NOODLES V0.3: A PROTOCOL SPECIFICATION

NICHOLAS BRUNHART-LUPO

Contents

1. Introduction

This document entails a specification for a distributed scene-graph wireline protocol suitable as a substrate for shared interactive visualizations. It also lays out concepts for the supporting implementations that would provide such visualizations.

2. Rationale & Design Goals

- The intent of this document is simplicity, to get a working version implemented so that further improvements can be identified.
- The structure here is not intended to mirror the use-case of the HTML DOM + Javascript where code is shipped to clients. That would be restrictive, as it requires the clients either interpret or compile and run code on command. Some clients, such as integrated head mounted systems, do not allow compilation, or are not sufficient computing platforms.
- Trying to mirror just the HTML DOM part has issues as well; a number of 3D declarative implementations (like QML 3D), all operate on a scenegraph under the hood. It seems more fruitful to just target the scenegraph for modification, and perhaps (as part of the server library) have a declarative component there.
- A shared document is desired here, as opposed to the standard browser case where every client has their own copy of state.
- Code listings are provided as an example and for exposition only. Clients and servers may be written in any language as long as they conform to the proper wireline protocol.

3. Architecture

The system envisions the use of four components, two of which fall under this specification.

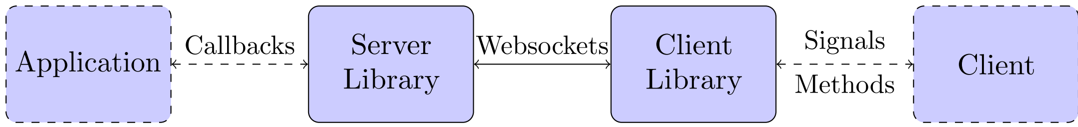

FIGURE 1. System architecture. Note that there may be more than one client. Elements with solid lines fall under this specification.

The Server Library presents a visualization to one or more connected clients through a synchronized scenegraph. Client requests and messages are passed on for handling to the application code, which can manipulate the scenegraph in response. These changes are then published and sent to clients.

The Client Library connects to a server, and maintains the synchronized scenegraph. This scenegraph is query-able by the client. Clients then can interpret and present the scenegraph to the user in the way they see fit. For example, an immersive graphics engine client can draw the scenegraph as is, while a 2D client can choose to present only a subset of the graph. A command line (i.e. Python) client may ignore the scenegraph completely to merely make use of the messaging and method invocation functionality. This also allows each client to customize the interactions available in a way that best aligns with their form factor.

3.1. **Communication.** Communication between the libraries is achieved over Websocket connections. All messages are sent over the binary channel of the WebSocket using Flatbuffers.

Client-to-client notification is not supported, and must first pass through the server.

The bulk of communication is from server to client.

This spec is intended to be implemented in a secure network, with the presumption that those that connect to the server are trusted. Provision for security will come later, as is the case with everything, because security is hard and makes my brain bleed.

3.1.1. *Flatbuffers.* For performance reasons, the *in-situ* capabilities of the serialization medium down-selected available options to Flatbuffers and Cap'n'proto. Both were explored. Table 1 compares the two in rough terms. In the end Flatbuffers won out due to more language support out of the box.

4. Concepts

The objective of the system is to synchronize, as best as possible, the document between the client and the server. This is accomplished through the use of discrete messages.

4.1. **Document.** The Document represents the visualization. It is an entity-component model, with an Object as the core entity, and Tables being a secondary entity.

The document is implicit. The other elements are explicit.

	Pro	Con
Cap'n'Proto	Created by Protobuf developers, strong pedigree. Excellent JSON inter-operation.	More complex internal formats. Fewer languages supported out of the box. Some packages for other languages are of lower quality. Default serialization code has performance issues.[1]
Flatbuffers	More languages supported out of the box. Simple internal format, more format features (such as maps, field deprecation, evolution). Easier to obtain performance increases using existing serialization code.	API for some languages is horrible. Some languages require schema to have specific design, adding indirection.

