



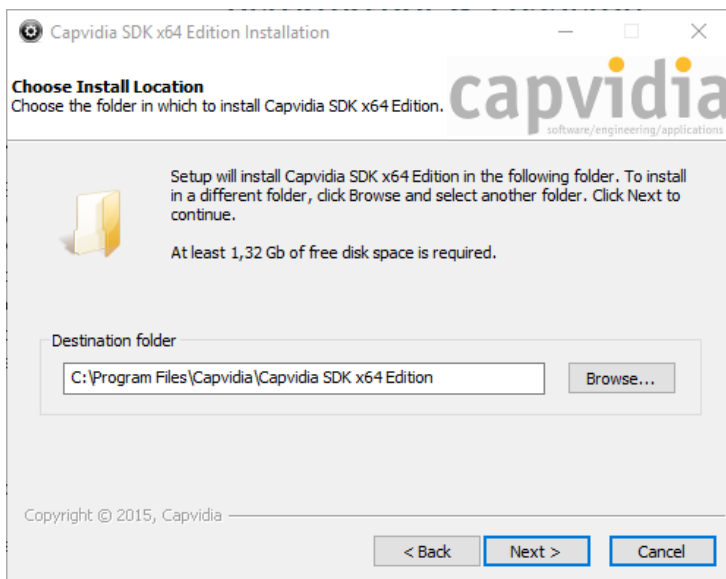
Capvidia SDK

Installation & Licensing

Hardware and Software Recommendations

- x86-64 compatible processor with SSE2 instructions
- At least 2 gigabytes (GB) of RAM (8 GB is recommended)
- At least 1.5 gigabytes (GB) of available space on the hard disk
- Keyboard and Mouse or some other compatible pointing device
- Video adapter and monitor with (1024 x 768) or higher resolution
- Operating Systems: Windows® 10 Home, Enterprise, or Pro edition, Windows® 8 Standard, Enterprise, or Professional edition, Microsoft Windows® 7 Enterprise, Ultimate, Professional, or Home Premium
- Platforms: 64-bit

Delivery



- `<InstallPath>/` - runtime components, libraries and resources
- `<InstallPath>/include/` - header files
- `<InstallPath>/lib/` - export libraries
- `<InstallPath>/sample/` - samples of the library usage
- `<InstallPath>/xsd/` - the internal xml file format documentation
- `<InstallPath>/CapvidiaSDK_Installation.pdf` - a basic description of the SDK
- `<InstallPath>/capvidia_api.chm` - API documentation

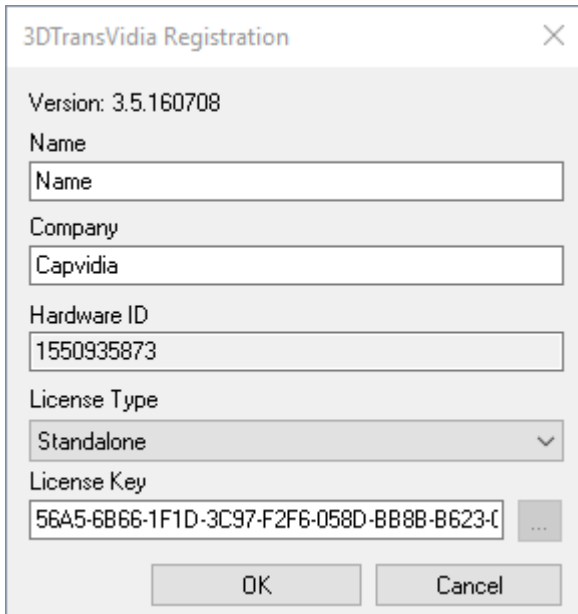
Where *<InstallPath>* is an *installation location* where the SDK was installed.

Installing Capvidia SDK

If you have a previous version of SDK installed, you should first uninstall the previous version. To use the SDK, the Visual Studio 2013 libraries are required. It is recommended to install all Windows critical updates.

Capvidia SDK Licensing Procedure

Follow these steps to obtain and input a license key for the Capvidia SDK:

The image shows a Windows-style dialog box titled "3DTransVidia Registration" with a close button (X) in the top right corner. The dialog contains several input fields and a dropdown menu. The "Version" field is pre-filled with "3.5.160708". The "Name" field is empty. The "Company" field is pre-filled with "Capvidia". The "Hardware ID" field is pre-filled with "1550935873". The "License Type" dropdown menu is set to "Standalone". The "License Key" field is pre-filled with "56A5-6B66-1F1D-3C97-F2F6-058D-BB8B-B623-C" and has a small "..." button to its right. At the bottom of the dialog are two buttons: "OK" and "Cancel".

