Esri Indexed 3d Scene (i3s/i3d)   
Format Specification

Version 1.1.50, 2013-12-11  
Esri R&D Centers Zurich & Beijing

This document specifies the Indexed 3D Scene delivery format used to stream 3D GIS data to mobile, web and desktop clients. It’s the default format delivered by the ArcGIS Scene Service. The first sections of this specification explain the conceptual structure of i3s, while the latter sections provide a detailed implementation-level view.

[Requirements 2](#_Toc367801577)

[The i3s Store – what goes into an Indexed 3D Scene? 2](#_Toc367801578)

[The Index Structure 3](#_Toc367801579)

[Level of Detail Concept 4](#_Toc367801580)

[Coordinate Reference Systems 5](#_Toc367801581)

[Structure of i3s resources 6](#_Toc367801582)

[Texture Recommendations and Requirements 7](#_Toc367801583)

[JSON Resources Schema and Documentation 9](#_Toc367801584)

[3dSceneServiceInfo.js 9](#_Toc367801585)

[3dSceneLayerInfo.js 10](#_Toc367801586)

[3dNodeIndexDocument.js 12](#_Toc367801587)

[FeatureData.js 14](#_Toc367801588)

[SharedResources.js 17](#_Toc367801589)

[Textures.bin 18](#_Toc367801590)

[Geometry.bin 19](#_Toc367801591)

[Persistence 19](#_Toc367801592)

[Storage in CouchDB 19](#_Toc367801593)

[Bundled Indexed 3d Scenes 20](#_Toc367801594)

[Storage in IndexedDB 20](#_Toc367801595)

[Exploded Files layout 20](#_Toc367801596)

[Annex I: Client Access Patterns 20](#_Toc367801597)

[Annex II: Cooking a Store Patterns 22](#_Toc367801598)

[Main Workflow 22](#_Toc367801599)

[Partial updates & store versioning 23](#_Toc367801600)

# Requirements

The Esri Indexed 3d Scene (i3s) format and the corresponding package format (i3d) are specified to fulfill this set of requirements:

1. **User Experience first:** Support a very good user experience – high interactivity, fast display, rendering of visually relevant features first
2. **Scalability:** Support very large scenes, with global extent and a very large number of features (up to 1 billion), as well as very heavy features
3. **Reusability:** Be useable both as the delivery format of the ArcGIS Scene Service, the ArcGIS “MultiPatch” Feature Service and as a format stored in a local file or database
4. **Level of Detail:** Support discrete Level of Detail concepts for generalization of very large/heavy features and for “semantic” Level of Detail approaches
5. **Distribution:** Allow distribution of resources in very large data sets
6. **Merging:** Allow combination/merging with data from other scene data sets
7. **User-controllable symbology:** Support client-side symbology rendering
8. **Extensibility:** Be extensible to support new features (e.g. geometry types) and new platforms (e.g. by allowing definition of different materials/shaders)
9. **Web Friendliness**, i.e. easy to handle and parse by web clients by using JSON and current web standards
10. **Compatibility:** Have a single structure that is useable by all ArcGIS Desktop, Web and native apps, cross platform and cross device usage, map well to GL APIS
11. **Declarative**: limit how much specific knowledge on the client-side is needed for format support (e.g. Index generation method only needs to be known while writing the format)
12. **Follow REST/JSON API best practices:** “Hypertext as the Engine of Application State” – make all resources navigable using hrefs from relevant other resources.

Some of these requirements (especially 8, 9, 10 and 12) are shared with the Khronos glTF[[1]](#footnote-1) format, which is an upcoming standard for transferring 3D content. In this version of i3s, the two formats share the specification of Geometry Typed Arrays.

# The i3s Store – what goes into an Indexed 3D Scene?

The basic unit of an Indexed 3D Scene is a Store[[2]](#footnote-2), which contains individual resources (files) for a set of layers, index, geometries, textures and more. Within such a store, the i3s format supports a wide range of types of 2D and 3D content needed for 3D GIS scenes. All content types supported are listed in Table 1 on the next page.

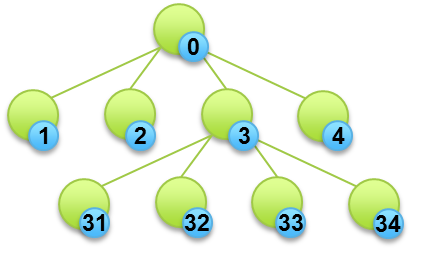
|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name *(example)*** | **Display Type** | **Fields** |
| 1 | Regular point array *(Grid)* | Triangles | No |
| 2 | Pre-Triangulated irregular point array *(TIN)* | Triangles | No |
| 3 | Integrated Mesh *(Acute3D)* | Triangles | Yes |
| 4 | Individual Feature Mesh *(Multipatch)* | Triangles | Yes |
| 5 | Point Cloud *(LAS)* | Points | Yes[[3]](#footnote-3) |
| 6 | Point Features *(GIS data)* | Points/Triangles | Yes |
| 7 | Line Features *(GIS data)* | Lines/Triangles | Yes |
| 8 | Polygon Features *(GIS data)* | Triangles | Yes |

Table 1: i3s Content Types supported in i3s

A single i3s store can contain data from multiple layers, but only one content type, as the different content types typically require different indexing and Level of Details methods to perform best. In many cases their schema also differs substantially. However, a single cache can contain multiple layers that share the same content type. Effectively these layers will share the same index, but they can still be accessed individually. This reduces the number of calls to a Scene Service, local database or the file system that need to be made by the client drastically and furthermore allows reducing the total data volume. In addition, layers in a shared stored can share resources, such as instance geometries.

# The Index Structure

Esri i3s is, as the name implies, an indexed, partitioned 3D Scene format with some similarities to regionated KML[[4]](#footnote-4) or X3D Earth. The purpose of any index is to allow fast access to (blocks of) relevant data. In an Indexed 3D Scene, the spatial extent is split into regions with a roughly equal amount of data in them, and an access data structure – the actual index – allows the client and the server to quickly discover which data the client actually needs. Such a region of a 3D Scene is called a ***Node***. Node creation is capacity driven – the smaller the node capacity is, the smaller the spatial extent of each node will be.

All Nodes have an ID that is unique throughout a cache. The ID format used is that of a treekey, i.e. the key directly indicates the position of the node in the tree. Treekeys allow sorting all resources on a single dimension and usually maintain 2D spatial proximity in the 1D ordering. The root node always gets ID “0”. All further nodes get keys according to the pattern shown in Figure 1. Please note that only 1..9 values are supported; thus, no indexing scheme can be used that assigns more than 9 children per parent.

