PROCEDURAL RUNTIME 2.6 ARCHITECTURE

Abstract

ArcGIS CityEngine is based on the procedural runtime, which is the underlying engine that supports also two GP tools in ArcGIS 10.X and drives procedural symbology in ArcGIS Pro. The CityEngine SDK enables you as a 3rd party developer to integrate the procedural runtime in your own client applications (such as DCC or GIS applications) taking full advantage of the procedural core without running CityEngine or ArcGIS. CityEngine is then needed only to author the procedural modeling rules. Moreover, using the CityEngine SDK, you can extend CityEngine with additional import and export formats. This document gives an overview of the procedural runtime architecture, capabilities, its API, and usage.

Esri R&D Center Zurich, Förrlibuckstr. 110, 8005 Zurich, Switzerland

Contents

Introduction

With the addition of CityEngine to Esri's family of desktop applications, the need for a tighter integration of the procedural 3D technology with Esri's line of 2D and 3D offerings emerged. The first step with CityEngine 2012 was extending the support of GIS data formats for seamless interoperability. With CityEngine 2013, the procedural core technology was modularized and now all Esri desktop applications use the same engine for procedural 3D model generation. Additionally, customers from the 3D content creation and entertainment industry asked for procedural technology independent of CityEngine for their applications and in-house pipelines which we wanted to answer with an independent procedural engine.

These are the main driving forces behind the development of the ArcGIS procedural runtime. The runtime is a software module that essentially encapsulates the procedural core of CityEngine into a few shared libraries that can be used independent of CityEngine by any client application running on Windows, Linux, or Mac OS. In addition to that, the runtime has been redesigned to better fit large-scale multi-core and server deployments. The goal of the CityEngine SDK is to offer you as a 3rd party developer a powerful yet convenient toolkit for your procedural modeling needs.

This document outlines the architecture, concepts, and design principles of the procedural runtime. Please refer to the API documentation for a more detailed discussion of the individual classes and functions.

Concepts

This section introduces the basic concepts of the procedural runtime. Prior knowledge of CGA (Computer Generated Architecture – a domain specific, proprietary procedural modelling language) and CityEngine helps understanding this section as an example use case of the procedural runtime.

Figure 1 gives an overview of the components involved. A client application (e.g. CityEngine [1]) invokes the runtime through the PRT API [2] whose key functionality is to trigger the procedural generation of 3D geometry which is performed by Shape Processing Units (SPUs [4]). In addition to the PRT API invocations, you as a 3rd party developer can hook into the generation process by extending the runtime through decoders, encoders and adaptors using the PRTX interfaces [3]. External resource access (e.g. for 3D models or textures) and caching is handled by the data backend [5]. The callback interface [6] closes the loop and communicates the generated 3D models back to the client application.

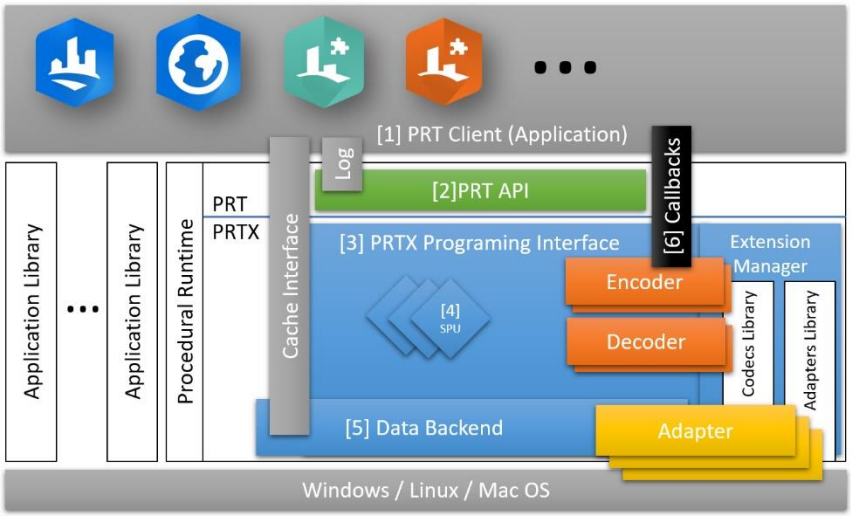


Figure 1 - Procedural runtime overview

The two worlds: PRT and PRTX

From a 3rd party developer point of view the procedural runtime consists of two worlds: PRT and PRTX. PRT is the API that you will invoke to trigger a procedural generation of 3D content. In order to do that, you configure PRT for your generation task and supply the necessary data, encoders, and rules to the runtime which in turn will produce the requested output.

PRTX is the extension interface of the procedural runtime (the X stands for eXtension). PRTX is a collection of classes and implementation guidelines that allow you to extend the runtime. Currently it can be extended by custom encoders and decoders (e.g. for 3D or raster file formats) as well as for accessing non-file based data sources (e.g. an asset library in a database). Codecs are not limited to file formats. For example, CityEngine uses an encoder that creates OpenGL data structures as part of the generation process.

Depending on your use case, you will work with both PRT and PRTX or only one of each. The following table highlights a few common use cases and their PRT and PRTX implications:

Use case examples	PRT	PRTX
Procedural 3D model generation in custom client application with the supplied codecs and callbacks (e.g. command line tool)	- Invoke PRT API from client application - Use one of the supplied callback implementations for results	
Integrate procedural 3D model generation with existing Digital Content Creation (DCC) application	- Invoke PRT API from DCC application - Implement custom callback for transferring 3D geometry data back to DCC application	- Write encoder for in-memory 3D geometry data suitable for custom callback implementation (or custom output format)
Create a language binding for PRT	- Wrap PRT API calls for target language - Implement custom callback for transferring 3D geometry data back to the caller	Write encoder for in-memory 3D geometry data suitable for custom callback implementation (or custom output format)
Extend CityEngine with custom export functionality		Write encoder for custom output format
Extend CityEngine with custom asset reader functionality		Write decoder for custom input format
Add custom asset repository access	Use PRT API with URI scheme for custom asset repository	Write adaptor for custom asset repository URI scheme
Custom (persistent) cache implementation / caching policy	Implement the PRT cache protocol	

PRT – Procedural Runtime Client API

PRT is a relatively concise API which allows you to submit generation jobs to the runtime and query the runtime for its capabilities in terms of encoders, decoders, and rule files. The API is organized in the `prt` namespace and minimizes the use of C++ constructs. Memory allocation and de-allocation is hidden behind a "compiler firewall" so the C-runtime library (CRT) of the client can be chosen freely. In turn, the API is free from STL and many C++ features (e.g. exceptions) in order to reduce compiler dependency. Compilers sharing the same Application Binary Interface (ABI) should be able to compile and link against the provided libraries and headers without special configuration. This ensures that a client application does not need to change its compiler and linker setting in order to access the PRT API. In consequence, the API is rather C-stylish and may need some interface glue in an advanced C++ client application. Esri provides headers and libraries for all supported platforms.

PRTX – Procedural Runtime Extension Interface

The PRTX programming interface is a larger collection of classes and functions that allow fine grained access to the data structures created during procedural 3D geometry generation (most notably the shape tree – see "Shape Tree" on page 7).

In contrast to the PRT API, the PRTX programming interface in the `prtx` namespace uses all available C++14 features (including exceptions which may occur at any point during execution) as well as a selected subset of STL classes. PRTX extensions are assumed to be provided in their own shared libraries and have to follow the same rules for compilation and linkage as the procedural runtime itself. Please refer to the individual platform technical notes for the specific compiler and linker settings.

Shape

Shape is one of the most fundamental data structures used throughout the procedural runtime. The CGA shape grammar is a language to define a set of configurations of shapes which are constructed by

applying CGA operations to shapes. Shapes are basically a labeled container for attributes and parameters. The starting shape for a generation process is called the *initial shape*. The various CGA operations modify shape attributes and/or create new shapes. A shape consists of:

- Symbol: The name or label of the shape. The symbol corresponds to the rule that will generate successor shapes. Note that multiple shapes with the same symbol may exist during a generation.
- Parameters: The CGA shape grammar is a parametric grammar and each shape can have a parameter list associated with it.
- Attributes: The attributes contain the spatial description (including the geometry) of the shape as well as key-value pairs. Attributes are accessible in the encoder and are used to create the resulting output geometry and materials. A shape has the following attributes:
    - Pivot: The pivot describes the shape's coordinate system and is defined by a position vector P (origin) and an orientation vector O (axis). The orientation vector is encoded as ordered rotation in degrees around x, y and z axis. The pivot is given in world coordinates.
    - Scope: The scope represents the oriented bounding box for the shape in space relative to the pivot and is defined by three vectors: the translation vector T, the rotation vector R (similar encoded as the orientation vector above), and the size vector S.
    - Geometry: The geometry attribute contains the 3D description of the shape's appearance (relative to the scope). The geometry is encoded as a polygonal mesh and may contain holes.
    - Shader and material attributes: These attributes further specify the appearance of the shape.
    - Generic attributes: These named attributes are defined in the CGA file and are typically user-settable in a client application (e.g. using the Inspector in CityEngine).