TABLE 1. Serialization Format Comparison

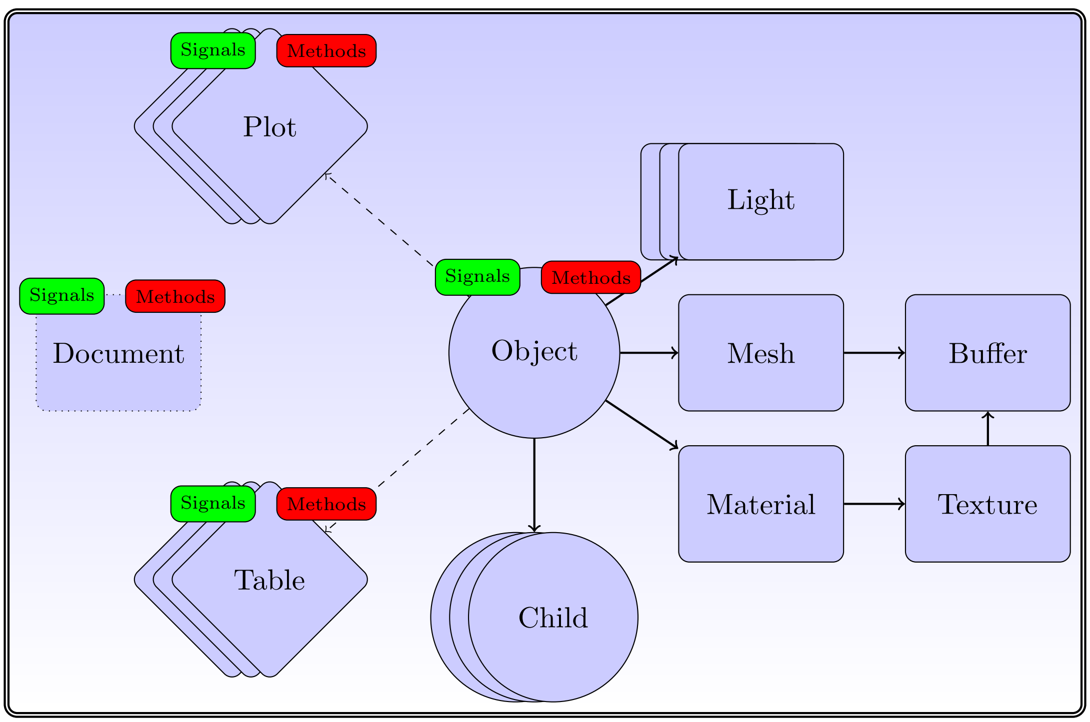

FIGURE 2. Document structure.

4.2. **Identifiers.** Identifiers are a pair of 32-bit unsigned integers; the first being a slot number, and the second being a generation count. This allows non-hashed storage, as there should be no two elements with the same slot number, so it can be used as an index in an array. The generation number is used to help identify if a slot has been recycled by the server, and thus allow detection of stale identifier use.

An identifier where either the slot and generation are the maximum unsigned integer value is the 'null' ID.

4.3. **Objects.** Each object is provided with an Object ID. Objects are rendered in a hierarchy, starting from a root object with the ID 0. Objects can have any number of children.

Each object is a possibly render-able object, and has a transformation, an optional name, a parent Object ID, a mesh (what to draw), a material (how to draw it), a number of lights, and links to tables. Objects also have a set of string tags, and attached methods and signals. Objects can also be instance rendered.

Objects are mutable.

4.4. **Tables.** Tables are a structured way to transmit row oriented data. They consist of a header (list of column names), and rows. Attached signals and methods are used to allow clients to modify the data in the table or fetch records (but only when first subscribed to).

4.5. **Plots.** Plots are a way to transmit and possibly synchronize 2D plots. They consist of either a simple textual plot definition (described below), or a URL to load in a browser.

4.6. **Signals and Methods.** Signals are notifications from the server to the client. They may contain data, and may come from the document, objects, or tables.

Methods are requests to the server from the client. They may take a set of data parameters, and they may return data as well. They must have a contextual object that they are called on, otherwise they are called on the Document. During the course of a method invocation, signals from the server could be generated.

Each method invocation is tracked by a client-generated arbitrary string. These shall be unique and never re-used. For servers, every method must generate a reply message; the only exception is if the client did not provide an invocation identifier string.

There is a possibility that a method could be called on an object, that is then subsequently deleted, or replaced. In this case, a reply is still generated, and not squashed by the server. Thus a client should be able to handle replies on objects that no longer exist.

Methods and signals are immutable.