The root node, inner nodes and leaf nodes may carry actual feature data. Inner nodes only carry feature data when features with highly differing spatial sizes are present or when Level of Detail (LoD) feature trees are used.

Figure 1: Tree Keys for a small Quadtree

The i3s format itself is agnostic to the specific indexing scheme used. Methods such as Quadtrees, Octtrees or R-Trees are equally supported. In the format, each node in the index declares all relationships to the parent, siblings and children in the index. Together with information on spatial extents and feature population, these structures allow a client to quickly discover where interesting data resides and to pick the data that is visually representative as well as sufficiently accurate.

# Level of Detail Concept

The Level of Detail concept introduced with this format specification covers several use cases, including splitting up very heavy features such as detailed buildings, very large features (coastlines, rivers, infrastructure), thinning/clustering for optimized visualization and semantic LODs, i.e. the usage of explicit, authored representations to be used for different viewing ranges.

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Definition** | **Examples** |  |
| *Discrete* | *Multiple representations*, a more detailed one fully replaces a co­arser representation | Image Pyramid, Height map pyramid, Line/Polygon Generalization | http://ggg.udg.edu/skylineEngine/docs/icons/userFriendlyBuildingGraphs.png |
| *Continous* | *Single representation* that is refined continuously | SMTerrain, TVTerrain, BitLOD, Progressive Meshes |  |
| *Semantic* | *Independent models* for the same feature | CityGML | http://www.cadmagazine.nl/dir_upload/site/a0799b3c8c7d394b284b80ea5cf40253/101/d2889541_BIMstandaard096-03.gif |

Table 2: LoD concepts support in i3s

Thus the format support for LoD is rather abstract. i3s implements a feature-based Level of Detail approach, i.e. each feature in a node can have higher-detail or lower-detail representations. This feature-based approach has the following properties:

* A Feature can participate in a so-called LoD tree.
* A LoD tree has a single root feature.
* Each non-leaf participating Feature has a set of n lodChildren.
* The client receives LoD information with the Node Index Document already, allowing making a choice whether to load a certain resource or not, and also to later identify which features replace which other features.
* lodChildren are guaranteed to be either in the same node or in a direct child node.
* Each Feature that participates in a LoD tree has a rank from 1..d.
* Each Feature that participates in a LoD tree and has a rank > 1 has a rootFeature reference. This reference enables the client to detect which features represent a single object, e.g. for picking purposes.

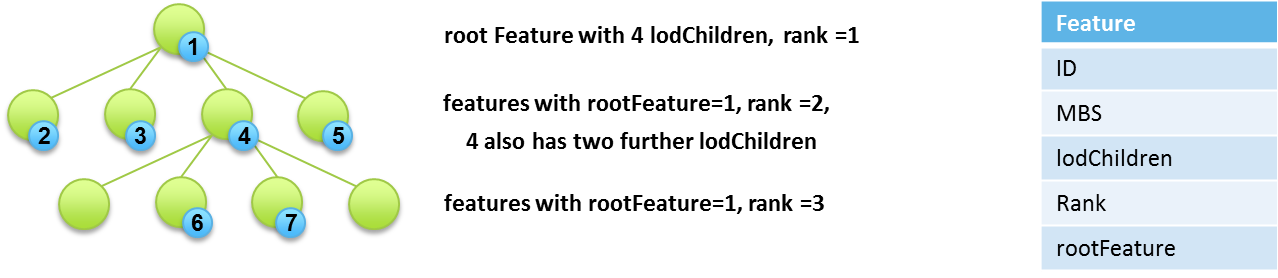
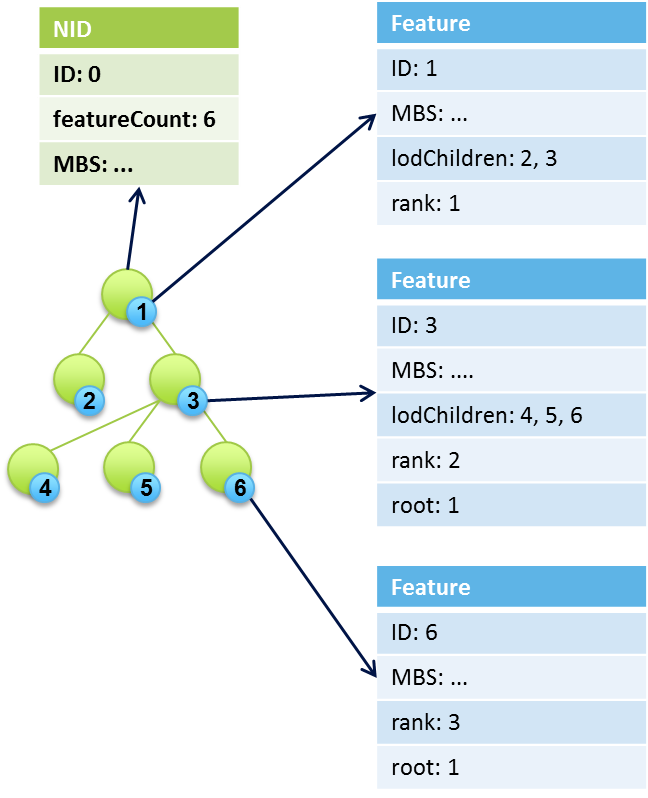


Figure 2: Structure of a Feature LoD tree

In cases where terrain data or integrated meshes are encoded in i3s, it is expected to have a single feature per node, with LoD children in the direct descendants, filling up the entire index with representations. The following figure shows an example of this:

In this example, from root to leaf nodes, each node carries a single feature, for a total count of six nodes and six features. This is typically the case with integrated meshes. Each of the features that is not a in a root node has a set of lodChildren, with the same set size as the number of node children.

The links between all features participating in a LoD tree are either created during the cache creation process, e.g. by breaking down a heavy and large feature, or they are predefined by the data provider, as it is the case with Integrated Meshes (Acute3D) data.