3DTransVidia Registration

Version: 3.5.160708

Name
Name

Company
Capvidia

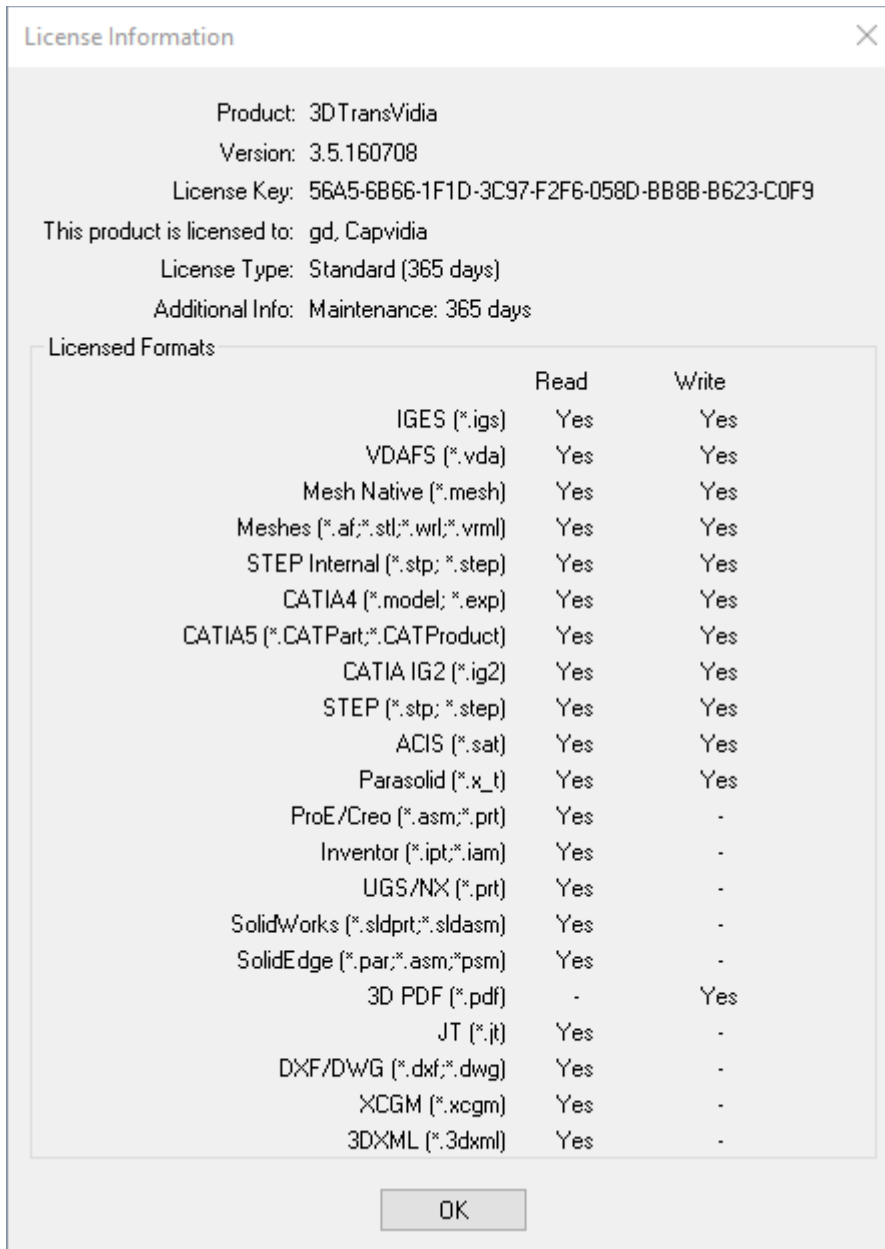
Hardware ID
1550935873

License Type
Standalone

License Key
56A5-6B66-1F1D-3C97-F2F6-058D-BB8B-B623-C

OK Cancel

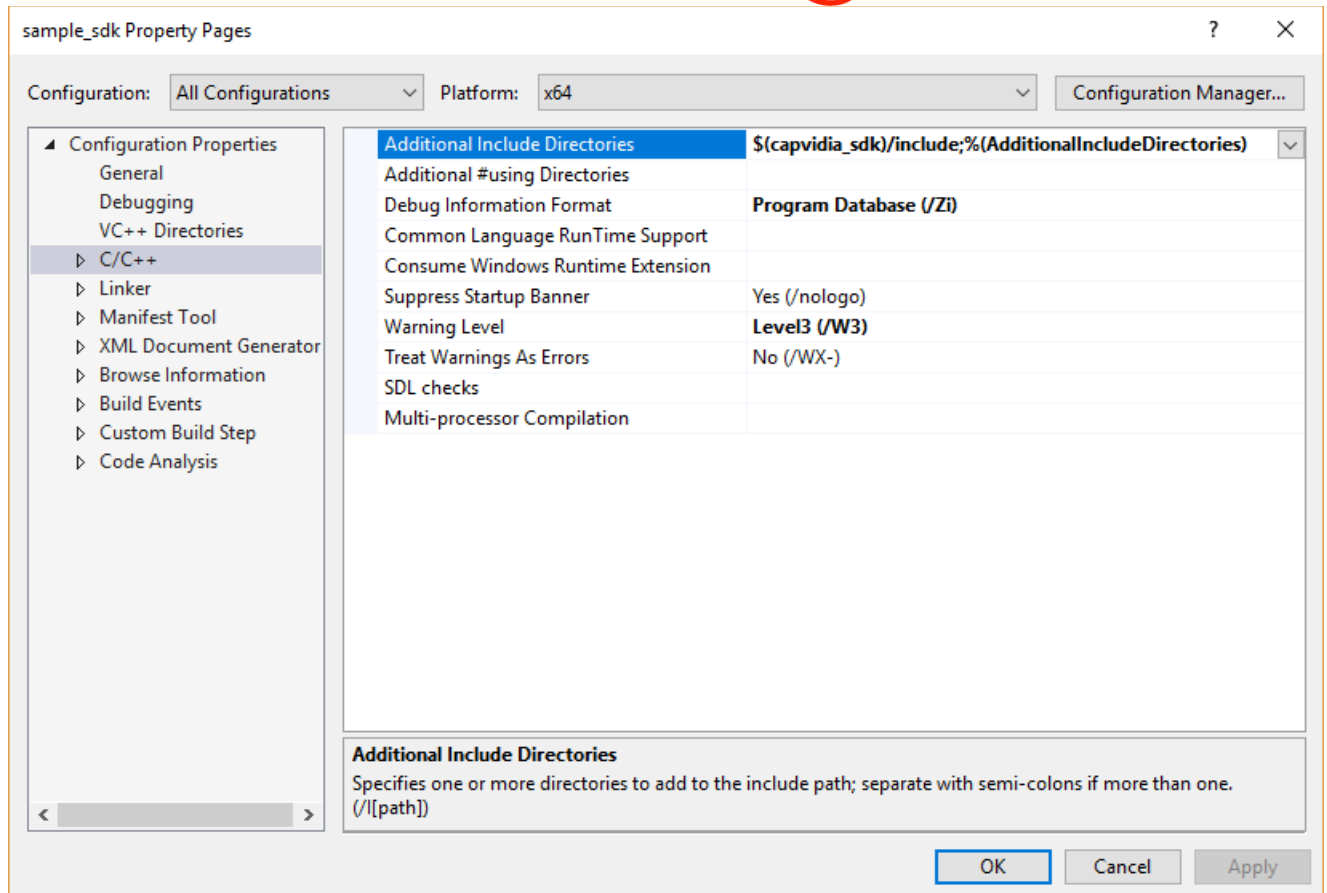
1. You can make a license request at the end of the SDK installation process or run the command *<InstallPath>\registration.exe 3DTransVidia /change* after the installation.
2. If you do not have a license or your license has expired, you will see the registration dialogue box.
3. Fill-in the form fields. After you enter the information, click the "Send" button. This will automatically generate an e-mail with your data to be sent to Capvidia support.
4. If you do not have direct access to the internet, choose the "Save" button to make a text file that can be send as an attachment to support@capvidia.com.