4.7. **Buffers.** A buffer is an opaque block of bytes. This allows for efficient storage and transfer of large assets. These assets can be sent either inline through the WebSocket, or can be supplied through a URL that the client can fetch the buffer from.

Buffers are immutable and referenced from meshes and textures.

4.8. **Mesh.** Meshes define the geometry that is to be rendered. They consist of references to a buffer for number of components (see Table 2).

Meshes are mutable.

Component	Type	Value Type	Count
Position	Vertex	float	3
Normals	Vertex	float	3
TexCoords	Vertex	unsigned short	2
Colors	Vertex	unsigned byte	4
Lines	Index	unsigned short	2
Triangles	Index	unsigned short	3

TABLE 2. Mesh components

4.9. **Materials.** This should be a PBR based material, featuring basic elements: base color, metallic, roughness, including an optional texture for base colors. The material only applies to the node it is attached to. Note that though the material is specified in PBR, the client may use Phong or other interpretations of the specified material in order to meet performance goals. The material may also specify that blending should be used; the blending function is src_α and $1 - src_\alpha$.

Materials are mutable.

4.10. **Textures.** Textures reference images (in Buffers) to be used by a material. Textures are mutable.

4.11. **Lights.** Lights describe illumination sources. They are mutable.

5. Common Message Elements

This section discusses common elements to both the server and client message portions of the specification.

5.1. **Any Type.** The any type is the foundation value type. it is composed as follows:

LISTING 1. Any Definition

```
union AnyIDType {
    ObjectID,
    TableID,
    SignalID,
    MethodID,
    MaterialID,
    GeometryID,
    LightID,
    TextureID,
    BufferID
}

table AnyID {
    id : AnyIDType (required);
}
```

```
table MapEntry {
    name  : string (key);
    value : Any;
}

table Text { text : string; }
table Integer { integer : int64; }
table IntegerList { integers : [int64]; }
table Real { real : double; }
table RealList { reals : [double]; }
table Data { data : [byte]; }
table AnyList { list : [Any]; }
table AnyMap { entries : [MapEntry]; }

union AnyType {
    Text,
    Integer,
    IntegerList,
    Real,
    RealList,
    Data,
    AnyList,
    AnyMap,
    AnyID
}
```

It is a discriminated union of text, integers, real, or bytes. It also contains a generic list, and string-keyed map. For efficiency, there is also a real-list and integer-list type which allow contiguous storage or access of those elements.

In method and signal API terms, real-lists and integer-lists are coerce-able. That means, for example, that a list of reals may be provided by the `RealList` type, or as a `AnyList` of `Real`.

6. Server Messages

Here we discuss the messages that are sent from the server to the client.

Almost all components have strict lifetimes defined by creation and deletion messages. Some messages are also used to update an existing component. Therefore, if a create-update message is received by the client for a component/entity of an ID that it has never seen before, that is the creation milestone. Otherwise, it is an update message. Update messages are treated with certain semantics: either an atomic or non-atomic updates. In non-atomic update, keys in the message add or replace keys in the destination. In an atomic update, the destination is completely replaced by the message.

6.1. **Root Message.** The server sends messages by the root type `ServerMessages`. This is an array of a union of creation/deletion messages.

Listing 2. Server Root Messages

```
union ServerMessageType {
```

```
    MethodCreate,
    MethodDelete,
    SignalCreate,
    SignalDelete,
    ObjectCreateUpdate,
    ObjectDelete,
    BufferCreate,
    BufferDelete,
    MaterialCreateUpdate,
    MaterialDelete,
    TextureCreateUpdate,
    TextureDelete,
    LightCreateUpdate,
    LightDelete,
    GeometryCreate,
    GeometryDelete,
    TableCreateUpdate,
    TableDelete,
    DocumentUpdate,
    DocumentReset,
    SignalInvoke,
    MethodReply
}

table ServerMessage {
    message : ServerMessageType;
}

table ServerMessages {
    messages : [ ServerMessage ];
}

root_type ServerMessages;
```

6.2. **Objects.** Objects are created, updated and destroyed in Listing 3. All coordinates are provided in the OpenGL right hand manner.