# Coordinate Reference Systems

Indexed 3D Scenes have to fulfill several in part conflicting requirements when it comes to the selection of spatial reference systems to use:

* Minimal reprojection on client side needed (such as “bending” of large features to the ellipsoid from a projected CRS to an internal geocentric CRS)
* Support true global extent data sets
* Ensure consistency between nodes of a single layer
* Ensure consistency of positions across multiple layers, potentially created from different source projections
* Render easily in Planar (Projected Cartesian) and Globe (Geocentric Cartesian) modes
* Support local data with very high positional accuracy
* Support global data sets with high positional accuracy
* Vertex positions have to be meter-based since CityEngine is meter-based

To match these requirements, the following approach is taken for this version of the format:

1. **Metadata**: We use a single, global (bounds -180.0000, -90.0000, 180.0000, 90.0000) Geographic CRS for horizontal location. Elevation and node minimum bounding sphere radius are given in meters. Allowed EPSG codes:
   1. EPSG:4326 (WGS84)
2. **Vertex Positions**: We allow the use of projected CRS, with meter-based x,y,z axes and with a per-node offset (from the center point of the node’s minimum bounding sphere) and using the WGS84 datum. Allowed EPSG codes:
   1. EPSG:32601 to EPSG:32660, EPSG:32701 to EPSG:32760 (UTM WGS84)
   2. EPSG:3857 (Web Mercator WGS84) or EPSG:32662 (Plate Carree WGS84) for large extent datasets (~12° to 360° horizontal extent)
3. **Axis Order:** All positions, independent of the used geographic or projected CRS, use the Easting, Northing, Elevation (x,y,z) axis order. The Z axis points upwards towards the sky.

# Structure of i3s resources

The i3s format contains different components – node index documents (NIDs), feature data, textures, geometry and resources shared across features of a given node. Feature data, textures, geometry and shared resources are all called resources and are always attached to a node.

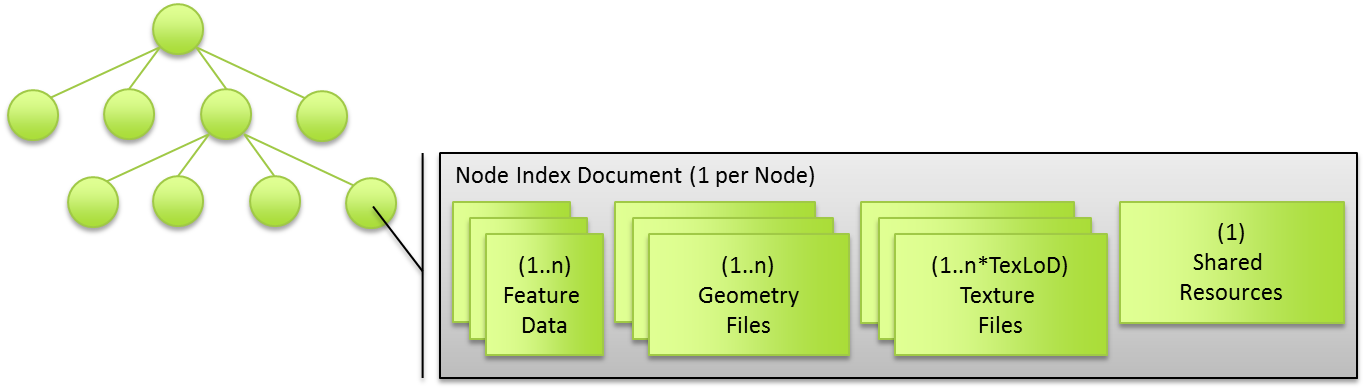
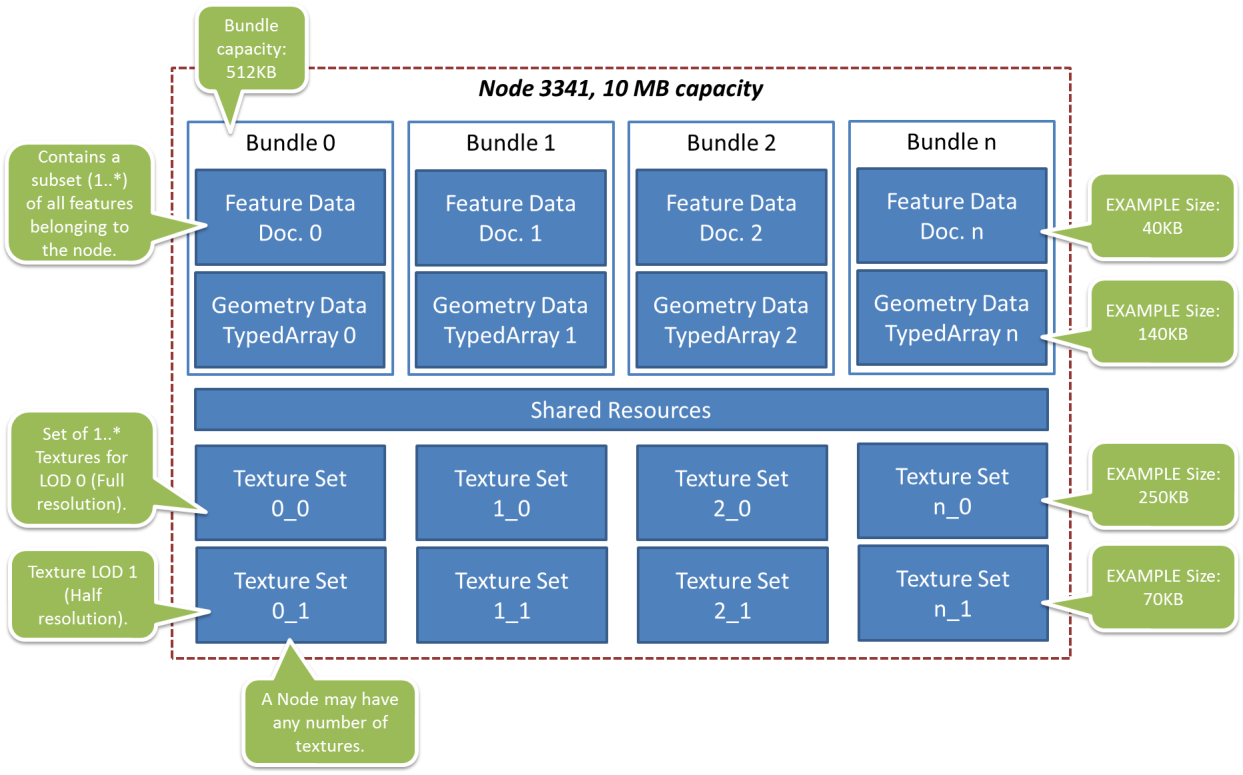


Figure 3: Structure of a single Node and its attached Resources

Per node, there is exactly one Node Index Document and one Shared Resource. Feature Data, Geometry files and Texture files are split into bundles for optimal network transfer and client-side reactivity. This allows balancing between index size, feature splitting (with a relatively large node capacity between 1MB and 10MB) and optimal network usage (with a smaller bundle size, usually in the range of 64kB to 512kB).