Our support staff will use this information to generate a license key. This key is a combination of 24 letters and characters and will be sent to you via e-mail.
5. After receiving the license key - enter the name and the company name into the registration dialogue box exactly as you did when you requested the license key. Type ('copy & paste' recommended) the key you received in the License Key field and push the OK button.
Note: There should not be any empty spaces or extra characters in front of and at the end of the license key string.
6. Information about the acquired license can be displayed by the command *<InstallPath>\registration.exe 3DTransVidia /info*



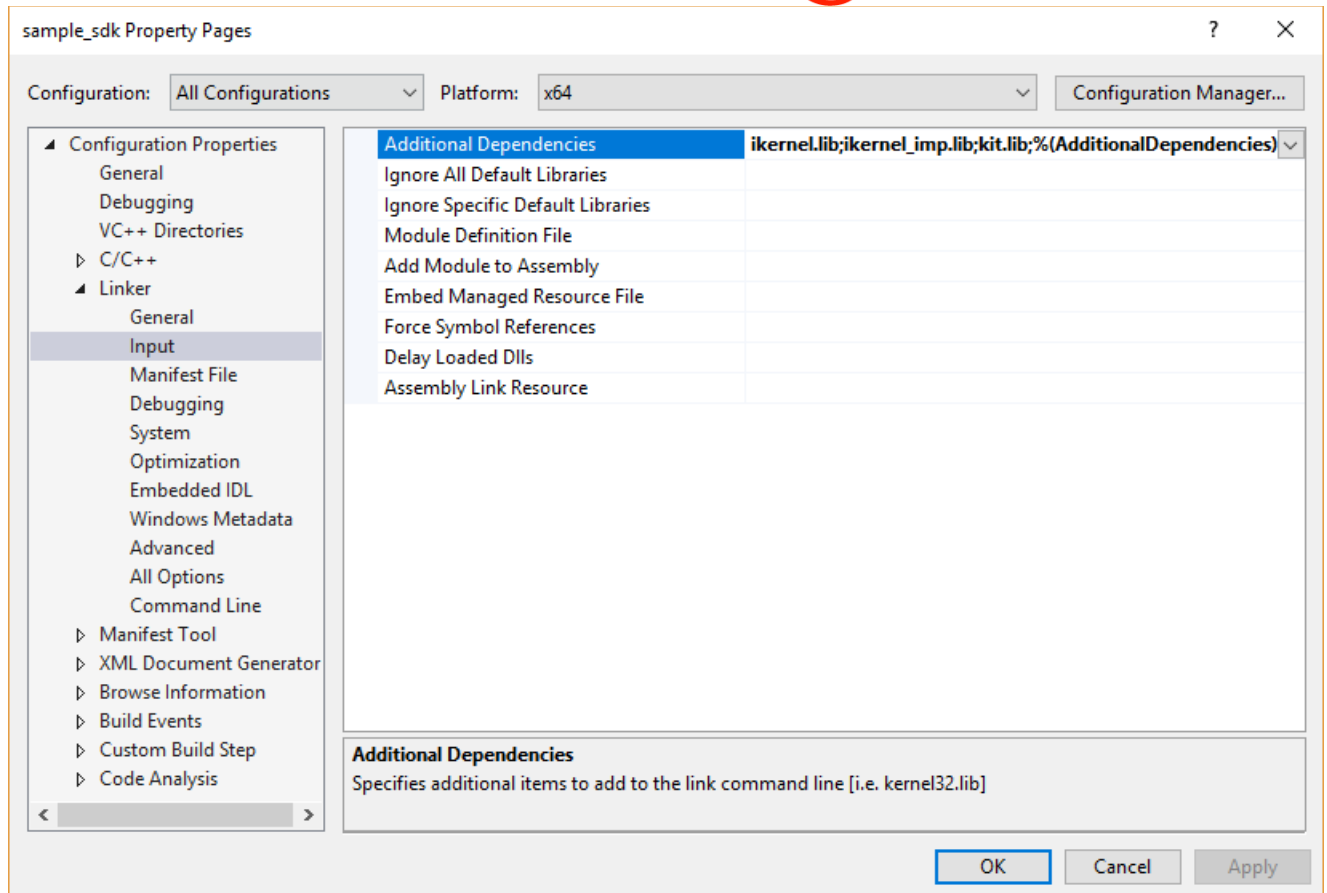
Integration of the SDK with an external application