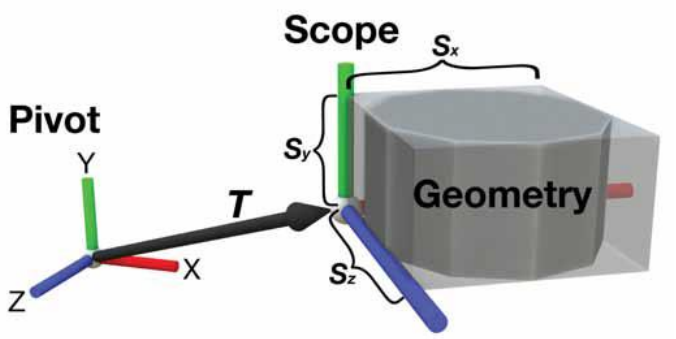


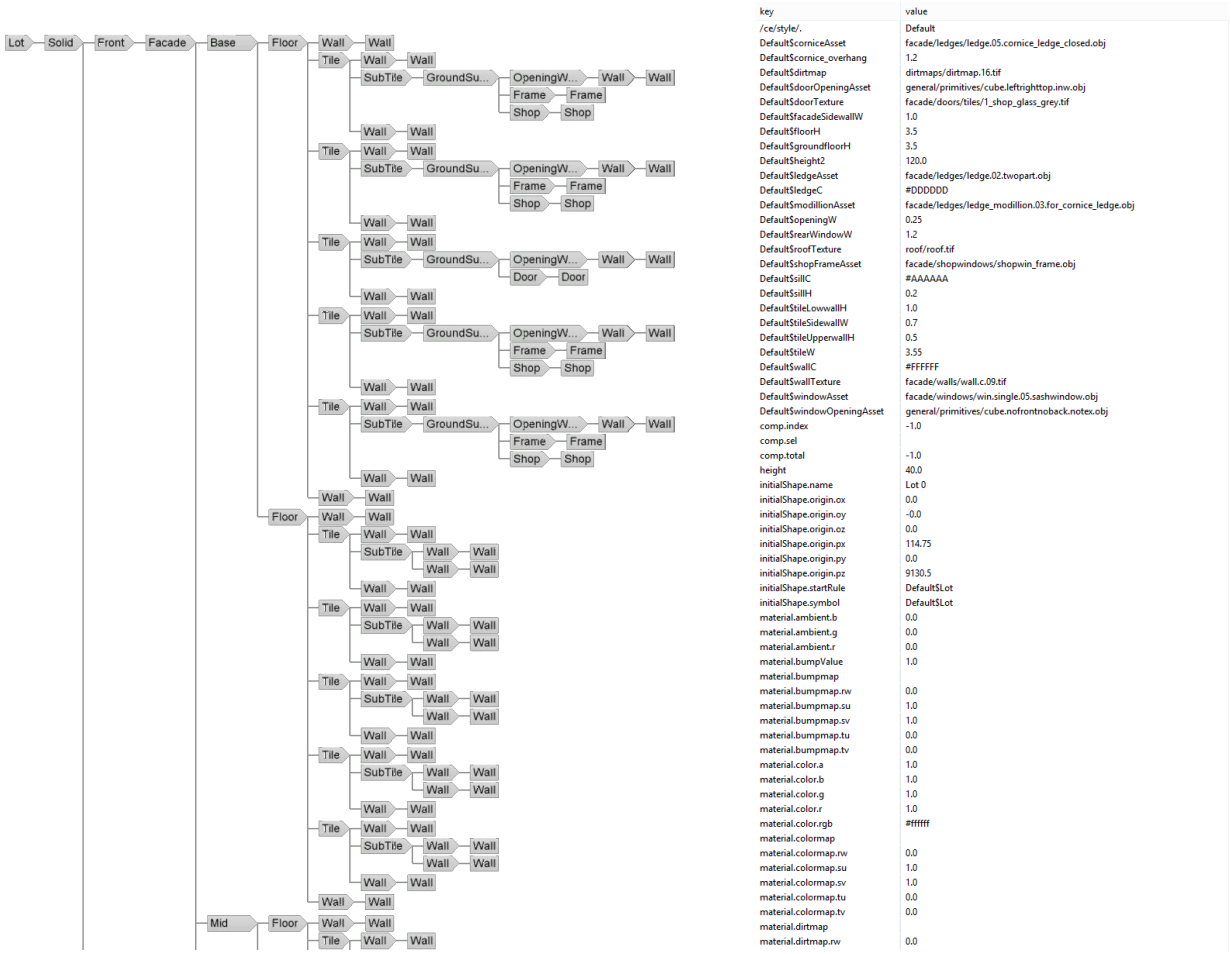
Figure 2 - Illustration of a CGA shape. Its parent coordinate system is defined by the pivot. Relative to the pivot, the scope defines a bounding box. Inside the scope, the geometry of the shape is contained. Note that CGA operates in right-hand Y-up coordinate system.

Shape Tree

The shape tree is the result of a generation and can be compared to a traditional scene graph with its transformational hierarchy. An encoder accesses the shape tree through an iterator that allows shape-tree traversal and accessing a shape and its attributes. The shape tree represents the generation history of the applied rules and contains in turn additional structural information. However, in most cases the structure itself is of no interest and only the geometries of the leaf nodes are combined into a 3D model by an encoder.

The main difference between a shape tree and the traditional scene graph is that each node in the shape tree contains geometry. Usually only the combined geometry of the leaf nodes are required for the final 3D model. But for some use cases or debugging purposes also intermediate shape geometries can be of interest. Figure 3 shows a part of a shape tree with leaf shapes as rectangular boxes and intermediate nodes as arrow boxes.

Because of its generative nature, it is very common and easy to repeat geometry with CGA operations. The procedural runtime detects and minimizes internal geometry duplications which enables efficient encoding for formats that support for example instancing.



key	value
/ce/style/.	Default
Default$corniceAsset	facade/ledges/ledge.05.cornice_ledge_closed.obj
Default$cornice_overhang	1.2
Default$dirtmap	dirtmaps/dirtmap.16.tif
Default$doorOpeningAsset	general/primitives/cube.leftrighttop.inw.obj
Default$doorTexture	facade/doors/tiles/1_shop_glass_grey.tif
Default$facadeSidewallW	1.0
Default$floorH	3.5
Default$groundfloorH	3.5
Default$height2	120.0
Default$ledgeAsset	facade/ledges/ledge.02.twopart.obj
Default$ledgeC	#DDDDDD
Default$modillionAsset	facade/ledges/ledge_modillion.03.for_cornice_ledge.obj
Default$openingW	0.25
Default$rearWindowW	1.2
Default$roofTexture	roof/roof.tif
Default$shopFrameAsset	facade/shopwindows/shopwin_frame.obj
Default$sillC	#AAAAAA
Default$sillH	0.2
Default$tileLowwallH	1.0
Default$tileSidewallW	0.7
Default$tileUpperwallH	0.5
Default$tileW	3.55
Default$wallC	#FFFFFF
Default$wallTexture	facade/walls/wall.c.09.tif
Default$windowAsset	facade/windows/win.single.05.sashwindow.obj
Default$windowOpeningAsset	general/primitives/cube.nofrontnoback.notex.obj
comp.index	-1.0
comp.sel	
comp.total	-1.0
height	40.0
initialShape.name	Lot 0
initialShape.origin.ox	0.0
initialShape.origin.oy	-0.0
initialShape.origin.oz	0.0
initialShape.origin.px	114.75
initialShape.origin.py	0.0
initialShape.origin.pz	9130.5
initialShape.startRule	Default$Lot
initialShape.symbol	Default$Lot
material.ambient.b	0.0
material.ambient.g	0.0
material.ambient.r	0.0
material.bumpValue	1.0
material.bumpmap	
material.bumpmap.rw	0.0
material.bumpmap.su	1.0
material.bumpmap.sv	1.0
material.bumpmap.tu	0.0
material.bumpmap.tv	0.0
material.color.a	1.0
material.color.b	1.0
material.color.g	1.0
material.color.r	1.0
material.color.rgb	#ffffff
material.colormap	
material.colormap.rw	0.0
material.colormap.su	1.0
material.colormap.sv	1.0
material.colormap.tu	0.0
material.colormap.tv	0.0
material.dirtmap	
material.dirtmap.rw	0.0

Figure 3 - A shape tree and a list of shape attributes.