There are always an equal number ***n*** of Feature Data and Geometry resources, and each set contains the corresponding data elements to be able to render a complete feature. Textures are not tightly coupled to bundles due to the fact that they can also be in the node as part of a shared resource that bubbled up. For each texture (atlas) in the Node (***m***), the number of Texture resources created is then equal ***m\*Texture LoD Steps***. The following figure illustrates an example set of bundles within a node:



## Texture Recommendations and Requirements

Especially for Web and Mobile clients, the number of textures and their volume is the limiting factor in how much data can be displayed at any given time, Thus, this specification provides several recommendations and requirements on texture resources that are delivered as part of an Indexed 3D Scene.

### Image Formats

At the current stage, two texture formats are recommended and will be supported by Esri Clients – JPEG for RGB and PNG for RGBA. They were chosen over S3TC because of low bandwidth consumption and widespread adoption in all steps of the toolchain, such as supporting alpha transparency rendering in WebGL. However, with more wide-spread client support for next-generation texture compression formats such as ASTC[[5]](#footnote-5) and BPTC[[6]](#footnote-6) will become available later on.

### Texture Sets

While this specification allows the combination of multiple textures into a single resource by using array buffer views, we generally recommend to use large atlases (e.g. 2048x2048px) and then to use exactly one texture per bundle.

### Atlas usage

Individual textures should be aggregated into texture atlases, where they become subtextures. As all texture resources, the atlas has to be 2n-sized on both dimensions, with n being in the range [3,16]. Width and height dimensions do not have to be equal, e.g. 512px x 256px. Subtextures contained within an atlas also need to be 2n-sized, with n being in the range [3,16]. Otherwise if their width or height dimension is not 2n, border artifacts are likely to appear when filtering or MIP-mapping. If source subtexture dimensions do not match this requirement, they need to be padded (with nearest/interpolated pixels) or scaled to the nearest lower 2n size. An image that is 140px x 90px would thus be rescaled to 128px x 64px before being inserted into the atlas or padded to 256px x 128px.

The pixels belonging to a subtexture are identified by the "subimageRegion": [0, 0, 0.5, 0.5] attribute. An atlas may have a maximum of 256 subimageRegions; the reason for this limitation is the amount of information that can be passed to the shader when not embedding them in the texture itself.

Implementation Hint: Region information can be passed on to WebGL and other APIs using Uniform Arrays[[7]](#footnote-7) and can be encoded in a 32bit Float per region using the following pattern:

* anchor x: 12 bit, value is 16 \* n, range of n: [1,4096], values: [16, 32, 48, 64, , …, 65536]
* anchor y: 12 bit, value is 16 \* n, range of n: [1,4096], values: [16, 32, 48, 64, , …, 65536]
* width: 4 bit, value is 2n, range of n: [1,16], values: [2,4,8,16,32,…,65536]
* height: 4 bit, value is 2n, range of n: [1,16] , values: [2,4,8,16,32,…,65536]

### Texture coordinates

Texture coordinates do not take atlas regions into account directly. They always range from 0…1 in U and V, except when using the “repeat” wrapping mode, where they may range from 0…n (n being the number of repeats). The client is expected to use the subimageRegion values and the texture coordinates to best handle repeating textures in atlases. This approach has been selected since client capabilities in dealing with more complex UV cases vary greatly.

# JSON Resources Schema and Documentation

This section provides a detailed, logical-level specification for each of the resource types.

## 3dSceneServiceInfo.js

The 3dSceneServiceInfo file is a JSON file that describes the capability and data sets offered by an instance of a 3dSceneService instance.

The 3dSceneServiceInfo has the following structure:



Figure 4: Logical schema of the 3dSceneServiceInfo document

This file is not generated by the authoring tools and is not part of a Bundled i3s file. It is generated solely by the 3D Scene Server for each service instance. Its description is contained here only for reference.

### Class 3dSceneServiceInfo

The 3dSceneServiceInfo is the major object in the 3dSceneServiceInfo document. There is always exactly one 3dSceneServiceInfo object in the document, which describes a running SceneService instance.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| serviceName | String | The type of the service; always SceneService. |
| serviceVersion | String | The version of the service protocol/REST endpoint. |
| supportedBindings | String[1..\*] | the list of bindings, should we ever need to add new bindings in addition to the REST binding initially supported |
| supportedOperations | String[1..3] | Supported profiles of the service from the choice {"Base", "Dynamic", "Editing"}. |

Table 3: Attributes of the Class 3dSceneServiceInfo within the 3dSceneServiceInfo document

### Class Layer

In the 3dSceneServiceInfo document, the Layer object provides a reference so that clients know which Layers are served by a given service.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | The ID of this layer, unique within a 3dSceneService. |
| name | String | The version of the service protocol/REST endpoint. |
| alias | String[0..1] | An optional display alias for the layer. |
| geometryType | String | The geometry type of the cache; selected from {FeatureMesh, IntegratedMesh, RasterTerrain, TINTerrain, Point, Line, Polygon, PointCloud}. |
| href | URL | The relative URL to the Layer resource giving full information on the Layer's schema and drawing info |

Table 4: Attributes of the Class Layer within the 3dSceneServiceInfo document

## 3dSceneLayerInfo.js

The 3dSceneLayerInfo file is a JSON file that describes the properties of a single layer in a store, including the default symbology to use. It shares the definition of this default symbology with the web scene item JSON file. The 3dSceneLayerInfo has the following structure:



Figure 5: Logical schema of the 3dSceneLayerInfo document

### Class 3dSceneLayerInfo

The 3dSceneLayerInfo is the major object in the 3dSceneLayerInfo document. There is always exactly one 3dSceneLayerInfo object in the document, which describes a Layer.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | Unique numeric ID of the Layer. |
| version | string | The ID of the last update session in which any resource belonging to this layer has been updated. |
| name | string | The name of this layer. |
| alias | string[0..1] | The display alias to be used for this layer. |
| description | string[0..1] | Description string for this layer. |
| copyrightText | String[0..1] | Copyright and usage information for the data in this layer. |
| capabilities | String[1..3] | Capabilities from the Set {View, Query, Edit} that are possible on this layer. If not served by a 3DSceneServer (e.g. exported by CE), "View" only. |