For integration of the SDK libraries to a project, you need to:

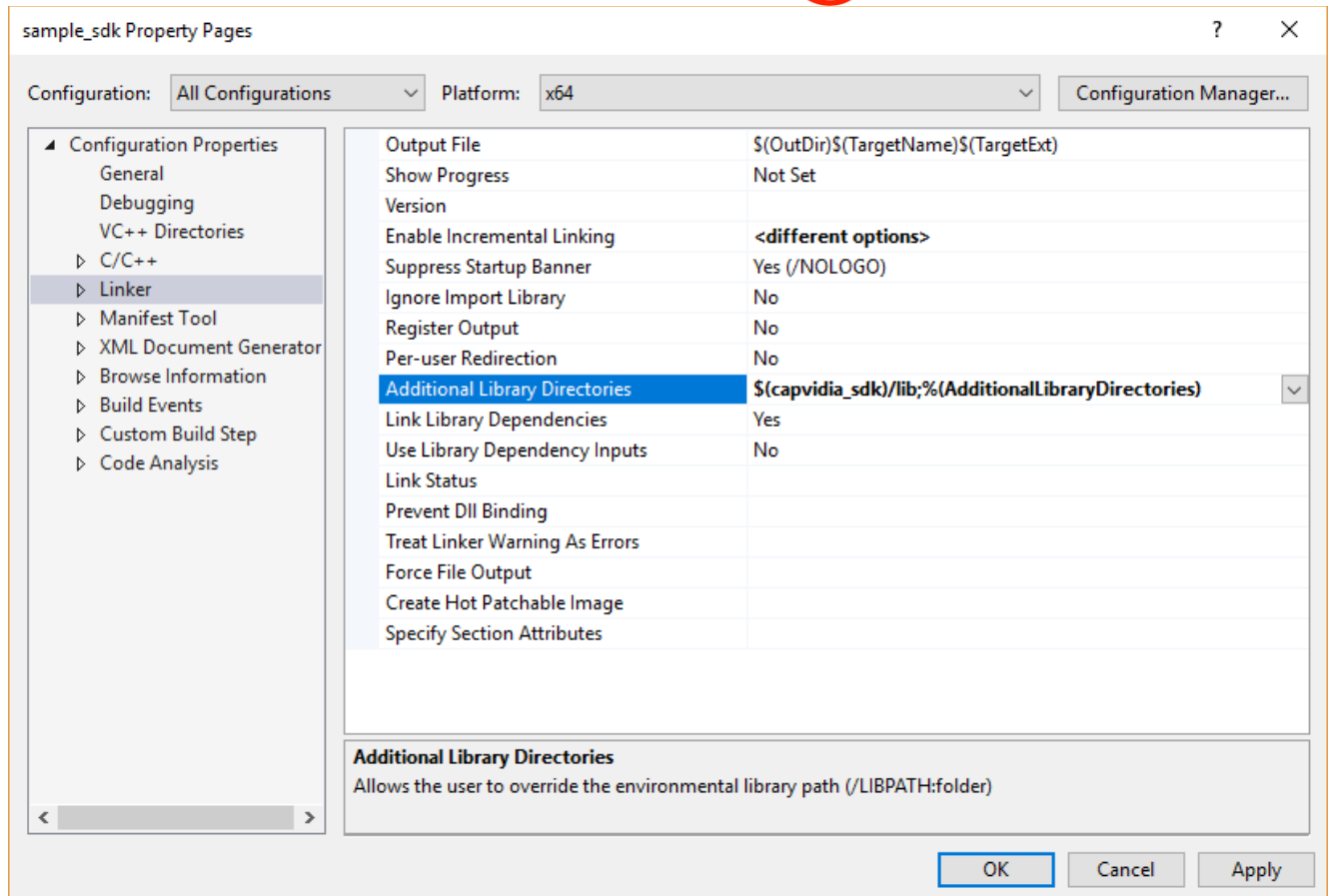
- Specify the *SDK root folder* location (<InstallPath>) in the "capvidia_sdk" environment variable (this step is normally performed by the installer).
- Specify the *SDK include folder* location in Visual Studio project settings:



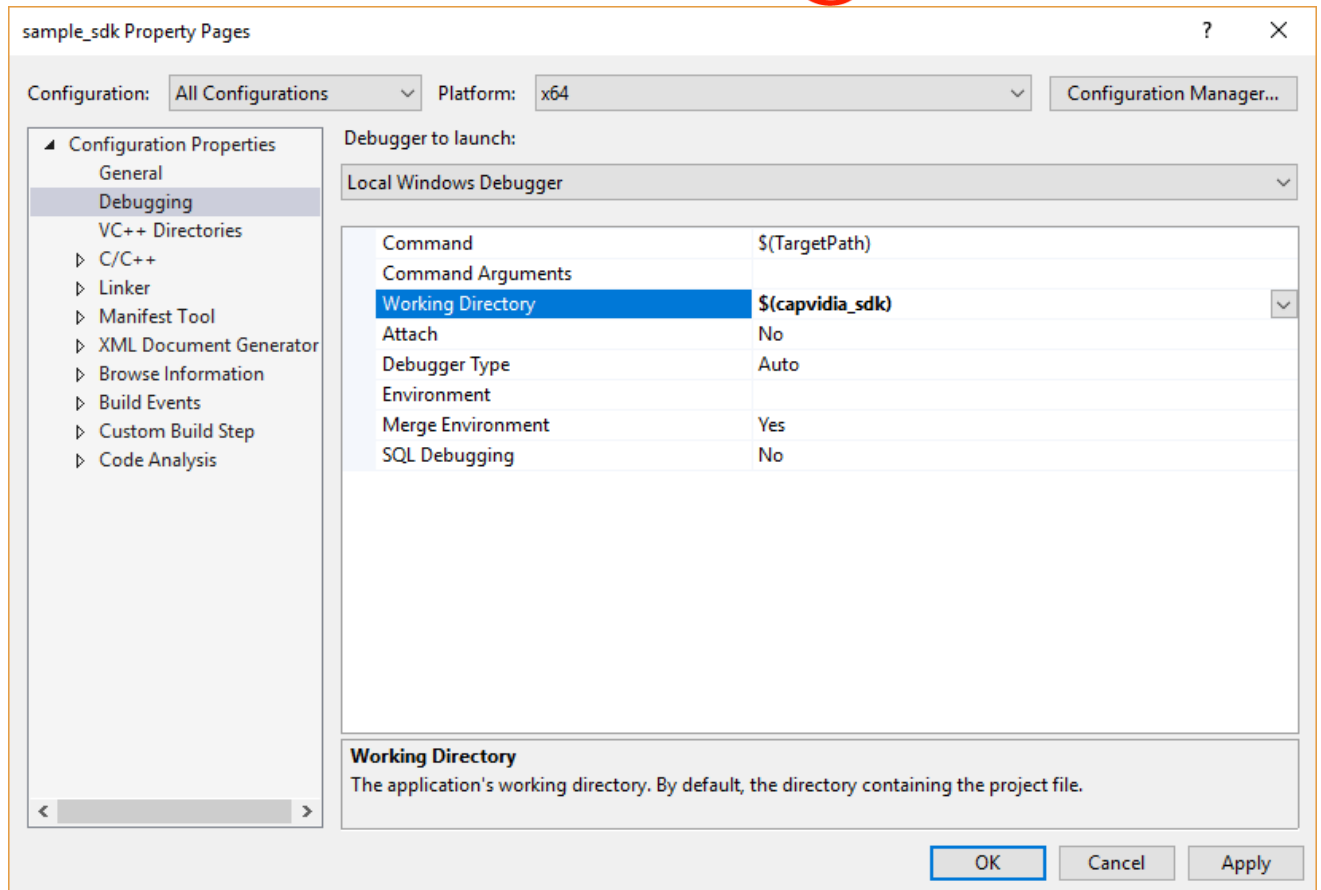
- Specify dependencies to the *SDK libraries* – *ikernel.lib*, *ikernel_imp.lib* and *kit.lib* in Visual Studio project settings:



- Specify the *SDK library folder* location in Visual Studio project settings:



- Add the "capvidia_sdk" variable to the system variable "Path" or add it into the "Working Directory" variable of the Visual Studio project settings.



Documentation of the internal XML file format

Documentation of the internal xml file format is represented in a form of the xsd schema files and located in the folder “<InstallPath>/xsd”. These xsd-files fully describe the model structure, model entities and relations between different entities.

Log files

The *workflow.~.xml* file is a log file that can help resolve potential issues. When contacting Capvidia for a support issue, please always attach this log-file. The log-file location: “%temp%/workflow.~.xml”.

Note: *workflow.~.xml* in SDK samples redirected to <testname>.test.xml.workflow.~.xml file in the folder of test file.

Note: The log-file grows until its size exceeds 1 MB after that it will be overwritten.

SDK samples

Overview

You can find an integration example (sample_sdk.sln) in the “<InstallPath>/sample” folder.

When you work with the SDK samples, we recommend that you copy the samples to a separate folder so you can preserve the original source code (otherwise, you would have to reinstall the Capvidia SDK). By

default, the SDK samples are located in the “%ProgramFiles%” folder; therefore, if you work with the SDK samples directly, you must use administrator credentials to run any code editing and build tools.

