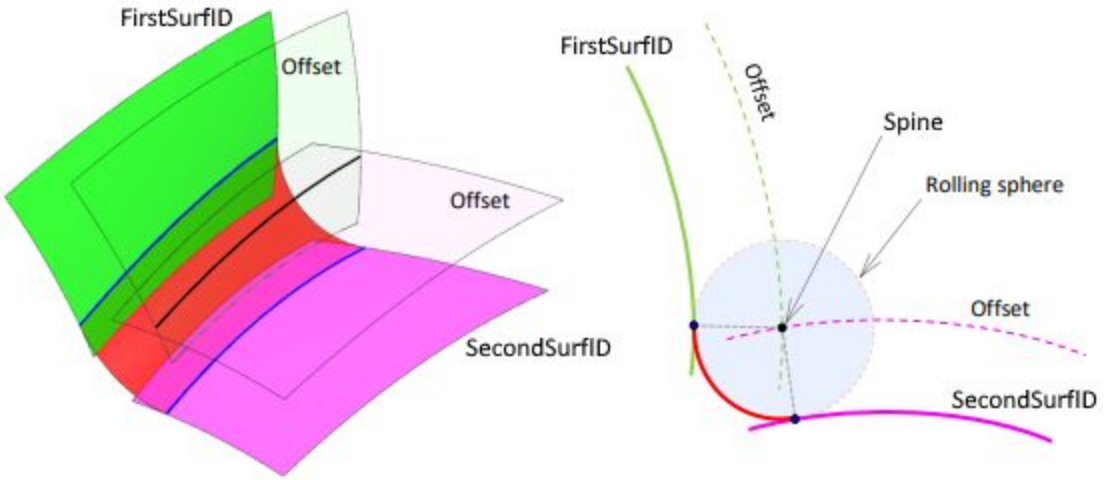
The spine is built as the intersection curve of two offset surfaces — the supporting surfaces displaced by the amounts `Range1` and `Range2`. The offset is measured along the unit surface normal multiplied by `Range` (and by −1 if the surface orientation is negative); `Range` values may also be negative. For a constant-radius blend, both values equal the ball radius.

> Where possible, blends are modeled by simpler surfaces — cylindrical, toroidal, or spherical — and complex ones (variable radius, non-circular cross-section) by NURBS surfaces. A dedicated blend surface (`blended_edge`) is used only when a simpler representation is not possible.

`CreateBlendedSurface(ID, FirstSurfID, SecondSurfID, Spine: string; Range1, Range2: STFloat; BlendedType: TSTBlendedType; p1: TST3DPoint; L1: TSTLimitType; p2: TST3DPoint; L2: TSTLimitType)` — create a blend surface.

- `ID` — identifier of the entity being created;
- `FirstSurfID`, `SecondSurfID` — the two supporting surfaces (adjacent to the original edge) between which the blend is built;
- `Spine` — spine: the axial curve of the blend, the path of the center of the rolling ball; built as the intersection curve of two offset surfaces;
- `Range1`, `Range2` — the offsets applied to the first and second surfaces respectively; in the figure they define the "Offset" offset surfaces (dashed), whose intersection gives the spine; for a constant radius they equal the ball radius;
- `BlendedType` — blend type (`TSTBlendedType`): `R` — rolling ball; the ball in the figure is labeled "Rolling sphere", its radius equals `Range`, and its center rolls along the spine; `E` — "cliff edge", when one of the surfaces is itself a blend with range `[0, 0]`, and the spine coincides with the cliff edge;
- `p1`, `L1` — start point and its boundary type (`TSTLimitType`, see Intersection curve);
- `p2`, `L2` — end point and its boundary type.

The fields `p1`/`L1` and `p2`/`L2` are specified (boundary type `L`) only in degenerate cases — when the spine is periodic but the blended edge has ends (for example, the spine is elliptical and the blend degenerates). In the ordinary case the boundaries are unbounded (type `U`).



`TSTBlendedType` and `TSTLimitType` are SGF types, not CAD system types. Therefore the parameter class uses the add-in's own enumerations, and before calling `CreateBlendedSurface` they are cast to SGF types (just as points are cast to `TST3DPoint`).