Table 5: Attributes of the Class Node within the 3dSceneLayerInfo

### Class Store

While Layers are the user-visible entry point to the 3dSceneServer resources (for web scene authoring and viewing), internally the service uses so-called stores. A store can contain 1…\* layers, which will share a common index and set of nodes, as well as resources. The Store object describes the exact physical storage of a Layer and enables the client to detect when multiple Layers are served from the same Store. Storing multiple layers in a single store – and thus having them share resources – enables efficient serving of many layers of the same geometryType, but with different attribute schemas or different symbology applied.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | UUID | A Store ID, unique across a SceneServer. Enables the client to discover which layers a part of a common store, if any. |
| rootNode | URL | relative URL to root node resource. |
| version | String | Format version of this resource; used here again if this store hasn't been served by a 3D Scene Server. |
| geometryType | String | The geometry type of this store; selected from {FeatureMesh, IntegratedMesh, RasterTerrain, TINTerrain, Point, Line, Polygon, PointCloud} |
| extent | Double[4] | The 2D spatial extent (xmin, ymin, xmax, ymax) of this store, in the horizontal geographicCRS |
| geographicCRS | URL | The horizontal CRS used for all minimum bounding spheres (mbs) in this store, identified by an OGC URL. |
| projectedCRS | URL | The horizontal CRS used for all "vertex positions" in this store, identified by an OGC URL. |
| nidEncoding | MIMEType | MIME type for the encoding used for the Node Index Documents; format: application/vnd.esri.i3s.json+gzip; version=1.1 |
| featureEncoding | MIMEType | MIME type for the encoding used for the Feature Data Resources |
| geometryEncoding | MIMEType | MIME type for the encoding used for the Geometry Resources |
| textureEncoding | MIMEType | MIME type for the encoding used for the Texture Resources |
| indexingScheme | String | Indexing Scheme used; selected from {esriRTree, QuadTree, AGOLTilingScheme} |
| featureOrdering | String[1..3] | Ordered list of keywords indicating the ordering scheme applied to sort features within a node; selected from {ID, Prominence, Layer} |

Table 6: Attributes of the Class Store within the 3dSceneServiceInfo document

### Class Field

The Field class is used to provide schema information for this Layer.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| name | string | The name of the field. |
| type | string | The type of the field, from this enum: {esriFieldTypeBlob, esriFieldTypeDate, esriFieldTypeDouble, esriFieldTypeGeometry, esriFieldTypeGlobalID, esriFieldTypeGUID, esriFieldTypeInteger, esriFieldTypeOID, esriFieldTypeRaster, esriFieldTypeSingle, esriFieldTypeSmallInteger, esriFieldTypeString, esriFieldTypeXML} |
| alias | string[0..1] | The display alias to be used for this field. |

Table 7: Attributes of the Class Node within the 3dSceneLayerInfo

### Class IndexScheme

The IndexScheme class declaratively describes computational and structural properties of the index used within an i3s store. This information can be used by clients to better understand how to work with the index.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| name | string | The name of the field. |
| inclusive | boolean | true indicates that the extent and mbs of all children nodes is fully within their parent nodes' extent/mbs |
| dimensionality | int | The number of dimensions in which this index differentiates. |
| childrenCardinality | int[2] | min/max number of children per node. |
| neighborCardinality | int[2] | min/max number of neighbors per node. |

Table 8: Attributes of the Class Node within the 3dSceneLayerInfo

### Class WebCimDrawingInfo

WebCimDrawingInfo and the associated classes contain the default symbology for this Layer. This part matches the WebCIM (Cartographic Information Model) as used by the other Esri REST services.

## 3dNodeIndexDocument.js

The 3dNodeIndexDocument JSON file describes a single index node within a store, with links to other nodes (children, sibling, and parent), metadata such as its spatial extent and optionally a list of features that the node contains. While this makes a relatively heavy node document, it also means clients have a rich set of information to use to further decide which data to retrieve. The features list already provides sufficient data for simple visualization by rendering the centroids as point features or the Minimum Bounding Sphere as spheres.

The 3dNodeIndexDocument has the following structure:



Figure 6: Logical schema of the 3dNodeIndexDocument

### Class Node

The Node is the major object in the 3dNodeIndexDocument. There is always exactly one Node object in a 3dNodeIndexDocument.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | Tree Key ID, unique within the store. ID 0 is always the root node. |
| level | int | Explicit level of this node within the index tree. |
| version | UUID | The version (store update session ID) of this node. |
| mbs | double[4] | An array of four doubles, corresponding to x, y, z and radius of the minimum bounding sphere of a node. |
| precision | double | The "epsilon" value for a node; i.e. the maximum error introduced through generalization of features, relative to the diameter of the MBS of this node. For all nodes/stores without LOD features, this value can be 0.0. |
| created | timestamp[0..1] | Creation date of this node in UTC. |
| expires | timestamp[0..1] | Expiration date of this node in UTC. |
| transform | Double[16] | 3D (4x4) transformation matrix expressed as a linear array of 16 values. |

Table 9: Attributes of the Class Node within the NodeIndexDocument

### Class NodeReference

A NodeReference is a pointer to another node – the parent, a child or a neighbor. NodeReferences contain a relative URL pointing to the referenced NID, as well as a set of metainformation that can be used by the client to determine whether to load that node or not, as well as maintaining store consistency.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | Tree Key ID of the referenced node. |
| mbs | double[4] | An array of four doubles, corresponding to x, y, z and radius of the minimum bounding sphere of the referenced node. |
| href | string | The relative URL to the referenced node resource. |
| version | UUID | Version (store update session ID) of the referenced node. |
| featureCount | int | Number of features in the referenced node and its descendants, down to the leaf nodes. |

Table 10: Attributes of the Class NodeReference within the NodeIndexDocument

### Class Resource

Resource objects are pointers to different types of resources related to a node, such as the feature data, the geometry attributes and indices, textures and shared resources.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| href | string | The relative URL to the referenced resource. |
| layerContent | string[1..\*] | The list of layer names that indicates which layer features in the bundle belongs to. The client can use this information to selectively download bundles. |

Table 11: Attributes of the Class Resource within the NodeIndexDocument

### Class Feature

Resource objects are pointers to different types of resources related to a node, such as the feature data, the geometry attributes and indices, textures and shared resources.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | A ID of the Feature, unique within the store (important to note when using Features from multiple stores!) |
| mbs | double[4] | An array of four doubles, corresponding to x, y, z and radius of the minimum bounding sphere of the referenced node. |
| lodChildren | long[0..\*] | IDs of Features in a lower LOD level, which can replace this feature when loaded. |
| rank | int[0..1] | The LOD level of this feature. Only required for Features that participate in a LOD tree and are not root features of that LOD tree. |
| root | long[0..1] | The ID of the root node of a feature LOD tree that this feature participates in. Only required if the feature participates in a LOD tree. |

Table 12: Attributes of the Class Feature within the NodeIndexDocument

## FeatureData.js

The FeatureData JSON file(s) contain geographical features with a set of attributes, accessors to geometry attributes and other references to styling or materials.