CGB Files

CGA rule files are text files which define rules (shape transformations) by means of operations and are written according to the CGA language specification. CGA files are compiled by the CGA compiler into a binary representation called CGB (CGA-Binary). The CGA compiler is built-into the CityEngine desktop application. CGB is the only rule file format that can be consumed by the procedural runtime.

In order to avoid the introduction of yet another platform-independent object file format, a CGB file follows the Java class file specification. Hence tools operating on Java class files can also be used on CGB files. Even though CGB files are valid Java class files, a JVM is not required for execution. The runtime

provides its own VM implementation in the form of the Shape Processing Unit (SPU) optimized for procedural 3D geometry generation.

The PRT API allows a client application to introspect a CGB file (`prt::getRuleFileInfo()`) and extract information about rules and attributes. CGA annotations such as `@Range()` or `@StartRule` are directly accessible from the rule file info and can be used to dynamically configure e.g. a user interface for initial shape editing or to configure initial shape attributes automatically.

Shape Processing Unit

A CGB file is interpreted by one or multiple SPUs which implement the subset of the JVM specification required for the execution of procedural rules. A SPU executes a rule by applying operations to a given shape which will in turn create successors shapes and/or modify shape attributes. These successor shapes are added to a set of active shapes from which a pool of SPUs consumes shapes and again applies rules. This process continues until the set of active shapes only contains leaf shapes. Figure 4 illustrates this process during which the successor-predecessor shape hierarchy naturally forms the shape tree.

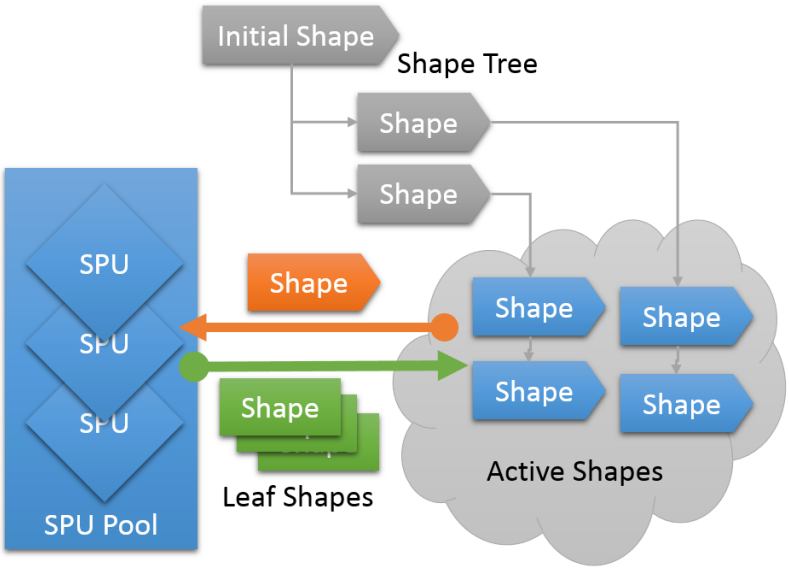


Figure 4 - A SPU removes a shape from the set of active shapes, creates successor shapes according to the shape rule and adds them to the set of active shapes.

Resolve Map

CGA defines operations (e.g. `i()` or `texture()`) that reference external assets such as 3D models (e.g. a Collada file) or textures (e.g. a TIFF file). These operations are used for setting the geometry and material attributes of a shape. Thus for most CGB files, supplementary assets are required in order to generate the expected 3D result. Asset references in CGA are *symbolic names* because they do not directly refer to a concrete resource but just act as a distinctive name. In order to translate a symbolic name to a resource access, the procedural runtime requires a supporting data structure that maps from symbolic names to URIs. This data structure is called the *resolve map* which contains symbolic names as keys and URIs as values. The client creates this map either programmatically using the `prt::ResolveMapBuilder` or extracts it from a rule package (see section “Rule packages”) by invoking `prt::createResolveMap()` and passes it to the runtime together with the initial shapes. The data

backend uses this map during the execution of operations that reference external assets for resolving the symbolic name to an actual URI. With the help of adaptors and decoders, the resource content is then consumed.

Rule packages

As mentioned above, most CGB files require additional assets in order to generate the expected 3D result. A rule package is an archive file with an `rpk` extension that encapsulate a CGB file and its supplementary assets. Rule packages are 7zip (www.7zip.org) archives and contain a CGB, 3D assets, and textures. Furthermore, the optional `.resolvemap.xml` entry in the root of the archive defines the mapping between symbolic CGA names and archive entries and is used by the procedural runtime to create a resolve map for a given RPK. Figure 5 shows the content of a rule package with the `.resolvemap.xml` at the top, a few assets and the CGB file at the bottom.

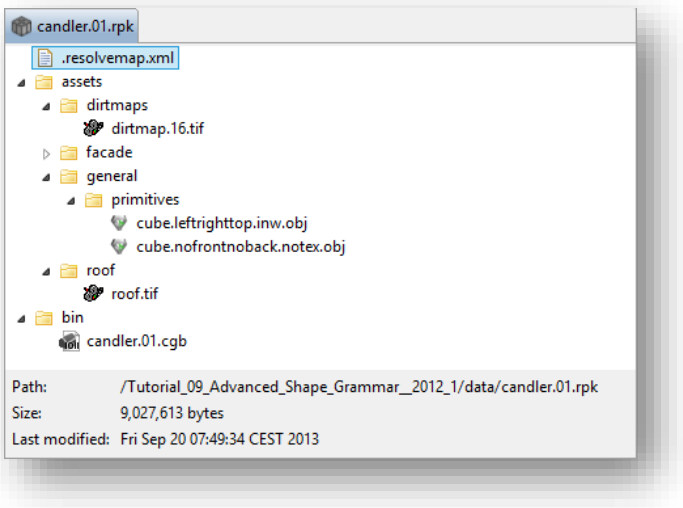


Figure 5 - The content of a rule package.

Rule packages are usually created with CityEngine or the CGA compiler. Rule packages written by CityEngine contain a single CGB file which has a single rule annotated with `@StartRule`. This is the required RPK format for the ArcGIS 10.2 GP tools. If you intend to write your own RPKs, it is recommended to follow to the single-CGB single-start-rule principle for maximum compatibility.

Although RPKs created with CityEngine reflect its project folder structure this is not required. A 3rd party application can equally create RPKs by just writing 7zip archives with any structure that is suitable. If you need more sophisticated resource organization and proprietary access for a specific use-case, PRTX adaptors allow fully customized implementations.

Adaptors

Adaptors are one of the two extension points of the procedural runtime. Adaptors can register themselves for a specific URI scheme and are responsible for resolving an URI and returning a stream that can then be processed by the decoders. In order to access a resource, the data backend uses the resolve map to translate a symbolic name into an URI. The URI's scheme is then used to find an adaptor that is able to handle the URI. The adaptor handles an URI by constructing a stream that can be passed

on to decoders which in turn read from that stream and create in-memory PRTX data structures (e.g. `prtx::Geometry`). Decoders are searched with the file extension extracted from the URI. Application specific URI schemes can be registered for special resource lookup and decoder combinations. ArcGIS for instance uses the `memory` URI scheme (see below) to give direct memory access to decoders running in the same address space. Currently the following adaptors are supplied:

- The file system adaptor handles the `file` URI scheme and returns a stream to the file content denoted by the URI. See http://tools.ietf.org/html/rfc1738 for a discussion of `file` URIs.
- The zip file adaptor handles the `zip` URI scheme and returns a stream of the zip file entry denoted by the URI. The `zip` URI scheme follows the http://www.iana.org/assignments/uri-schemes/prov/jar syntax.
- The 7zip adaptor handles the `rpk` URI scheme and returns a stream of the rule package entry denoted by the URI. The `rpk` URI scheme follows the http://www.iana.org/assignments/uri-schemes/prov/jar syntax.
- The memory adaptor handles the `memory` URI scheme and returns a memory block denoted by the URI as a stream. The scheme syntax is as follows: **`memory://`**`<address>/<size>.<ext>` whereas `<address>` is a hexadecimal number (without prefix) denoting a 64-bit address in the address space of the current process, `<size>` is the number of bytes referenced by this URI starting from `<address>`, and `<ext>` is the file extension used for decoder lookup.