To compile sample_sdk.sln you need to:

- Open sample_sdk.sln
- Build the “Release|x64” configuration

To run tests you need to:

- To run all tests:
 - Start the *sample_sdk.exe* with the command line “<SampleSolutionPath>/test/_all.xml”
 - or start “<SampleSolutionPath>/test/run_tests.bat”
- To run one test:
 - Start the *sample_sdk.exe* with the command line “<SampleSolutionPath>/test/<testname>.xml” (note: fitting tests located in subfolder “<SampleSolutionPath>/test/fit”).
 - or start “<SampleSolutionPath>/test/run_test.bat <testname>”, where <testname> - test file name relative “<SampleSolutionPath>/test” folder
- Normally for a unit job file “<testname>.test.xml” Capvidia SDK produces the following output files in the folder of test file: “testname.test.xml.<suffix>.<extension>”. Caution: New output files overwrite existing files.

List of SDK samples

The SDK package includes sample functions that are listed in the table below. Every sample function is delivered with one or more test data files, e.g. the fitting function from fit.cpp is accompanied with more than 500 tests demonstrating different use cases.

Source code	Description
asm_filepath.cpp	Query the assembly component paths
approximatecrv.cpp	Approximation of curves by NURBS curve
attributes.cpp	Work with attributes (UDAs)
boundingbox.cpp	Scene bounding box calculation
calcmassproperties.cpp	Scene mass properties calculation
characteristicsqif.cpp	Traverse the scene characteristics
cleanpaths.cpp	Building cross-sections and cleaning section paths
cloud_join.cpp	Join of point clouds
cloud_meshing.cpp	Conversion of a point cloud to a mesh
cloud_noise_reduction.cpp	Point cloud de-noising
cloud_normals.cpp	Estimation of point cloud normal vectors
cloud_outliers.cpp	Outlier detection and removal
cloud_simplification.cpp	Point cloud simplification
cloud_split.cpp	Split a point cloud by a set of planes
convert.cpp	Model conversion from one format to another
convert_optional.cpp	Model conversion from one format to another with use of the options

convertpmi.cpp	Conversion of a model with PMI to STEP and QIF files with external references
copyentity.cpp	Copying different model entities from one scene to another
copyvertices.cpp	Copying vertices from one scene to another
createbox.cpp	Creation of a simple box (from 6 planar faces)
createcoonssurface.cpp	Creation of a curve on surface
createdim.cpp	Creation of a dimension
createface.cpp	Creation of trimmed and untrimmed faces
createfacemesh.cpp	Creation of a mesh face
curvatureextreme.cpp	Calculation of curvature characteristics
defeaturing.cpp	De-featuring
diagnostics.cpp	Diagnose a parametric model
duplicates.cpp	Find duplicate faces (completely and partially) in a model
edittrng.cpp	Triangle editing
featuresqif.cpp	Feature recognition and traversing of the feature tree
findcorners.cpp	Finding corner points on a curve
fit.cpp	Use of the fitting algorithms
formats.cpp	List of supported formats
formsubstitutionskin.cpp	Substitution skin generation
gencrv.cpp	Generation a curve/edge
genpts.cpp	Generation of points/vertices
gensrf.cpp	Generation of a surface/face
geom.cpp	Calculation of geometric data for different model entities
intersectfaceface.cpp	Intersection of faces
intersectline.cpp	Intersect a model with a line
intersectsrfsrf.cpp	Intersection of surfaces
isedgesonline.cpp	Check if the edges lie on one line
joinedge.cpp	Joining of edges
linearization.cpp	Scene linearization
mesh_border_correction.cpp	Mesh boundary correction
mesh_curvatures.cpp	Mesh curvature characteristics calculation
mesh_detect_cylinders.cpp	Cylinder detection
mesh_diagnostics.cpp	Diagnose and fix geometric and topological mesh inconsistencies
mesh_edit.cpp	Mesh editing operations: triangle creation, triangle removal, edge flip, edge split, unsew edges, join and removal of vertices
mesh_filling_holes.cpp	Fill a mesh hole, a half opened mesh hole and build a bridge between two opposite parts of a mesh hole
mesh_join.cpp	Mesh join
mesh_normals.cpp	Calculation of mesh normal vectors
mesh_orient.cpp	Mesh orientation conforming
mesh_remeshing.cpp	Re-meshing algorithm
mesh_segmentation.cpp	Mesh segmentation