Features have the following structure:



Figure 7: Logical schema of the FeatureData document

### Class Feature

A Feature is a single object within a GIS data set, usually representative of a feature present in the real, geographic world.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | long | Feature ID, unique within the store. |
| position | double[2] | An array of two doubles, giving the x,y (easting/northing) position of this feature's minimum bounding sphere center, in the projectedCRS. |
| pivotOffset | double[3] | An array of three doubles, providing an optional, "semantic" pivot offset that can be used to e.g. correctly drape tree symbols. |
| mbb | double[6] | An array of six doubles, corresponding to xmin, ymin, zmin, xmax, ymax and zmax of the minimum bounding box of the feature, expressed in the projectedCRS, without offset. The mbb can be used with the Feature’s Transform to provide a LOD0 representation without loading the GeometryAttributes. |
| layer | string | The name of the Feature Class this feature belongs to. |

Table 13: Attributes of the Class NodeReference within the FeatureData document

### Class FeatureAttribute

A FeatureAttribute is a field carrying a value.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| name | String | The name of the attribute. |
| value | String | The value of the attribute. |

Table 14: Attributes of the Class FeatureAttribute within the FeatureData document

### Class Geometry

Objects of this type include the Geometry definition. Note that one Feature can have multiple geometries, which can consist of multiple components.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| type | String | The type denotes whether the following geometry is defined by using array buffer views (ArrayBufferView), as a reference to a shared Resource (SharedResourceReference) or embedded (Embedded). |
| transformation | double[16] | 3D (4x4) transformation matrix expressed as a linear array of 16 values. |

Table 15: Attributes of the Class FeatureAttribute within the FeatureData document

### Class Component

Component objects provide information on parts of the geometry they belong to, specifically with which material and texture to render them.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | Long | The ID of the component, only unique within the Geometry |
| materialID | UUID | ID of the material, as defined in the shared resources bundle, to use for rendering this component |
| textureID | Long[0..1] | Optional ID of the texture, as defined in shared resources, to use with the material to render this component |
| regionID | Long[0..1] | Optional ID of a texture atlas region which to use with the texture to render this component |

Table 16: Attributes of the Class Component within the FeatureData document

### Class GeometryAttribute

Each GeometryAttribute object is an accessor, i.e. a view, into an arraybuffer. There are two types of GeometryAttributes – VertexAttributes and FaceAttributes. While the first describe properties that are valid for a single vertex, the second are used to describe faces and other structures by providing a set of indices. As an example, the faces.position index attribute is used to define which vertex positions make up a face.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| byteOffset | long | The starting byte position where the required bytes begin |
| count | long | The number of elements. Multiply by number of bytes used for valueType to know how many bytes need to be read. |
| valueType | String | The element type, from {UInt8, UInt16, Int16, Int32, Int64 or Float32, Float64} |
| valuesPerElement | short | The number of values need to make a valid element (such as 3 for a xyz position) |
| componentIndices | int[0…\*] | An optional array that indicates how many of the elements in this view belong to the first, second and consecutive components of the geometry. The number of entries in this array, when present, has to be equal to the number of entries in the components List of the enclosing Geometry object. The entire field is optional when no components have been declared for this Geometry. |

Table 17: Attributes of the Class FeatureAttribute within the FeatureData document

## SharedResources.js

Shared resources are models or textures that can be shared among features within the same store. They are stored as a JSON file entirely, comparable to the encoding used for geometry and textures in a 3ws 2.2 file. Each node has a shared resource which contains materials and symbols used by more than a single feature in that node or in features which are stored in the subtree of the current node. This approach ensures an optimal distribution of shared resources across nodes, while maintaining the node-based updating process.



Figure 8: Logical schema of the FeatureData document

### Class SharedResource

The SharedResource class collects Material definitions, Shader definitions and geometry symbols that need to be instanced.

### Class Material

MaterialDs describe how a Feature or a set of Features is to be rendered. This includes which shading and which colors to use. The following table provides the set of attributes and params for the “type”:“Standard” material.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| name | String | A name for the material as assigned in the creating application. |
| type | String | Indicates the material type, chosen from the supported values {Standard, Water, Leafcard, Billboard} |
| params.vertexColors | Boolean | Indicates whether this Material use Vertex Colors. |
| params.transparency | Float | Indicates whether the transparency of this material; 0 = opaque, 1 = fully transparent. |
| params.reflectivity | Float | Indicates reflectivity of this Material. |
| params.shininess | Float | Indicates shininess of this Material. |
| params.ambient | Float[3] | Ambient color of this Material. |
| params.diffuse | Float[3] | Diffuse color of this Material. |
| params.specular | Float[3] | Specular color of this Material. |
| params.renderMode | String | Rendering mode, any one of {textured, solid, untextured, wireframe} |

Table 18: Attributes of the Class Material within the SharedResources document

### Class Texture

A Texture is a set of images, with some parameters specific to the texture/uv mapping to geometries.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| encoding | MIMEtype | The encoding/content type that is used by all images in this map |
| wrap | String[2] | UV wrapping modes, from {none, repeat, mirror} |
| atlas | Boolean | true if the Map represents a texture atlas. |
| uvSet | String | The name of the UV set to be used as texture coordinates. |
| Channels | String | indicates which channels are stored in which channel of this map. Possible values: r=red, g=green, b=blue, a=alpha, r=bump, d=displacement, ... |

Table 19: Attributes of the Class Texture within the SharedResources document

### Class Image

An image is a binary resource, containing a single raster that can be used to texture a feature or symbol. It represents one specific texture LoD.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | string | A unique ID for each image. Generated using the BuildID function that is documented in the spec. |
| size | int | x size of this image. |
| pixelInWorldUnits | float | maximum size of a single pixel in world units (used by the renderer to pick the image to load/map) |
| href | URL | The href to the image. The resource ID is identical to the id. |
| byteOffset | long[0..1] | The byte offset of this image in the block in which this texture image resides. |
| length | long[0..1] | The length in bytes of this image. |