As a 3rd party developer you can extend the procedural runtime with your own adaptors to handle specific schemes such as database access or a proprietary asset repository on a network server.

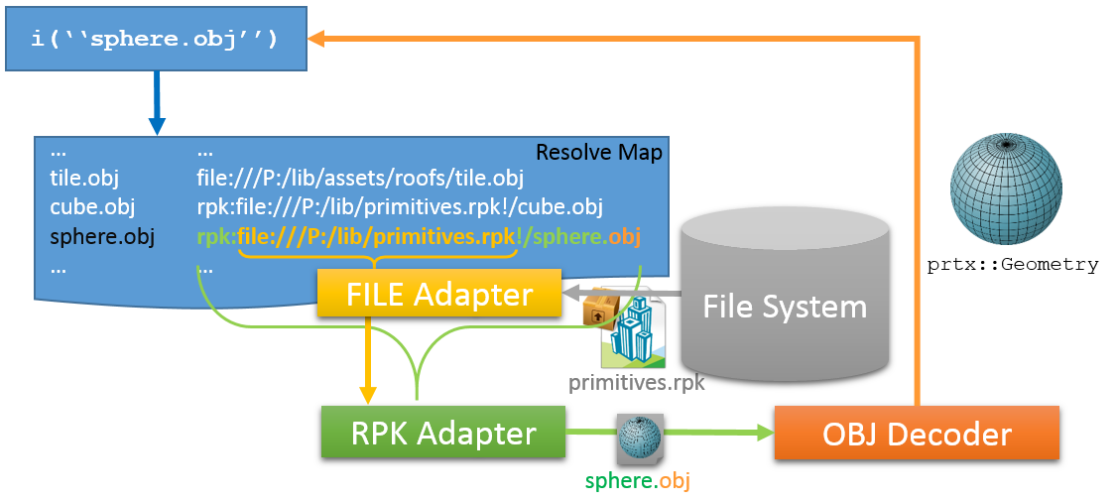


Figure 6 - Procedural runtime resource access. A symbolic name ("sphere.obj") is translated to an URI with the help of the resolve map. The URI scheme is then used to find adaptors which create a stream for further processing by decoders.

Codecs

Codecs are the second extension option of the procedural runtime for 3rd party developers besides the adaptors. Codecs translate between an external representation (e.g. a 3D file) and an internal data structure (e.g. a `prtx::Geometry`) for a given content type. There are two kinds of codec: *decoders* which read assets from external representations (e.g. the JPGDecoder or the OBJDecoder) and *encoders* which transform internal representations such as the shape tree to an external representation (e.g. the

FBXEncoder). Encoders are used to export procedural 3D models from CityEngine. Codecs can also convert between different in-memory representations, bypassing any serialization / deserialization which may be more time efficient e.g. in a game engine or a DCC application. The following content types are currently defined for codecs:

- `CT_GEOMETRY`: A geometry codec operates on `prtx::Geometry` objects. Currently Multipatch, Collada, OBJ, FBX, glTF, DWG, IFC and USD decoders are supplied as well as Multipatch, Collada, OBJ, FBX, 3WS, VOB, glTF, DWG, IFC, Alembic, SLPK, Datasmith and USD encoders.
- `CT_MATERIAL`: A material codec operates on `prtx::Material` objects. Currently decoders and encoders for both the MTL and .cgamat format are supplied.
- `CT_TEXTURE`: A texture codec operates on `prtx::Texture` objects. The texture codecs are based on GDAL (www.gdal.org) and decoders are supplied for all GDAL raster formats. In addition to that, PNG and JPEG encoders are supplied also based on GDAL.
- `CT_SHADER`: Shader definition. Currently only the built-in CityEngine shader definition is allowed and no custom implementation is possible.
- `CT_CGB`: There is a built-in decoder for CGB files which have the same format as Java class files.
- `CT_CGAERROR`: There is a built-in encoder for CGA runtime error messages which forwards to the callback interface `cgaError()` member function.
- `CT_CGAPRINT`: There is a built-in encoder for the CGA `print()` operation and function which forwards to the callback interface `cgaPrint()` member function.
- `CT_CGAREPORT`: There is a built-in encoder for the CGA `report()` operation which forwards to the callback interface `cgaReportBool()`, `cgaReportFloat()`, and `cgaReportString()` member functions.
- `CT_ATTRIBUTE`: There is a built-in encoder for attribute evaluation which forwards to the callback interface `attrBool()`, `attrFloat()`, and `attrString()` member functions.
- `CT_SHAPETREE`: CityEngine uses a dedicated encoder for encoding the shape tree structure. Shape trees can be cached for advanced use cases. See the encoder documentation for more details.
- `CT_STRING`: Generic string, used for text files (see the CGA `readTextFile()` function).
- `CT_TABLE`: Basic table, see CGA `readStringTable()` and `readFloatTable()` functions.

Client Callbacks

The main focus of the procedural runtime is shape processing. All functionality not related to this is either handled by subsystems such as adaptors and codecs or delegated to client callbacks. The client callback plays a central role in the communication between the generation process and the client application. The following functionality must be provided by the client callback:

- The client callback consumes encoder output. It is the responsibility of the client callback to write data received from an encoder to a file or other data sink and ensure consistency and locking. Data received from the runtime only remains valid during the execution of the specific callback function. If data has to be preserved for later use, the callback must copy the data.
- The client callbacks handles CGA features that directly produce output such as `print()`, `report()`, attribute evaluation, and error messages.
- The client callbacks translate between the world coordinate system used internally during the generation and various external representations such as WGS84.

- The client callback handles progress reporting and controls continuation of the generation process. For example user cancellation of the generation process can be implemented by the client application through the progress and continuation callback.

The following two callback implementations are currently provided and may be extended if needed by a specific client application:

- The `prt::FileOutputCallbacks` implementation redirects output to the file system.
- The `prt::MemoryOutputCallbacks` implementation redirects output to in-memory blocks.

Client Side Caching

Although the procedural runtime is stateless, state can be preserved between multiple calls on the client side if desired. This can speedup resource access considerably if e.g. geometry assets do not have to be decoded over and over again. Because no single cache policy can fit all client application needs, a client application can provide its own cache implementation. For example CityEngine as well as ArcGIS Professional provide a cache implementation that optimizes resource usage across the whole application among multiple subsystems.

The data backend, which resolves an URI (e.g. `file:/P:/lib/assets/sphere.obj`) to a content object (e.g. `prtx::Geometry`), uses the cache interface to access data associated with an URI. There are two main types of cache entries:

- Persistent data blocks: these are typed, serializable blocks of memory that may be persisted by the application. This is typically raw, decoded pixel-data or texture meta-data.
- Transient: these are typed, non-serializable, internal objects (e.g. `prtx::Geometry`, or `prtx::Texture`). Some transient resources are an aggregation of persistent data blocks. This allows for redundancy detection at the persistent data block level and prefetching of data blocks into the cache from a persistent source. CityEngine for examples stores persistent cache blocks in the ".prtcache" directory in the platform's temporary folder.

Please see the cache API documentation and examples for further details regarding the cache protocol. The following cache implementations are currently provided by the procedural runtime and can be created by calling `prt::CacheBuilder::createCache()`:

`CACHE_TYPE_DEFAULT`: The default implementation that uses a simple in-memory cache which simply grows if not flushed.

`CACHE_TYPE_NONREDUNDANT`: The non-redundant implementation uses an in-memory cache which simply grows if not flushed but detects and unifies duplicate persistent blocks, effectively avoiding data redundancy in the cache.

Logging

The procedural runtime provides an extensible logging interface which is usually configured before the initialization of the runtime. The following log levels are defined:

- `LOG_TRACE`: Internal call tracing. This log level is disabled in release builds of the runtime.
- `LOG_DEBUG`: Verbose debug information.
- `LOG_INFO`: Informational messages such as configuration information during initialization.

- `LOG_WARNING`: Warning messages issued at a condition that did not abort an ongoing execution.
- `LOG_ERROR`: Error messages issued at a condition that altered/aborted the ongoing execution.
- `LOG_FATAL`: Fatal errors that leave the procedural runtime in an inconsistent /non-operational state.

Clients can issue log messages by invoking `prt::log()` with the appropriate log level and message which will be processed by the logging facility the same way as procedural runtime internal messages. Log messages are silently discarded by default and have to be explicitly intercepted by one or more log handlers in order to be processed. The following log handlers are provided:

- `prt::ConsoleLogHandler`: Writes log messages to `stdout`.
- `prt::FileLogHandler`: Write log messages to a text file.