Example of saving a blend surface.

```
// the add-in's own enumerations (CAD side, without SGF types)
enum CADBlendKind  { RollingBall, Cliff }              // corresponds to R / E
enum CADBorderKind { NotSet, Terminator, ArbLimit }    // corresponds to U / T / L

// Parameters of a CAD system blend surface
class CADBlendParam
{
    public CADSurface    Surface1, Surface2;  // the two supporting surfaces
    public CADCurve      Spine;               // spine (axial curve of the blend)
    public double        Range1, Range2;      // offsets of the offset surfaces (for
a constant radius = ball radius)
    public CADBlendKind  Kind;                // blend type (CAD side)
    public CADPoint      P1, P2;              // boundary points (usually not used)
    public CADBorderKind L1, L2;              // boundary types (NotSet in the
ordinary case)
}

bool SaveBlendedSurface(string id, CADBlendParam param)
{
    // first save the supporting objects: both surfaces and the spine
    if (!SaveSurface(param.Surface1) || !SaveSurface(param.Surface2)
|| !SaveCurve(param.Spine))
        return false;

    // cast the CAD enumerations to SGF types via switch
```

```
    TSTBlendedType blendedType;
    switch (param.Kind)
    {
        case CADBlendKind.Cliff: blendedType = TSTBlendedType.E; break;   //
"cliff edge"
        default:                 blendedType = TSTBlendedType.R; break;   //
rolling ball
    }

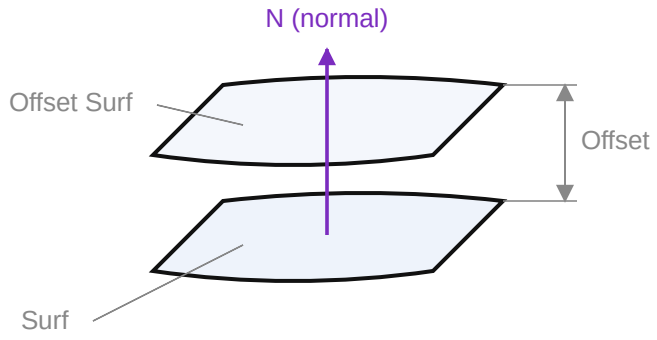
    // blend surface (points are cast to TST3DPoint)
    return sgr.CreateBlendedSurface(id, param.Surface1.Id,
param.Surface2.Id, param.Spine.Id,
                                    param.Range1, param.Range2, blendedType,
                                    ToPoint(param.P1), ToLimit(param.L1),
                                    ToPoint(param.P2), ToLimit(param.L2));
}

// boundary-type cast function (used for L1 and L2)
TSTLimitType ToLimit(CADBorderKind k)
{
    switch (k)
    {
        case CADBorderKind.Terminator: return TSTLimitType.T;
        case CADBorderKind.ArbLimit:   return TSTLimitType.L;
        default:                       return TSTLimitType.U;   // not set
    }
}
```

Offset surface

`CreateOffsetSurf(ID, Surf: string; Offset: STFloat)` — create an offset surface. It is built by shifting every point of the source surface `Surf` along its unit normal by the distance `Offset`. The direction of the offset is determined by the sign of `Offset` and the sense (orientation) of the source surface.

- `ID` — identifier of the entity being created;
- `Surf` — identifier of the source surface;
- `Offset` — signed offset distance along the normal (`> 0` — along the normal `N`, `< 0` — against it).



Surface parametrization

`SetSurfaceUVDomain(ID: string; SrcUmin, SrcUmax, SrcVmin, SrcVmax, PrxUmin, PrxUmax, PrxVmin, PrxVmax: STFloat; TransposeUV: boolean)` — change the parametrization of an already created surface by its identifier. No new object is created: the surface is wrapped in a proxy that defines a linear remapping between two U/V parameter ranges.

- `ID` — surface identifier;
- `SrcUmin`, `SrcUmax` / `SrcVmin`, `SrcVmax` — the Src range: the parametric space in which the surface is "seen" from the outside — the one in which the loop coordinates of the face referencing this surface are given;
- `PrxUmin`, `PrxUmax` / `PrxVmin`, `PrxVmax` — the Prx range: the space into which the external parameter is remapped for evaluating the surface itself (for NURBS — the knot range, *knot-span*);
- `TransposeUV` — when `true`, the U and V axes are swapped (for example, when the U/V order in the CAD system does not match the one used in SGF — as happens for a torus).