Table 20: Attributes of the Class Image within the SharedResources document

The Id of an image is generated using the following method:

UInt64 BuildID(LONG id, int w, int h , int l, int al)

{

   UInt64 l\_al = ((UInt64)al)<<60;

   UInt64 l\_l = ((UInt64)l)<<56;

   UInt64 l\_w = ((UInt64)w)<<44;

   UInt64 l\_h = ((UInt64)h)<<32;

   UInt64 id64 = l\_al + l\_l + l\_w + l\_h + (UInt64)id;

   return id64;

}

### Usage syntax

UInt64 BuildID(Long id, int w, int h , int l, int al);

### Parameters

|  |  |
| --- | --- |
| id | Index of the texture in the cache, start from 1 |
| w | Width of the texture |
| h | Height of the texture |
| l | Index of the level that the texture belong to, start from 0 |
| al | Level count of the texture |

### Class ShaderDefinition

ShaderDefinitions are, in this version of the i3s specification, an optional feature to provide API-dependent shader programs with a layer.

### Class Symbol

For Symbols, the same model is used as in the FeatureData Geometry.

## Textures.bin

The Textures file is a binary resource that contains one or multiple images that are used as textures of features in the store. A single Texture.bin file contains 1…n textures for a single specific texture LoD. It can contain a single texture atlas or multiple individual textures; the decision how this is bundled is left to the authoring application so that specific aspects of the materials and textures used can be taken into account, such as tiling. For recommendation on texture handling, please refer to the section “Texture Recommendations and Requirements”.

## Geometry.bin

The binary geometry attribute file follows the Khronos Typed Array[[8]](#footnote-8) specification in the Editor’s Draft version of 10th April 2013. Citing the overview of that spec:

“*This specification defines an ArrayBuffer type, representing a generic fixed-length binary buffer. It is not possible to manipulate the contents of an ArrayBuffer directly. Instead, a group of types are used to create views of the ArrayBuffer. For example, to access the buffer as an array of 32-bit signed integers, an Int32Array would be created that refers to the ArrayBuffer.*

*Multiple typed array views can refer to the same ArrayBuffer, of different types, lengths, and offsets. This allows for complex data structures to be built up in the ArrayBuffer. As an example, given the following code:*

// create an 8-byte ArrayBuffer

var b = new ArrayBuffer(8);

// create a view v1 referring to b, of type Int32, starting at

// the default byte index (0) and extending until the end of the buffer

var v1 = new Int32Array(b);

// create a view v2 referring to b, of type Uint8, starting at

// byte index 2 and extending until the end of the buffer

var v2 = new Uint8Array(b, 2);

// create a view v3 referring to b, of type Int16, starting at

// byte index 2 and having a length of 2

var v3 = new Int16Array(b, 2, 2);

*This defines an 8-byte buffer b, and three views of that buffer, v1, v2, and v3. Each of the views refers to the same buffer -- so v1[0] refers to bytes 0..3 as a signed 32-bit integer, v2[0] refers to byte 2 as a unsigned 8-bit integer, and v3[0] refers to bytes 2..3 as a signed 16-bit integer. Any modification to one view is immediately visible in the other: for example, after v2[0] = 0xff; v2[1] = 0xff; then v3[0] == -1 (where -1 is represented as 0xffff).”*

# Persistence

All storage methods store the Indexed 3D Scene in a simple key-value structure, with the key representing the access URL and the value being the JSON document or other resource type.

## Storage in CouchDB

The 3d Scene Server stores i3s resources in a document-oriented database. After testing, CouchDB was selected as being suitable. Especially large stores benefit from this type of storage.

## Bundled Indexed 3d Scenes

A store can also be delivered as a single file. Such a file takes all the resources and their attachments and stores them as entries in a MIME/multipart file. This format was selected because it is well-known, there is robust support for all languages and clients (browsers included) and it is also used as a ‘dump’ file format for CouchDB. Consequently, a *Bundled Indexed 3D Scene* can be loaded to CouchDB (the scene service storage backend) easily.

## Storage in IndexedDB

*IndexedDB* is a Key-value document store available in many current browsers, such as Firefox, Chrome and Internet Explorer[[9]](#footnote-9). IndexedDB offers a method of storing data client-side and allows indexed database queries against JSON documents.

In IndexedDB, each resource is stored separately.

## Exploded Files layout

All resources are stored as individual files on the file system, using a folder structure aligned with the index structure to keep the number of folder and files per folder on a manageable level.

This part of the specification is currently not maintained as it has been superseded by CouchDB and Bundled i3s storage.

# Annex I: Client Access Patterns

This section describes how a client is expected to load and handle resources from an Indexed 3D Scene. The general pattern consists of these phases:

1. Handshake & capabilities negotiation – ensure that the service has the expected resources, that client and server have a common set of formats for these that can be used and inform the client which capabilities can be used, such as editing. Within this phase, the client utilizes the following resources:
   1. *SceneServiceInfo*: General service information
   2. *3dSceneLayer*: Information on available layers, including symbology and encoding
2. Identify suitable entry points into the Indexed 3D Scene data – in this optional phase, the client asks a service to identify the node covering the clients’ Area of interest best. In this phase, none of the i3s resources are utilized, only an operation that the scene server provides:
   1. *FindNode* – Find the Node that best matches the client’s AOI.
3. Retrieve index – either starting from the root node or from the identified node, the client retrieves Node Index Documents and decides – based on properties such as the precision value and the feature locations in the node – whether it wants to download and render their attached resources. Within this phase, the client utilizes the following resource:
   1. *NodeIndexDocument*: Summary of the content of a single node of the index, references children, parent and neighbor nodes, indicating what can be found there
4. Retrieve resources – when a client has decided that it wants to render the content of a node, it starts to retrieve the attached resources:
   1. *FeatureData*: Attributes of GIS features, accessors to GeometryData and TextureData
   2. *SharedData*: Material defintions, shared geometries for instancing
   3. *GeometryData*: Geometry attributes such as positions and indices
   4. *TextureData*: Images used as texture maps

The retrieval of resources belonging to a node should happen as outlined in the activity diagram shown in Figure 8. Whenever a FeatureData bundle, the associated Geometry and the Textures are retrieved, they can be rendered by the client. 