Client applications can implement their own subclasses of `prt::LogHandler` for their specific logging needs. CityEngine for example has a log handler that writes log messages to CityEngine's log window.

Design Principles

The procedural runtime is based on a couple of design principles that are also reflected in the API and the architecture. The following sub sections outline these principles and implications.

Modularity

Since the procedural runtime targets a broad range of use cases, it has a modular design enabling fine-tuning for a specific environment. Currently the procedural runtime can be customized with the following 3rd party implementations:

- Callback interface: Output handling is entirely delegated to the PRT callback interface.
- Logging: Clients can add their own log handlers to the runtime logging facility.
- Encoders: Writer for custom file formats or in-memory data structures can be implemented by writing PRTX encoders.
- Decoders: Reader for custom file format can be implemented by writing PRTX decoders.
- Adaptors: Custom resource access can be implemented by writing PRTX adaptors.
- Resolve Map Provider: Lists embedded secondary resources used by a PRTX decoder.
- Caching: Persistent caches or specific cache policies can be implemented through the PRT cache protocol.

Memory Management

- All PRT objects which are created through the API inherit from prt::Object and must be disposed by calling the destroy() member function. This ensures that object are allocated and freed by the same C-runtime and make the client implementation independent of the PRT C-runtime version. It is the responsibility of the client to destroy objects that it has created.
- PRTX utilizes shared pointers whenever possible. Therefore procedural runtime extensions generally do not need to care about freeing memory and pointer ownership.

Data Types

- In order to minimize locks, most PRT objects are immutable. Usually builders are used to create PRT objects (e.g. `prt::InitialShapeBuilder`). Builders are not thread safe and it is the responsibility of the client that a particular builder instance is not used by multiple threads.
- The procedural runtime is stateless. The client can freely destroy PRT objects after calls to the API.
- Double precision floating point numbers: the PRT API and PRTX interfaces use only double precision floating point numbers.
- Although the API is unit-less, the procedural runtime implicitly assumes that values and coordinates are in meters and Cartesian where appropriate. It is the client responsibility to convert values and project coordinates if it is using a non-metric or non-Cartesian systems.
- Wide strings: the PRT API and PRTX use `wchar_t*` and `std::wstring` respectively to avoid character encoding ambiguities (essentially using UTF-16 with the Basic Multilingual Plane) whenever possible. In cases where 8-bit character are required (e.g. `prt::Object.toXML()`), UTF-8 is the standard encoding. The only exception are subclasses of `prt::SimpleOutputCallbacks` which must be able to encode wide strings with the current platform encoding (`NATIVE`), UTF-8 (`UTF8`) and UTF-16 (`UTF16`).

- PRT objects can be converted to an XML representation with `prt::Object.toXML()` which simplifies and homogenizes debugging, logging, and serialization.

Multi-Threading

The procedural runtime is designed and implemented with today's multicore processors in mind. It supports coarse-grained parallelism through its thread-safe API. The implementation minimizes global locks, and parallel invocations of the PRT API typically progress concurrently. The SPU implementation may use fine-grained parallelism when appropriate.

Despite the procedural runtime's multi-threaded implementation, PRT API clients as well as PRTX codecs and adaptors can ignore concurrency most of the time:

- PRT API calls can be issued from any thread, just use your application's threading model.
- PRTX codecs are created and used on a single thread. No synchronization is necessary by the codec implementation unless global data must be shared between codecs.
- PRTX adaptors are created and used on a single thread. No synchronization is necessary by the adaptor implementation unless global data must be shared between adaptors.
- The procedural runtime accesses resources in a read-only manner through adaptors. All low-level write operations (e.g. exporting to a specific 3D file format) are delegated to the client callbacks. This avoids implementation of file locks or complex transaction behaviors as part of the procedural runtime.

Nevertheless, some calls are made concurrently by the procedural runtime and synchronization, locking, or transactions have to be implemented by the client for correctness.

- Callback implementations may be called from multiple threads in parallel. All callback member functions must be synchronized.
- Cache implementations may be called from multiple threads in parallel. All cache member functions must be synchronized.
- Log implementations may be called from multiple threads in parallel. All log member functions must be synchronized.

Error Handling

Because of the limited C++ usage for the PRT API, all PRT API calls return a status value (`prt::STATUS_xxx`) to indicate success or abnormal conditions and never throw exceptions. However, PRTX uses C++ exceptions extensively and PRTX 3rd party code can throw exceptions if appropriate. Error and warning conditions are usually also reported by corresponding messages through the logging interface. PRT objects implement a `toXML()` member function which allows easy printing and parsing of PRT data structures.

Internationalization

While great attention has been paid for supporting a wide range of localized environments (UTF-16, URI percent encoding, use of XML) the procedural runtime itself is English only. Since most of the runtime data has symbolic names anyway, mapping them to localized user interface strings should be straight forward in a client application. Only log messages are hard-coded in English and cannot be localized in the current version.

Using the Procedural Runtime API

This section highlights important parts of the PRT API which you will likely use from a client application. The PRT API lets you generate 3D models with the supplied adaptors and codecs. For example the ArcGIS Desktop geoprocessing tools use the memory adaptor, the Multipatch codec as well as the memory output callbacks as-is to implement in-memory 3D model generation using simply the PRT API.

The code samples below do not form a working example so please check the CityEngine SDK for complete, ready-to-customize code. All error handling as well as memory management has been omitted to keep the code samples small. In an actual application scenario it is highly recommended to check the status values, and `prt::Object`s that are created must be destroyed to avoid memory leaks.

Initialization and Shutdown

Before using any functionality of the procedural runtime it has to be initialized. During initialization, the runtime allocates internal data structures and dynamically loads codecs and adaptors from shared libraries. The only exception is the logging framework which accepts log handlers before initializations. Most client application register a log handler as a first step so they can also intercept runtime log messages during initialization. As stated above, error and memory handling has to be done on the client side. Calling `destroy()` is particularly important for the library handle returned by the `prt::init()` call. A typical initialization (and shutdown) sequence may look as follows:

Setup console logging:

```
prt::ConsoleLogHandler* logHandler
        = prt::ConsoleLogHandler::create(prt::LogHandler::ALL, prt::LogHandler::ALL_COUNT);
prt::addLogHandler(logHandler);
```

Prepare the procedural runtime extension library path (codecs and adaptors):

```
const wchar_t* extensionPath = L"C:\\dev\\ce_sdk\\lib"; // ce_sdk = CityEngine SDK
```

Initialize the procedural runtime:

```
const prt::Object* libraryHandle = prt::init(&extensionPath, 1, prt::LOG_INFO);
```

Release the library handle at application termination:

```
void shutdown() {
  libraryHandle->destroy();

  prt::removeLogHandler(logHandler);
  logHandler->destroy();
}

...
atexit(shutdown);
...
```

Resolve Map, Callbacks, and Cache

The procedural runtime centers around procedural 3D model generation and any auxiliary functionality is delegated to extensions or the client. Thus the client has to provide data and implementations that fit its needs. Several default implementations are provided for 3rd party developers as part of the SDK and can be directly created by the client or extended as sub-classes.

- The resolve map allows the data backend to translate symbolic names in CGA files to URIs for locating resources and is usually created from a RPK URI.
- The callbacks is the key interface through which generation results are communicated back to the client. Usually encoders send data to the client through `write()` calls. But application may use a dedicated pair of callbacks and encoder to optimize data transfer (e.g. by passing memory references instead of serialized geometry data) between the encoder and the client.
- The cache allows a client to carry data from one invocation of `prt::generate()` to the next which can speed-up the generation process considerably because assets are stored in an optimized internal format in the cache for reuse.

Create the resolve map for an RPK:

```
const prt::ResolveMap* resolveMap = prt::createResolveMap(L"rpk:file:/P:/lib/sample.rpk");
```

Create callbacks which writes to the file system:

```
prt::FileOutputCallbacks* callbacks = prt::FileOutputCallbacks::create(L"P:\\models");
```

Create a default cache:

```
prt::Cache* cache = prt::CacheBuilder::createCache(prt::CacheBuilder::CACHE_TYPE_DEFAULT);
```

Rule File Information

The procedural runtime defines several concepts and formats which a client must able to create and query. Although the runtime uses well documented standard formats (such as 7zip archives or Java class files) where possible, a client should not need to re-implement parsing of these formats. Therefore a few PRT utility functions allow convenient access to that information.

A resolve map is a simple key-value map which can be iterated and queried by the client. Because a resolve map can be created from an RPK URI, this enables easy introspection of the RPK structure and content.