mesh_sew.cpp	Mesh sewing
mesh_simplification.cpp	Mesh simplification
mesh_smoothing.cpp	Mesh smoothing
mesh_split.cpp	Split a mesh by planes or curves
movebody.cpp	Move a body from one location to another
nearestptface.cpp	Find nearest point on face
nearestptshell.cpp	Find nearest point on shell
progress.cpp	Progress indicator
projedge.cpp	Project a curve to a parametric face
projptcrvonsrf.cpp	Project a point on a curve based on the underlying surface
projtrngpath.cpp	Project a path on a mesh
recognizecubetopo.cpp	Recognize cube topology on a solid
registration.cpp	Register meshes or point clouds
relation.cpp	User-defined relation
reverse.cpp	Reverse algorithm
reverse_srf.cpp	Reverse (single face) algorithm
savedview.cpp	Generate and traverse the saved views
saveselected.cpp	Save selected entities to a file
section.cpp	Form sections and section area hatching
selectby.cpp	Select entities by size, volume and bounding box
selectbyrectangle.cpp	Select entities by a rectangle window
selfintersectsurface.cpp	Check surface for self-intersections
setlengthunit.cpp	Set length units
sew.cpp	Sew scene
simplifyfacemesh.cpp	Simplification of a mesh face
splitbody.cpp	Split body
splitedge.cpp	Split edge
splitface.cpp	Split face
splitfacemesh.cpp	Split face mesh
splitshell.cpp	Split shell
splittrng.cpp	Split triangulation in a set of mesh faces
splittrngpaths.cpp	Split a mesh by paths
transaction.cpp	Use of transactions (the transaction manager)
transform.cpp	Apply transformation to model entities (block, body, face, edge, etc.)
traverse.cpp	Scene traversing
version.cpp	Query the SDK version
viewer.cpp	Scene viewer

Batch job XML file format

The Capvidia SDK supports batch job files that can include a number of other job files (unit or batch), e.g.:

```
<jobs>
<job name="ct-femur\_all.xml" />
```

```
<job name="hip_right\_all.xml" />
<job name="mould\_all.xml" />
<job name="polster\_all.xml" />
</jobs>
```

Quick Start (Reverse engineering applications)

Overview

The main header file for the reverse engineering application area is "ik_reverse.h". Other header files are required but they will be included implicitly.

Code Sample

Brief instructions of all necessary steps are:

1. Call the `ii::initKernel("3DTransVidia")` function before any other API calls.
2. Get an instance of a class factory using the `ik::factory()` function. Create a new instance of the reverse algorithm by calling the `IFactory::createReverse()` method.
3. Call `initModel()`. The model can be used for one or more mesh builds.
4. Call `initMesh()` to initialize the current mesh to be reversed.
5. Call `addPatch()` for each patch of the mesh. If the patch contains a whole mesh, then this method should not be used.
6. Call `build()` on the initialized mesh. One of the parameters is a bit field, where you may specify what kind of surfaces to use.
7. Call `getPatchInfo()` to get information about the resulting patches. It contains an error code, default error message and resulting surface type data.
8. At this point, you can call `saveModel()` to write the current model to an IGES or STEP file. You can also continue to use the reverse algorithm on new meshes of the same model (`initMesh()` + `build()`) in order to save all results in a single file.
9. Call `killModel()` to free all resources allocated for the model.
10. Call `release()` to delete the `ik::IReverse` instance. It frees all allocated resources that kept for the instance. Note: `killModel()` frees the current model resources only but not the algorithm core.
Alternatively, you can use the smart pointer `TPtrShared<ik::IReverse>` to delete it automatically.
11. The final call is `ii::unInitKernel()`.

Reverse job example

This is an example a reverse engineering job file:

```
<sample_reverse mesh="*.stl" bSew="1"/>
```

Attribute	Value	Default	Description
mesh	a mesh file or wildcards	-	Defines what mesh files should be processed. You can define a single file path or you can use wildcards to specify multiple files in the folder of the job file. It is also possible to use many

			wildcards and exclude some files, using ' ' as the separator and '\' as the exclude mark. For example the mask "1*.stl 2*.stl *3*.stl *4*.stl" means: all STL files which names are started with '1' or '2' but without files that names contain '3' or '4'. Note that the exclude symbol '\' in the mask can be used only once.
export	resulting file extension	igs	Defines the resulting file extension.
each	0 or 1	0	0 - all result faces shall be stored in one file 1 - for each specified input mesh file shall be produced a separate output IGES file
bSew	0 or 1	0	Specifies whether the resulting faces shall be sewn.
nPatch	an integer number	0	Specifies a desirable number of patches.
bSaveGrid	0 or 1	0	Specifies whether the built grid shall be saved.
epsSewMesh	decimal number	1.5e-8	Specifies a tolerance for the mesh sewing during the mesh file loading stage.

All available reverse options are described in the "ik_options.h" file inline documentation.

Output files:

- "<testname>.<extension>": the resulting model with reversed faces
- "<testname>.reverse.log.xml": the low-level log-file formed by the reverse algorithm
- "<testname>.sum.xml": summary of the reverse algorithm results

If in a job file the 'each' attribute was set to '1' (each="1") then "*.igs" and ".log.xml" files are produced for each input mesh file. The summary log-file is always the same for a given job file.

Test generation

To generate a new test file specify a test file name by calling setSaveTest() for COptReverse.

Quick Start (Fitting Algorithms)

Overview

The fitting API primarily consists of geometric primitives (such as circle, sphere, cylinder, etc.) and fitting algorithms grouped by fitting type (i.e. least squares, min zone, etc.) and available as method-functions of corresponding fitter classes, listed below.