Listing 3. Object Messages

```
table EmptyDefinition {
    padding : bool = false; // cannot be empty
}

// atomic update semantics
table TextDefinition { // Text plane, normal +z, up is +y, center: obj
   origin
    text   : string (required); // String to render
    font   : string (required); // Approximate font to use (e.g. Arial)
    height : float; // The height of the text plane.
    width  : float = -1; // Optional width of text, infer from height if <
       0
```

```
}

// atomic update semantics
table WebpageDefinition {
    url    : string (required);
    height : float = .5;
    width  : float = .5;
}

// atomic update semantics
table RenderableDefinition {
    material    : MaterialID (required);
    mesh        : GeometryID (required);
    instances   : [Mat4]; // optional
    instance_bb : BoundingBox; // optional override for instanced object
        culling
}

union ObjectDefinition {
    EmptyDefinition,
    TextDefinition,
    WebpageDefinition,
    RenderableDefinition
}

struct ObjectVisibility {
    visible : bool;
}

// nonatomic update semantics
table ObjectCreateUpdate {
    id           : ObjectID (required);
    name         : string;
    parent       : ObjectID;
    transform    : Mat4;
    definition   : ObjectDefinition;
    lights       : [LightID];
    tables       : [TableID];
    plots        : [PlotID];
    tags         : [string];
    methods_list : [MethodID];
    signals_list : [SignalID]; // Dont use "signals" to avoid Qt conflict.
    influence    : BoundingBox;
    visibility   : ObjectVisibility;
}

table ObjectDelete {
    id        : ObjectID (required);
}
```

Each element is optional, with the exception of the object id.

If the object ID has not been seen before by the client, it is assumed to be a new object. If there is no transform in the message, it is assumed to be the identity.

If the object ID has been seen before and not deleted, it should update the existing object with the elements that are provided in the message. For example, a message for Object ID 5 that contains a transform will only update object 5's transform, and not change other elements. In another example, to detach a material from an object, an update message with a null material ID is used.

Instances of the underlying meshes are specified by a list of matrices, with a matrix per instance. Using column major ordering, Matrix 1, shows how position p, rotation r (as a quaternion), color c, and scale s are specified. Access is assumed to be by column, i.e. $M_0 = p$. Transforms should be applied in the following order: scaling, rotation, translation.

$$\begin{pmatrix} p_x & c_r & r_x & s_x \\ p_y & c_g & r_y & s_y \\ p_z & c_b & r_z & s_z \\ 0 & 0 & r_w & 0 \end{pmatrix} \tag{1}$$

Objects have a definition as to how they should be represented. By default, objects use the `EmptyDefinition`. Objects can also use a `TextDefinition`, which describes that the object should be rendered as a text annotation. Text is to be rendered how the client sees fit, with the orientation to be centered at the object, the text perpendicular to $+Z$ and up being $+Y$. The height of the text must be specified; the text will then automatically use the font information to compute the text width. If the optional width is specified, then the text shall, keeping the proper font aspect ratio, try to fill the bounds provided. For more general 2D content, `WebpageDefinition` can be used to declare that the object is a web page. `RenderableDefinition` is used to declare that the object is supposed to render 3D geometry, with optional instances.

6.3. **Tables.** Tables are created and destroyed in the following messages.

Listing 4. Table Messages

```
// non-atomic update semantics
table TableCreateUpdate {
    id           : TableID (required);
    name         : string;
    meta         : string;
    methods_list : [ MethodID ];
    signals_list : [ SignalID ];
}

table TableDelete {
    id      : TableID (required);
}
```

Tables have names (which shall be unique), a metadata JSON string, methods, and signals.

6.4. **Buffers.** Buffers are created and destroyed in the following messages.

LISTING 5. Buffer Messages

```
table BufferCreate {
    id        : BufferID (required);
    bytes     : [byte]; // This could be empty
    url       : string; // This could be empty too
    size      : uint64; // Size of buffer

    // If both are empty, data is coming, just hang on for another message of
    // this kind with the bytes. you may have to request a refresh message
}

table BufferDelete {
    id     : BufferID (required);
}
```

Buffers are either inline (in the **bytes** field) or provided as a URL. If a URL is supplied the size of the buffer must be passed as well. If neither is supplied, the server has the data inline, but has deemed it too large to send immediately to avoid stalling clients. In this case, the server would do well to supply the data through another port and use the URL feature, but some servers are unable to do this. In this case, where both the bytes and URL feature are empty, the `url_size` field must still be filled for client pre-allocation. At intervals, the client can send a refresh message to fill in the missing buffers to avoid burdening the websocket (see Section 7.4).