Figure 8: UML activity diagram showing the retrieval of i3s resources.

Figure 8 also shows one aspect of the handling of LoD children in the resource loading process: When a feature that had previously already been sent to the rendering pipeline participates in a LoD tree, it can be replaced as soon as all its LoD children have been loaded. Figure 9 provides the context for this operation, also as an UML activity diagram. When LoD trees are used in an i3s store, the general pattern is extended to check and replace features with their coarser or finer representations:



Figure 9: UML activity diagram showing LoD feature replacement.

# Annex II: Cooking a Store Patterns

This format has been designed with fully distributable encoding in mind, i.e. all phases needed to “cook” typical GIS data such as features in a geodatabase into i3s resources can be run in parallel by as many workers as preferred. This section explains the general pattern of how to do this, so that even very large data sets can be “cooked” in acceptable time.

## Main Workflow

The general sequence of how to turn an arbitrary set of input resources into a single i3s store is as follows:

1. Read data from all required input formats and reproject if necessary
2. Analysis (done per feature)
   1. Calculate per-feature data volume
   2. Split up and/or generalize features that exceed the node capacity thresholds
   3. Create MBS and MBB for all features
   4. Calculate absolute feature prominence (if chosen in the featureOrdering parameter)
3. Calculate Data Density distribution across total extent
4. Indexing (done on MBS + FeatureID + Feature Size only)
   1. Create neighborhood partitioning based on initial data density analysis
   2. Create spatial index per neighborhood
   3. Create common index or fit into tiles
5. Identify shared resources and attach them to the index (done per node)
6. Cache Encoding (done per node)
   1. Order features in a single node/neighborhood by layer, ID or prominence
   2. Encode features and group into bundles
   3. Create texture bundles/atlases
   4. Write Node Index Document and shared resources bundle

The only step that cannot be fully distributed is 4.3, but even for large datasets with millions of features, this step, which combines multiple subtrees to a single index, should take only seconds[[10]](#footnote-10).

## Partial updates & store versioning

Another core consideration of the i3s format was that partial updates need to be possible. It would be really problematic if, whenever a new building is inserted into a large city, the whole i3s store would have to be regenerated. To counter this, **the transactional unit of an i3s store is always a node including its resources**; i.e. when a single feature’s texture is updated, that Node’s is considered modified and gets a new version. The same is true when an entire feature is inserted, deleted or updated: at least the node containing the feature has to be updated. In many cases such as the modification of non-geometric attributes, this update doesn’t affect the actual index structure – only the appropriate FeatureData resources need to be updated. Only when the update is so large that capacity bounds of a Node are violated, the index needs to be modified.

### Modifying the Index

If a new feature is inserted and the affected node’s capacity isn’t sufficient to contain the new feature, at least two nodes (depending on the index properties) are affected: The original node, plus one or more new child nodes that are created and into which some of the original node’s feature are passed on. The opposite thing happens when features are removed from a node; at a given capacity threshold of the affected node, the node is removed and its remaining features, if there are any, are moved up or to a neighbor, extending the neighbor’s extent.

One challenge is the updating of shared resources. It is not expected to recalculate the position of a shared resource after each partial update to a store automatically, which means that after a long series of partial updates, there may be orphaned shared resources or even duplicates. For this purpose we recommend having a specific tool that can be run asynchronously to actual cache updates to recalculate shared resource positions for a store.

### Store Update Sessions

In any case, each resource in the store is versioned – both explicitly in the NodeIndexDocument and implicitly by CouchDB. While the internal CouchDB versions are created automatically whenever an individual resource in CouchDB is modified, the versions used in the NodeIndexDocuments are **store update session IDs and have a much larger scope than one document**. Such a store update session ID is generated whenever the cooker initiates an update process and is used to mark all nodes affected by the respective update.

A client needs to quickly see when it might have stale data. For this purpose, the node index documents contain two sets of information – expiration timestamps to indicate when a dataset is expected to change and versioned node references that link a node to a defined version (as in created or updated in a given store update session) of its parent, children and neighbor nodes:

"parentNode": {

"id": 5, // the ID of the parent node.

"version": "ee4fbf04-e882-444e-854d-cd519b68594a",

"href": "../5",

"mbs": [122.2, 39.9, 421.0, 5462.1],

"featureCount": 4

}

The client now only needs to compare the expected store update session ID to the store update session ID indicated by the referenced Node; if it is not equal, the older Node needs to be refreshed. This is the sequence of steps the store creator and a client needs to go through when a partial update happens:

|  |  |
| --- | --- |
| **Store Cooker** | **Client** |
|  | Load NID 3, version abc, references child NID 31, version def |
| Load NID 31, insert new or update Feature in NID 31 + attached resources |  |
| Update NID 31 version to efg |  |
| Update NID 31 reference in NID 3 to version efg |  |
|  | Load Node 31, find that it’s not version def as expected from the reference |
|  | Reload Node 3, stop since it’s still version abc |

1. Please refer to <https://github.com/KhronosGroup/glTF/> for the current state of the specification. glTF is currently work-in-progress. [↑](#footnote-ref-1)
2. In I3S versions prior to 1.0, this was called a “Cache”. [↑](#footnote-ref-2)
3. Fields in Point clouds are limited as follows: Per point: numeric only, per cloud: all types allowed [↑](#footnote-ref-3)
4. See <http://code.google.com/p/regionator/wiki/Welcome> for more information. [↑](#footnote-ref-4)
5. Please refer to <http://en.wikipedia.org/wiki/Adaptive_Scalable_Texture_Compression> [↑](#footnote-ref-5)
6. Please refer to <http://www.g-truc.net/post-0340.html> [↑](#footnote-ref-6)
7. Please refer to <http://jsperf.com/webgl-uniform-uploading/6> [↑](#footnote-ref-7)
8. Please refer to <http://www.khronos.org/registry/typedarray/specs/latest/> for the latest version of this specification. [↑](#footnote-ref-8)
9. Please refer to <http://caniuse.com/#feat=indexeddb> for the current state of support. [↑](#footnote-ref-9)
10. Please refer to papers on distributed spatial index generation such as *Parallel bulk-loading of spatial data with MapReduce: An R-tree case*, Y Liu, N Jing, L Chen, H Chen - Wuhan University Journal of Natural Sciences, 2011 – Springer. If the used index is a simple quadtree or tiling scheme, this step is even less problematic. [↑](#footnote-ref-10)