- ik::IFitLeastSquares
- ik::IFitMinZone
- ik::IFitMinCircumscribed
- ik::IFitMaxInscribed

- `ik::IFitOneSided`

The full list of primitives includes:

- `ik::IGeomFitCircle2D` – 2D circle
- `ik::IGeomFitCircle3D` – 3D circle
- `ik::IGeomFitEllipse2D` – 2D ellipse
- `ik::IGeomFitEllipse3D` – 3D ellipse
- `ik::IGeomFitLine2D` – 2D straight line
- `ik::IGeomFitLine3D` – 3D straight line
- `ik::IGeomFitParallelLines2D` – a pair of 2D parallel lines
- `ik::IGeomFitParallelLines3D` – a pair of 3D parallel lines
- `ik::IGeomFitElongatedHole2D` – 2D elongated hole
- `ik::IGeomFitElongatedHole3D` – 3D elongated hole
- `ik::IGeomFitFreeFormCurve` – freeform curve
- `ik::IGeomFitSphere` – sphere
- `ik::IGeomFitCylinder` – cylinder
- `ik::IGeomFitCone` – cone
- `ik::IGeomFitTorus` – torus
- `ik::IGeomFitPlane` – plane
- `ik::IGeomFitParallelPlanes` – a pair of parallel lines
- `ik::IGeomFitElongatedCylinder` – elongated cylinder
- `ik::IGeomFitFreeFormSurface` – freeform surface

All supported «primitive/fitting type» combinations can be found in the table.

	Least Squares	Min Zone	Min Circumscribed	Max Inscribed	One Sided
2D/3D circle	+	+	+	+	
2D/3D ellipse	+	+			
2D/3D line	+	+			+
2D/3D parallel lines	+	+	+	+	
2D/3D elongated hole	+	+			
Freeform curve	+	+			
Sphere	+	+	+	+	
Cylinder	+	+	+	+	
Cone	+	+			
Torus	+	+	+	+	
Plane	+	+			+
Parallel planes	+	+	+	+	
Elongated cylinder	+	+			
Freeform surface	+	+			

Unconstrained Fitting

The code below assumes that following header files are included and approximation points are stored as a vector of 3D points.

```
#include "ik_geom_fit.h"
#include "ik_fit.h"
TVector<D3> aPt;
kt::xmlReadD3Ex(xml, "aPt", aPt);
```

A typical working scenario with the fitting API starts with the creation of a fitter instance and an instance of geometric primitive. For example, in case of the least squares sphere you can go in the following way:

```
TPtrUnique<ik::IFitLeastSquares> leastSquares(ik::factory().createLeastSquaresFit());
TPtrUnique<ik::IGeomFitSphere> sphere(ik::factory().createGeomFitSphere());
```

To obtain an unconstrained fitting result with no hints simply call the function below, necessary initial guesses will be generated by the fitting function internal logic:

```
leastSquares->fitSphere(*sphere, NULL, aPt.ac());
```

An unconstrained fitting method name is built accordingly with has the following naming convention:

```
fit{primitive}(...)
```

Where {primitive} is a substring of a primitive class name IGeomFit{primitive}.

The fitting algorithm accepts multiple hints:

```
TVectorP<ik::IGeomFitSphere> aHint;
leastSquares->fitSphere(*sphere, &aHint, aPt.ac());
```

For each initial guess an optimization task will be solved and the best (in terms of the fitting type) solution will be returned as a result.

In case if you do not have any hints available, you can pass an empty array of hints as a function argument. If the fitting algorithm will succeed then a set of internally generated initial guesses will be returned.

Constrained Fitting

A constrained fitting can be performed in almost the same fashion as an unconstrained one. For example, there are several options for a constrained sphere. For the sphere fitting with only moving along a specified axis allowed: only one is hint required and the constrained parameters will be extracted from the hint.

```
TPtrUnique<ik::IGeomFitSphere> hint = ...;
leastSquares->fitSphereZ(*sphere, *hint, aPt.ac(), dirTrans, ptTrans);
```

A constrained fitting method name is built accordingly with has the following naming convention:

```
fit{primitive}{unconstrained_DOFs}(...)
```

Where {primitive} is a substring of a primitive class name IGeomFit{primitive}.

{unconstrained_DOFs}:

- Locked – moving and rotation are restricted
- UVW – allow any rotation
- W – allow rotation around the Z-axis
- ZW – allow moving along the Z-axis and rotation around the Z-axis
- XYW – allow moving in the XY-plane and rotation around the Z-axis (the plane normal)
- Z – allow moving along the Z-axis

Another example – fit a cylinder allowing movement in the XY-plane and rotation around the plane normal:

```
leastSquares->fitCylinderXYW(*cylinder, hint, aPt.ac(), nmPln, ptPln);
```

Note that each hint passed as a function parameter should meet the requirements that are set by constraints. For example, in case of a sphere constrained by an axis the hint center should lie on the axis.

Saving Tests

In order to create a new test the following function should be called before running the fitting algorithm:

```
fit->setSaveTest("test_file_name");
```

Where "test_file_name" a file name of the new test. If "test_file_name" is NULL no tests will be generated.

Quick Start (Viewer)

Overview

The SDK provides an easy and fast way to create a fully functional 3D scene viewer. It is possible to create several viewers for a single scene.

In order to create the viewer, do the following steps:

- 1) Open (or create) a scene using `ik::modeler`
- 2) Create a window where you plan to place the viewer in.
- 3) Create the viewer using `ik::modeler`

```
ik::modeler().createView(pScene, hwnd)
```

where:

pScene – a pointer to the scene

hwnd – a window handle, where the viewer should be attached to

createView method returns *TPtrShared* object that will be destroyed automatically together with the parent class. IMPORTANT: all viewers must be destroyed before their scene.

For more information please take a look at the sample code in 'viewer.cpp' and the SDK help.