Why this is needed. Some CAD systems store the loop of a face in one parametric space, while defining the underlying surface in another. If they are not reconciled, the CAM system will project the loop to the wrong place, and the face will be built incorrectly. `SetSurfaceUVDomain` defines the linear correspondence Src↔Prx, and the loop is projected correctly.

A special case is surface extension: when the parametric domain of the face is wider than the natural domain of the surface. This is what import from KOMPAS does for NURBS faces: the actual range of the face (`MathSurface.ParamU/V`) extends beyond the NURBS knot range. The proxy "stretches" the surface over the entire domain of the face, and the face loop is projected onto the NURBS correctly. But if the face domain lies entirely within the knot range (ordinary "trimming inside a NURBS", a typical fillet) — reparametrization is not needed, otherwise it would only distort the geometry.

Example: aligning the parametrization of a surface.

```
// Align the parametrization of a face and its underlying NURBS surface (following
the KOMPAS import pattern).
void AlignSurfaceUVDomain(string surfaceId, CADFace face)
{
    CADSurface surf = face.Surface;

    // Src — the UV space in which the face loops are given
    double srcUmin = face.ParamUMin, srcUmax = face.ParamUMax;
    double srcVmin = face.ParamVMin, srcVmax = face.ParamVMax;

    // Prx — the natural domain of the surface (for NURBS — the knot range)
    double prxUmin = surf.KnotUMin, prxUmax = surf.KnotUMax;
    double prxVmin = surf.KnotVMin, prxVmax = surf.KnotVMax;
```

```
    // If the face domain lies entirely within the surface domain, this is ordinary
    // "trimming inside a NURBS", and reparametrization would only distort
the geometry.
    const double eps = 1e-9;
    bool insideU = srcUmin >= prxUmin - eps && srcUmax <= prxUmax + eps;
    bool insideV = srcVmin >= prxVmin - eps && srcVmax <= prxVmax + eps;
    if (insideU && insideV)
        return;

    // Otherwise the face domain is wider — the proxy extends the surface over its
entire range
    // (TransposeUV = false: the U/V axes are not swapped).
    sgr.SetSurfaceUVDomain(surfaceId,
                           srcUmin, srcUmax, srcVmin, srcVmax,
                           prxUmin, prxUmax, prxVmin, prxVmax,
                           false);
}
```

PMI (Product Manufacturing Information)

PMI (Product Manufacturing Information) — annotations of manufacturing information: dimensions, tolerances, threads, etc.

A PMI object is created by a `BeginObject_*` call, filled in through its interface, and written to the SGF by a call to `EndObject`.

Method	Description
`BeginObject_PMIThread(CADID: string): IPMIGeomThread`	Create a thread object and return the `IPMIGeomThread` interface (see IPMIGeomThread) for filling it in. `CADID` is the object's unique identifier in the CAD.
`EndObject(obj: IUnknown): boolean`	Write the filled-in object (for example, the one obtained from `BeginObject_PMIThread`) to the SGF file. Returns a flag indicating whether the write succeeded.

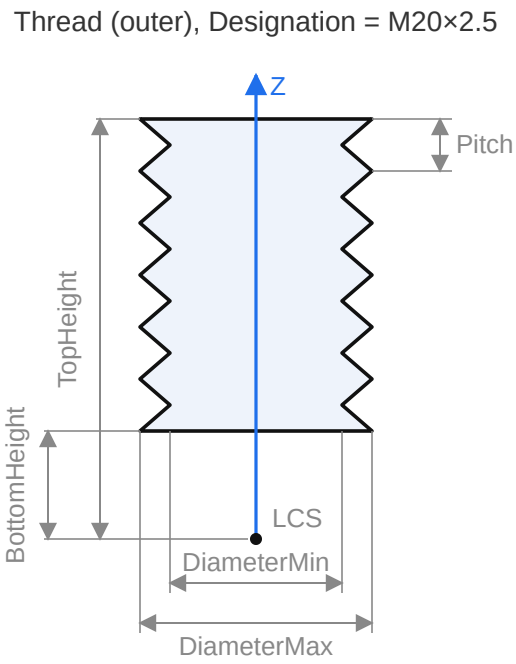
Description of PMI objects:

- IPMIGeomThread — the thread object: parameters, local coordinate system, associative references to geometry, and a save example.