Similarly, information about a rule file can be obtained for a given CGB URI. The `prt::RuleFileInfo` exposes the internal structure and data related to the source CGA file such as attributes, rules, and annotations. The information in the rule file info is sufficient for a client to dynamically configure the attributes of an initial shape. For example the inspector of CityEngine builds its user interface entirely from the rule file info (see Figure 7).

Find CGB file in resolve map:

```
size_t          nKeys;
wchar_t*        cgbURI = L"";
const wchar_t** keys   = prtNode->resolveMap->getKeys(&nKeys);
std::wstring    CGB(L".cgb");
for(size_t k = 0; k < nKeys; k++) {
  std::wstring key = std::wstring(keys[k]);
  if(std::equal(CGB.rbegin(), CGB.rend(), key.rbegin())){
    cgbURI = resolveMap->getString(keys[k]);
    break;
  }
}
```

Find start rule in CGB file by iterating through the rule file info:

```
prt::Entry*             startRule = 0;
```

```
const prt::RuleFileInfo* info      = prt::createRuleFileInfo(cgbURI, 0);
for(size_t r = 0; r < info->getNumRules(); r++) {
  if(info->getRule(r)->getNumParameters() > 0) continue;
  for(size_t a = 0; a < info->getRule(r)->getNumAnnotations(); a++) {
    if(!(wcscmp(info->getRule(r)->getAnnotation(a)->getName(), L"@StartRule"))) {
      startRule = info->getRule(r);
      break;
    }
  }
}
```

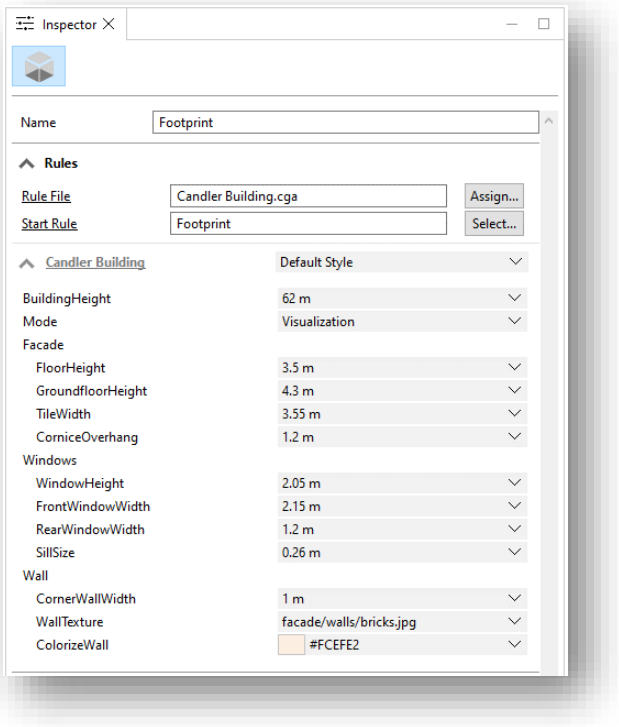


Figure 7- Inspector user interface dynamically created by CityEngine based on rule file info.

Initial Shape

The initial shape is the starting point for every generation process. Several attributes are mandatory and must be set before a generation can successfully take place. These are the URI of the CGB file, the name of the start rule (which becomes the shape symbol), the random seed, a (probably empty) map of CGA attributes, and a resolve map for resource lookup. In addition to that, a geometry has to be set which becomes the geometry attribute of the initial shape. The geometry is specified in a Y-up right-handed coordinate system and the unit is assumed to be meters (see “Shape” on page 6) by invoking:

```
prt::InitialShapeBuilder::setGeometry(double const* vertexCoords,
size_t vcCount, uint32_t const* indices, size_t indicesCount, uint32_t const* faceCounts, size_t
faceCountsCount, const uint32_t* holes, size_t holesCount);
```

- `vertexCoords` : The vertex coordinates as *(x,y,z)* tuples: $\{ x_0, y_0, z_0, x_1, y_1, z_1, \dots \}$.
- `vcCount`: The number of vertices in `vertexCoords`. The size of `vertexCoords` is 3 * `vcCount`.
- `Indices`: The vertex indices per face: $\{f_0i_0, f_0i_1, f_0i_2, f_0i_{f0Count-1}, \dots, f_1i_0, \dots\}$
- `indicesCount`: The size of `indices`.
- `faceCounts`: The number of vertices per face: $\{ f_{0Count}, f_{1Count}, \dots\}$

- `faceCountsCount`: The number of faces.
- `holes`: List which assigns hole-faces to faces. Holes must have the opposite vertex-ordering as the encircling face.
- `holesCount`: The number of entries in `holes`.

The operations of the start rule will be applied to that geometry as a first step in the generation process and typically create successor shapes which result in the shape tree.

Initial shape attributes are matched by name and type against attribute definitions in the CGB file and the generation will use the values from the CGA attribute map if they match. Typical examples of initial shape attributes for a buildings are e.g. `numberOfFloors`, `roofType`, or `LOD`.

A simple quad geometry:

```
static const double   Quad_vertices[]   = {0, 0, 0, 0, 0, 30, 30, 0, 30, 30, 0, 0};
static const uint32_t Quad_indices[]    = {0, 1, 2, 3};
static const uint32_t Quad_faceCounts[] = {4};
```

Create builders and set initial shape attributes:

```
prt::AttributeMapBuilder* amb = prt::AttributeMapBuilder::create();
amb->setFloat(L"numberOfFloors", 12);

prt::InitialShapeBuilder* isb = prt::InitialShapeBuilder::create();
isb->setAttributes(
        cgbURI,
        startRule->getName(),
        0,
        L"Quad",
        amb->createAttributeMapAndReset(),
        resolveMap);
```

Set initial shape geometry:

```
isb->setGeometry(Quad_vertices, 12, Quad_indices, 4, Quad_faceCounts, 1);
```

Create initial shape from builder:

```
const prt::InitialShape* shape = isb->createInitialShapeAndReset();
```

Generating 3D models

Having all data ready, generating a 3D model mainly becomes choosing a set of encoders and setting their options. A client can query the procedural runtime by invoking `prt::createEncoderInfo()` or you as a developer can have a look at the encoder documentation for available options of the supplied encoders. Encoder options are simple key-value maps and have to be validated before they can be passed to `prt::generate()`. Since encoders may cover a wide range of use cases, not all combinations of options are valid. The validation step ensures that only non-conflicting options are passed to `prt::generate()`. The `prt::EncoderInfo.createValidatedOptionsAndStates()` member function is free to change any option in order to create a valid configuration.

For every encoder option there is an associate `xxx_state` key that maps to one of the following string values:

- `"ENABLED"`: The corresponding option can be configured as part of this configuration.

- `"DISABLED"`: The corresponding option cannot be configured as part of this configuration and should not be changed.
- `"HIDDEN"`: The corresponding option is ignored as part of this configuration.

Although a client may find out more about the validation and option configuration by looking at the option state, its main purpose is supporting the dynamic generation and updating of a user interface for an encoder. For example CityEngine queries the encoder for its options and creates corresponding user interface elements. The visibility and enable/disable state of these user interface elements is directly controlled by corresponding state values (see Figure 8).

Even though the internal execution of `prt::generate()` may use multiple threads, the call itself is synchronous and all results are guaranteed to be passed to the callbacks before the call returns. The client is free to destroy all created PRT objects after `prt::generate()`.

Because `prt::generate()` tends to be expensive in term of CPU time, usually multiple encoders are passed to `prt::generate()` and run at the same time. If your application has multiple encoders which have to run together (e.g. writing transformation node information to one file together and the geometry to another file) it is also easier to pass multiple encoders than combining output of multiple `prt::generate()` calls as a post-processing step. The list of encoder of a typical invocation of `prt::generate()` include `com.esri.prt.core.CGAPrintEncoder`, `com.esri.prt.core.CGAErrorEncoder`, optionally the `com.esri.prt.core.CGAReportEncoder` if you need access to CGA reports, and content specific encoders. If some output is not required, it is more efficient to omit the corresponding encoder in the `prt::generate()` call than discarding the output in the callbacks. Even though the shape tree is only generated once for all encoders, every encoder iterates the shape tree individually and may cause considerable overhead for large shape trees.

Before calling generate you may setup occluders. See the documentation of `prt::generateOccluders()` and the CGA reference for occlusion queries for a detailed discussion of occlusion and the generation process.