IPMIGeomThread

Describes a thread object for PMI. An instance is returned by the `ISTGeomReceiver.BeginObject_PMIThread` method (see PMI (Product Manufacturing Information)); after being filled in, the object is written to the SGF via `EndObject`.

Each property is accessed through a pair of `Get<Name>` / `Set<Name>` methods (for example, `GetPitch` / `SetPitch`).



Property	Type	Purpose
`Designation`	`string`	Thread designation.
`LCS`	`TST3DMatrix`	Local coordinate system of the thread.
`TopHeight`	`STFloat`	Upper bound (height).
`BottomHeight`	`STFloat`	Lower bound.
`Pitch`	`STFloat`	Thread pitch.
`StartCount`	`integer`	Number of starts.
`DiameterMin`	`STFloat`	Minimum diameter.
`DiameterMax`	`STFloat`	Maximum diameter.

Property	Type	Purpose
`ConeAngle`	`STFloat`	Cone angle (for a tapered thread).
`DirectionIsLeft`	`boolean`	Left-hand thread.
`IsOuterThread`	`boolean`	External thread.

Associative references to geometry (optional). A thread can reference the source model geometry it was built from, by lists of string CAD identifiers. These references are optional: the thread is saved correctly without them; they merely establish an associative link to the geometry.

Method	Description
`AddBaseObjectCADID(Value: string)`	*(optional)* Associative reference to the base object — the supporting surface on which the thread was created.
`AddInitialBorderCADID(Value: string)`	*(optional)* Associative reference to the geometry of the thread's initial boundary.
`AddFinalBorderCADID(Value: string)`	*(optional)* Associative reference to the geometry of the thread's final boundary.

Filling order. The thread object is created by a call to `BeginObject_PMIThread` (which returns the `IPMIGeomThread` interface), filled in with parameter setters, and written to the SGF by a call to `EndObject`. The thread's position is defined by its local coordinate system `LCS` (`TST3DMatrix`), and its extent along the axis by `TopHeight` / `BottomHeight`. The associative references to the source geometry (`AddBaseObjectCADID`, `AddInitialBorderCADID`, `AddFinalBorderCADID`) are optional — they can be left unset.

Example of saving a thread (PMI).

```
// Save a thread (PMI) from CAD system data
bool SaveThread(CADThread thread)
{
    // open the thread object and get the interface for filling it in
    IPMIGeomThread pmi = sgr.BeginObject_PMIThread(thread.Id);
    if (pmi == null)
        return false;

    // thread parameters
    pmi.SetDesignation(thread.Designation);     // designation, e.g. "M12x1.5"
```

```
    pmi.SetPitch(thread.Pitch);                 // pitch
    pmi.SetStartCount(thread.StartCount);       // number of starts
    pmi.SetDiameterMax(thread.OuterDiameter);   // outer diameter
    pmi.SetDiameterMin(thread.InnerDiameter);   // inner diameter
    pmi.SetConeAngle(thread.ConeAngle);         // cone angle (0 for a
cylindrical thread)
    pmi.SetDirectionIsLeft(thread.IsLeft);      // left-hand thread
    pmi.SetIsOuterThread(thread.IsOuter);       // external thread

    // position and extent
    pmi.SetLCS(ToMatrix(thread.Placement));     // local coordinate system
→ TST3DMatrix
    pmi.SetTopHeight(thread.TopHeight);
    pmi.SetBottomHeight(thread.BottomHeight);

    // optional associative references to the source geometry — only if they are set
    if (thread.BaseSurfaceId != null) pmi.AddBaseObjectCADID(thread.BaseSurfaceId);
// supporting surface
    if (thread.StartBorderId != null)
pmi.AddInitialBorderCADID(thread.StartBorderId);  // initial boundary
    if (thread.EndBorderId   != null) pmi.AddFinalBorderCADID(thread.EndBorderId);
// final boundary

    // write the filled-in object to the SGF
    return sgr.EndObject(pmi);
}
```

Deprecated and unused methods

> Deprecated / unused. The methods below are not in use and are kept only for reference and backward compatibility. There is no need to call them in new Add-ins.