Get encoder info:

```
const wchar_t*           encoder    = L"com.esri.prt.codecs.OBJEncoder";
const prt::EncoderInfo*  encInfo    = prt::createEncoderInfo(encoder);
const prt::AttributeMap* encOptions = 0;
```

Create encoder options:

```
encInfo->createValidatedOptionsAndStates(0, &encOptions);
```

Trigger procedural 3D model generation:

```
prt::generate(&shape, 1, 0, &encoder, 1, &encOptions, callbacks, cache, 0);
```

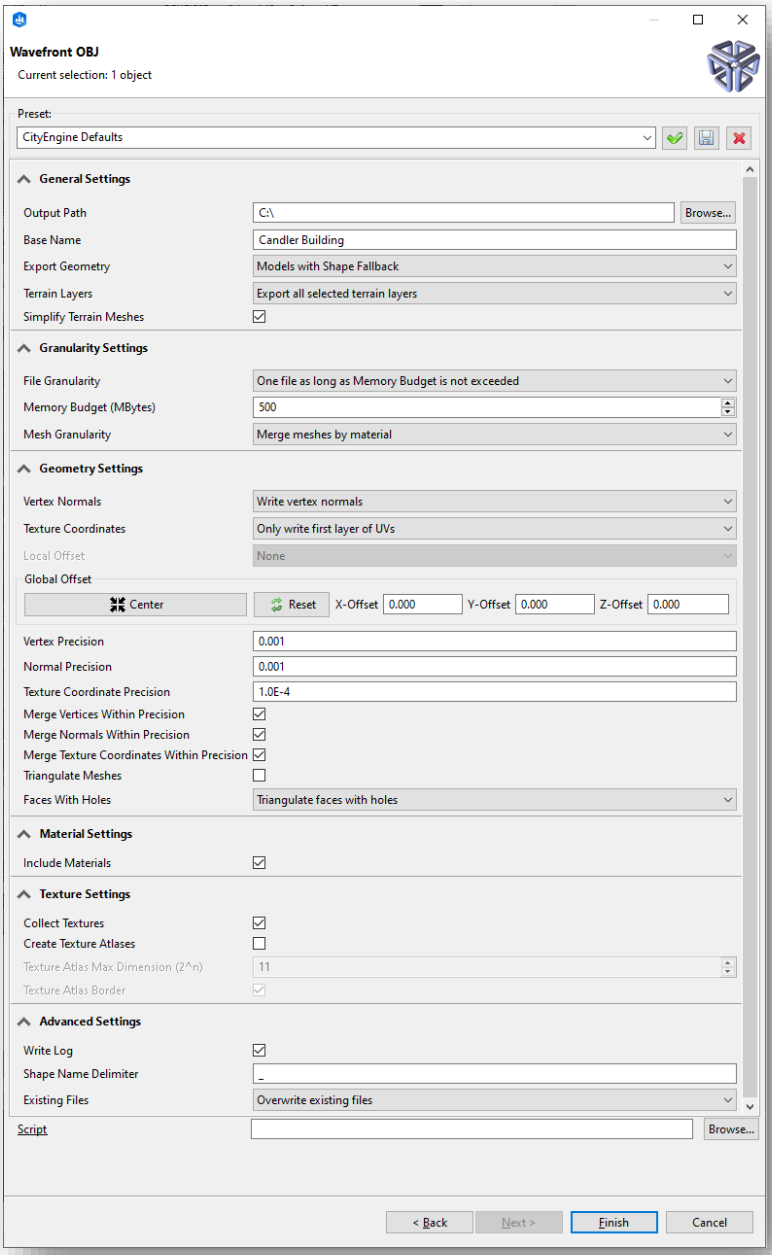


Figure 8- Export user interface dynamically created by CityEngine based on validated encoder options.

Writing Procedural Runtime Extensions

While the PRT API allows you as a 3rd party developer to work with rule packages and procedurally generate 3D models, you are limited to the SDK adaptors and codecs. To get past the PRT API you have to build a shared library and extend PRTX classes with custom code. This section outlines the implementation requirements for adaptors and codecs. Please note that all your classes and functions discussed here must be visible in the shared library and you have to export these symbols with `__declspec(dllexport)` on Windows and with `__attribute__(visibility("default"))` on Linux and Mac OS and are omitted in the code samples below for simplicity. The SDK includes complete examples of extensions which are recommended as a starting point for writing your own extension.

Extension Life Cycle

All procedural runtime extensions follow the same, managed life cycle. Extensions have to register factories which are used by the runtime to create adaptor and codec instances during PRT API calls. For every resource access or codec operation a new instance from the corresponding factory is requested. While there may be many instances of the same type in use at a given point in time, member functions on extensions are guaranteed to be invoked on a single thread only and therefore no synchronization is necessary (see “Multi-Threading” on page 16). Extensions should not allocate or use any global data because this would violate the stateless design principle (see “Memory Management” on page 15). Codecs have access to a cache instance for preserving state over multiple invocations although state is typically managed by the data backend for you.

In order to be loaded and initialized by the runtime during the `prt::init()` call, extension libraries must at least export the following functions:

```
extern "C" {
  void registerExtensionFactories(prtx::ExtensionManager* manager) {
    manager->addFactory(new MyFactory());
    ...
  }

  void unregisterExtensionFactories(prtx::ExtensionManager* manager) {}
  int getVersionMajor() {return 1;}
  int getVersionMinor() {return 0;}
}
```

The `getVersionMajor()` and `getVersionMinor()` functions state the runtime version this extension can work with. If the version constraint is satisfied, the runtime calls `registerExtensionFactories()` during initialization which allows an extension to register all its adaptor and codec factories with the runtime’s extension manager.

Factories

For each extension type there is a factory base class available in in the PRTX interface which you have to subclass as a 3rd party developer in order to provide custom functionality:

- `prtx::AdaptorFactory`: An adaptor factory which creates `prtx::Adaptor` instances. Implement a subclass if you want to add your own adaptor.
- `prtx::DecoderFactory`: An adaptor factory which create `prtx::Decoder` instances. Implement a subclass if you want to add your own decoder.
- `prtx::EncoderFactory`: An adaptor factory which create `prtx::Encoder` instances. Implement a subclass if you want to add your own encoder.

Factory implementations have to provide at least a unique ID, a name, and a description. It is recommended to use a reverse-domain scheme (e.g. "*esri*.**com**" becomes "**com**.*esri*.xxx") for the ID in order to avoid ID conflicts among different 3rd party implementations. Depending on the factory type, more information is required such as icons in the case of encoder factories. If multiple factories can handle a specific request, `prtx::Extension.getMerit()` is called for disambiguation. Higher merits are preferred over lower merits thus 3rd party extensions may override the supplied implementations by returning a high merit. Merits < 10.0 are reserved and the supplied extensions have a default merit of 1.0. Please see the extension specific class documentation for details regarding factory implementation.

Adaptors

Most adaptors will subclass `prtx::StreamAdaptor` which create a `std::istream` instance for a given URI. If the adaptor is able to handle structured content (e.g. for an archive file or a database) the `prtx::Adaptor.createResolveMap()` member function should be implemented as well. The following example shows a simple adaptor for `file` URIs:

```
std::istream* FileSystemAdaptor::createStream(prtx::URIPtr uri, const prt::ContentType& ct) {
  if (uri->getScheme() == prtx::URIUtils::SCHEME_FILE) {
    const boost::filesystem::path p(uri->getPath());
    if (!boost::filesystem::exists(p))
      throw std::invalid_argument("file system path does not exist:" + p.string());
    return new boost::filesystem::ifstream(p, std::ifstream::in | std::ifstream::binary);
  } else {
    log_werror(L"FileSystemAdaptor::createStream():unable to handle URI='%s'")%uri->wstring();
    throw prtx::StatusException(prt::STATUS_INVALID_URI);
  }
}

void FileSystemAdaptor::destroyStream(std::istream* stream) {
  delete stream;
}

prt::ResolveMap* FileSystemAdaptor::createResolveMap(prtx::URIPtr) const {
  throw util::RuntimeErrorST("FileSystemAdaptor::createResolveMap() is not implemented.");
}
```

Decoder

Decoders are constructed by the corresponding decoder factory which also supplies the decoder info. A PRT client can query that information by calling `prt::getDecoderInfo()`. A decoders is a codec implementation which converts a `std::istream` to a PRTX content object such as `prtx::Geomtry` or `prtx::Texture`. The PRTX interface provides the following base classes which a 3rd party extension writer can subclass:

- `prtx::GeometryDecoder`: Create a `prtx::Geometry` object from a given stream.
- `prtx::MaterialDecoder`: Create a `prtx::Material` object from a given stream.
- `prtx::TextureDecoder`: Create a `prtx::Texture` object from a given stream.

Decoders tend to be more complex than adaptors because they have to deal with complex data types such as geometry or material. Decoders are typically not written from scratch but use existing libraries for the main part of their work. For example the supplied texture decoders are based on GDAL and are just a small wrapper around GDAL functions.
The decoder's work has to be performed in the `decode(ContentPtrVectorVariant& results, std::istream& stream, prt::ICache* cache, const std::wstring& key, prtx::ResolveMap const* resolveMap, std::wstring& warnings)` member function. While typically only a single object is constructed by a decoder, some formats may translate to multiple instances (e.g. material files

with multiple material definitions or geometry with multiple level of detail) and thus a decoder fills the `results` vector instead of simply returning an instance. A key and a resolve map is also provided for cache lookups, or if a decoder needs access to URI specific components (e.g. the query part of an URI). Because a `prtx::Texture` and `prtx::Material` are attributable objects they supports optional metadata such as a spatial reference for textures. Please see the PRTX documentation for predefined metadata keys.

Resolve Map Providers

Some formats (e.g. geometry formats such as FBX, glTF, USD) use embedding of related assets (e.g. textures for FBX geometry). For such formats, Resolve Map Providers are used to list all secondary assets inside a file. Typically, a Resolve Map Provider works in tandem with an Adaptor. The Resolve Map Provider links all embedded assets to a URI that has the custom Adaptor's URI scheme as outer most scheme. The related decoder must be aware how the keys are constructed for these embedded assets to resolve them.

Encoder

Encoders are constructed by the corresponding factory which also supplies the encoder info. A PRT client can query that information by calling `prt::getEncoderInfo()`. The encoder info is crucial for implementing a generic client that can dynamically build for example a user interface without having prior (compile-time) information about an encoder.

Encoders transform the shape tree into a final 3D model or extract shape tree information (see "Shape Tree" on page 7). Like decoders, encoders are typically based on existing libraries and act as a wrapper around these library calls. For example the `com.esri.prt.codecs.VOBEncoder` is based on the closed-source VOB libraries provided by e-on software. Currently there are two base classes defined for geometry and texture:

- `prtx::GeometryEncoder`: Encode a 3D model through the callbacks interface based on shape tree information.
- `prtx::TextureEncoder`: Encode a 2D raster image through the callbacks interface based on `prtx::Texture`.

Usually encoders use the client callbacks `write()` member function to send data to the client. Such encoders are generic in the sense that all client implementations that use a subclass of `prt::SimpleOutputCallbacks` are able to process the encoder output. CityEngine for example is able to dynamically create a user interface for such an encoder and write the output to files. But some encoders may only work with a specific callbacks implementation and pass data by reference instead of serializing it via `write()` calls. CityEngine for example uses a render encoder that creates OpenGL data structures in memory that are passed as opaque handles back via an extended callbacks interface known to both the encoder and the client. The renderer uses these handles for rendering, unaware of the underlying geometry and material data on the GPU.

Encoder Life Cycle

Encoders are created as part of the generation process when the runtime is preparing the output. Before calling `prt::generate()`, the client usually prepares validated encoder options by calling `prt::EncoderInfo.createValidatedOptionsAndStates()`. Hence there is a close relationship

between the encoder info returned from the encoder factory and the actual encoder implementation. The actual encoding goes through the following steps:

1. After creation, `prtx::Encoder.init()` is called once with the current `prtx::GenerateContext` where an encoder can allocate data structures for subsequent `prtx::Encoder.encode()` calls.
2. Initial shapes are filtered by `prtx::Encoder.validate()`. This member function is invoked for every initial shape passed to `prt::generate()`.
3. For every initial shape that passed the `prtx::Encoder.validate()` filter, `prtx::Encoder.encode()` is called where the actual encoding takes place. Encoders may aggregate and collect information in data structures allocated during `prtx::Encoder.init()`. Please note that `prtx::Encoder.encode()` may also be called as part of a nested encoding. For example an encoder for a 3D file format may call a texture encoder during its execution.
4. After all initial shapes are processed, `prtx::Encoder.finish()` is called which allows the encoder to finish up and free resources allocated during `prtx::Encoder.init()`. Some encoder may defer part of their work to `prtx::Encoder.finish()` and only collect data during the `prtx::Encoder.encode()` calls. For example most of the supplied geometry encoders allow optional sorting by material which can only take place after all shape tree data from all initial shapes has been processed.

Encoder Utilities

Some encoding tasks are very generic and the PRTX interface provides a set of utility functions that relieve encoder writers from re-implementing this functionality. Please see the PRTX utility documentation for a detailed description. Here only the `prtx::EncodePreparator` and `prtx::NamePreparator` bear mentioning. The name preparator helps organizing name spaces according to specific rules e.g. for unique material and file names. The encode preparator handles most of the burden of combining and optimizing shape geometries into a final 3D model.

Check callbacks type, allocate and initialize members and utility objects:

```
void MyEncoder::init(prtx::IGenerateContext& context) {
  prt::SimpleOutputCallbacks* callbacks =
dynamic_cast<prt::SimpleOutputCallbacks*>(getCallbacks());
  if(callbacks == 0)
    throw prtx::StatusException(prt::STATUS_ILLEGAL_CALLBACK_OBJECT);

  mNamePreparator             = new MyNamePrepartor();
  mNamespaceMaterials         = mNamePreparator.newNamespace();
  mNamespaceMeshes            = mNamePreparator.newNamespace();
  mNamespaceDocumentFilenames = mNamePreparator.newNamespace();

  mPreparator = prtx::EncodePreparator::create(true, mNamePreparator, mNamespaceMeshes,
  mNamespaceMaterials);
}
```

Collect leaf shapes, extract combined geometry according to flags from encode preparator, and write them to callbacks. Please see the `prtx::EncodePreparator::PreparationFlags` for a detailed description:

```
void MyEncoder::encode(prtx::IGenerateContext& context, size_t initialShapeIndex) {
  const prtx::InitialShape& is = *context.getInitialShape(initialShapeIndex);

  try {
    prtx::LeafIteratorPtr li = prtx::LeafIterator::create(context, initialShapeIndex);
    for(prtx::IShapePtr shape = li->getNext(); shape.get() != 0; shape = li->getNext())
      mPreparator->add(context.getCache(), shape, is.getAttributeMap());
```

```
  } catch(...) {
    log_wtrace(L"MyEncoder::encode: error");
  }

  util::computeLocalOffset(&context, initialShapeIndex, mLocalOffsetMode, mPreparator,
mLocalOffset);

  std::vector<prtx::EncodePreparator::FinalizedInstance> finalizedInstances;
  mPreparator->fetchFinalizedInstances(finalizedInstances,
               prtx::EncodePreparator::PreparationFlags()
                       .instancing(false)
                       .mergeByMaterial(true)
                       .triangulate(true)
                       .mergeVertices(true)
                       .cleanupVertexNormals(true)
                       .cleanupUVs(true)
                       .vertexNormalAction(VertexNormalProcessor::PASS)
                       .holeAction(HoleProcessor::PASS)
                       .indexSharing(INDICES_SEPARATE_FOR_ALL_VERTEX_ATTRIBUTES)
                       .mergeToleranceVertices(1e-3)
                       .mergeToleranceNormals(1e-3)
                       .mergeToleranceUVs(1e-3)
                       .offset());

  prt::SimpleOutputCallbacks* callbacks =
    dynamic_cast<prt::SimpleOutputCallbacks*>(getCallbacks());

  const std::wstring name = util::getFilename(mBaseName, EXT.c_str(), mFileGranularity,
    mFileIndex, initialShapeName, mNamePreparator, mNamespaceDocumentFilenames);

  write3DModel(context, finalizedInstances, name, callbacks);
}
```

Write combined materials and cleanup:

```
void MyEncoder::finish(prtx::IGenerateContext& context) {
  prt::SimpleOutputCallbacks* callbacks =
    dynamic_cast<prt::SimpleOutputCallbacks*>(getCallbacks());

  writeMaterials(context, callbacks);

  delete mNamePreparator;
}
```

Conclusion

This document gives an overview of the ArcGIS procedural runtime. Please check the SDK examples and the SDK documentation for more detailed descriptions of classes and member functions. Other concepts such as occlusion have to be seen in the context of CGA thus the CGA online help is also a valuable source of information.

The goal of the CityEngine SDK is to give to you as a 3rd party developer a powerful procedural modeling toolkit in the most convenient way. Do not hesitate to contact Esri if you have specific needs, comments, or feedback regarding the procedural runtime and we will evolve the product accordingly.

List of Figures