They are grouped by area — in the same order as the current chapters: file → solid → curves → surfaces → PMI.

Model parameter initialization

A pair of methods that used to initialize the importer parameters before reading SGF. The functionality has been moved into `StartFile`.

Method	Description
`StartModel()`	Initialize the importer parameters (after `StartFile`).
`CloseModel()`	Reset the parameters (before `CloseFile`).

Solid assembly

The old way of building a solid without `ComboSolid` (Working with a part): the vertices and edges of the solid were declared directly, and a group of faces was marked as a closed shell for stitching by the CAM system. Replaced by `ComboSolid`.

Method	Description
`CreateSolidVertex(ID: string; P: TST3DPoint)`	A solid vertex (end point of an edge).
`CreateSolidEdge(ID, SourceCurveID, svID, tvID: string)`	A named edge between the vertices `svID`⋯`tvID` based on the curve `SourceCurveID`.
`SetGroupEntityIsSolid(Value: boolean)`	Stitching flag: marks the surfaces inside the current folder (`StartGroupEntity`) as a closed shell.

Line properties

The type and width of the line for subsequent objects. Not used in SGF.

Method	Description
`SetCurrentLineType(Value: TSTLineType)`	Line type (see `TSTLineType`).
`SetCurrentLineWidth(Value: integer)`	Line width.

The `TSTLineType` enumeration: `ltSolid`, `ltDash`, `ltDashDot`, `ltDot`.

Ignoring out-of-range parameters

Controlled whether the surface is evaluated beyond its domain of definition (the U/V parameter range). No need to call: with correct input data the point parameters do not go beyond the surface range, so the flag is not required.

Method	Description
`SetIgnoreOutOfRange(Value: boolean)`	A one-shot flag: it is set before creating a surface, applies only to the single next surface, and is then reset. `true` — out-of-range is ignored (the surface is extrapolated beyond its boundaries); `false` — the range is strictly enforced.

PMI annotations

The old mechanism for building PMI views and annotations. The current PMI mechanism is described in PMI (Product Manufacturing Information).

Method	Description
`CreateView(vT, vZ, vX: TST3DPoint; Scale: STFloat; id: string)`	Create a PMI view.
`AddViewEntity(ViewID, EntityPath: string)`	Attach an entity to a view.
`BeginPmiEntity(id: string; vT, vZ, vX: TST3DPoint; PmiType: TSTPmiType)`	Begin a PMI annotation (see `TSTPmiType`).
`PmiAddArrow(vT: TST3DPoint; Angle: STFloat; ArrowType: TSTPmiArrowType)`	Add an arrow (see `TSTPmiArrowType`).
`BeginPmiText(vT: TST3DPoint; Angle, Height, Length: STFloat; NumberLines: integer; TextType: TSTPmiTextType)`	Begin a text block (see `TSTPmiTextType`).

Method	Description
`AddPmiText(Text: string)`	Add a line of text.
`EndPmiText()`	Finish the text block.
`PmiAddCurve(CurveID: string; CurveType: TSTPmiCurveType)`	Add a curve (see `TSTPmiCurveType`).
`PmiAddMesh(ID: string)`	Add a mesh to the annotation.
`SetPmiPreferenceI(Path: string; Value: integer)`	Integer annotation parameter.
`SetPmiPreferenceD(Path: string; Value: STFloat)`	Double annotation parameter.
`SetPmiAssociativity(i: integer; ObjId: string)`	Attach the i-th anchor object to the annotation.
`EndPmiEntity()`	Finish the PMI annotation.

PMI enumerations:

Enum	Values / purpose
`TSTPmiType`	Types of PMI annotations (28 values: dimensions, leaders, notes, etc.).
`TSTPmiCurveType`	Types of PMI curves (15 values).
`TSTPmiArrowType`	Types of PMI arrows (22 values).
`TSTPmiTextType`	Types of PMI text (14 values).