6.5. **Geometry.** Geometries are created, and destroyed in the following messages.

LISTING 6. Geometry Messages

```
table ComponentRef { // All elements required, but not expressable in this spec
    id      : BufferID (required);
    start   : uint64;
    size    : uint64;
    stride  : uint64;
}

table GeometryCreate {
    id          : GeometryID (required);

    extent : BoundingBox;

    positions  : ComponentRef; // Vec3
    normals    : ComponentRef; // Vec3
    tex_coords : ComponentRef; // Vec2
    colors     : ComponentRef; // U8Vec4
```

```
    lines      : ComponentRef; // U16Vec2
    triangles  : ComponentRef; // U16Vec3
}

table GeometryDelete {
    id : GeometryID (required);
}
```

Limitations in Flatbuffer IDL require some notes here. Geometries are defined by ranges of a buffer for their components. These ranges, in the `ComponentRef` type, require a buffer, and also *require* a start byte offset of the buffer, as well as a byte size field, and the byte stride between vertex elements.

The `min_extent` and `max_extent` fields are required so that clients can efficiently determine culling boundaries.

The position field is required. The other vertex components (`normals, texCoords, colors`) are optional, but recommended. If there is no normal, the mesh should be rendered without lighting to avoid graphical artifacts. If there are no texture coordinates, the coordinate $(0, 0)$ should be assumed for each vertex. If there are no colors, the color $(1, 1, 1, 1)$ should be assumed for each vertex.

Index elements are specified in `lines` and `triangles`. Only one of these should be active. These specify the indices for line segments and triangles respectively. The stride for these components must be zero, i.e. they must be tightly packed.

6.6. **Texture.** Textures are created, updated, and destroyed in the following messages.

LISTING 7. Texture Messages

```
// non-atomic update semantics
table TextureCreateUpdate {
    id        : TextureID (required);
    reference : BufferRef (required);
}

table TextureDelete {
    id        : TextureID (required);
}
```

Textures, specify a buffer range for an image. For portability, these are to be in the on-disk format for PNG, JPG, or EXR. KTX is also allowed, but the user should be aware that not all clients can support it. Textures shall be interpreted as in the OpenGL coordinate system

6.7. **Material.** Materials are created, updated, and destroyed in the following messages.

LISTING 8. Material Messages

```
// non-atomic update semantics
table MaterialCreateUpdate {
    id           : MaterialID (required);
    color        : Vec4;
```

```
    metallic      : float;
    roughness     : float;
    use_blending  : bool;
    texture_id    : TextureID;
}

table MaterialDelete {
    id            : MaterialID (required);
}
```

The `color` key defines colors in the range of $0 - 1$ for r, g, b, a. Other PBR parameters are also in the $0 - 1$ range. The texture ID field is optional, and could also be null to indicate no texture.

Materials can be updated and are not immutable.

6.8. **Lights.** Lights are created, updated, and destroyed in the following messages.

Listing 9. Light Messages

```
enum LightType : byte {
    POINT = 0,
    SUN,
}

// non-atomic update semantics
table LightCreateUpdate {
    id          : LightID (required);
    color       : Vec3;
    intensity   : float;
    light_type  : LightType; // after being set once, updates ignored
}

table LightDelete {
    id          : LightID (required);
}
```

The `color` key defines colors in the range of $0 - 1$ for r, g, b. Intensity of the light is unbounded. Note that, for the moment, changing the light type after the initial creation message is not allowed.

6.9. **Signals & Methods.** Signals and Methods are created, and destroyed in the following messages.

Listing 10. Signals Messages

```
table MethodArg {
    name : string;
    doc  : string;
    hint : string;
}
table MethodCreate {
    id              : MethodID (required);
```

```
    name          : string;
    documentation : string;
    return_doc    : string;
    arg_doc       : [ MethodArg ];
}

table MethodDelete {
    id            : MethodID (required);
}
```

Methods must be provided with a human friendly name. Two methods may not share the same name; there is no overloading. Documentation is recommended, but not required, as is return value documentation. Argument information must be provided, at the very least given a name. You may not call a method with more arguments then as specified; use an argument that takes an array type to permit this option.

Signal must be provided with a human friendly name, and also may not share the same name. Arguments follow the same requirements as methods.

6.10. **Signal Invoke & Method Reply.** Signals may be invoked on the client and client methods replied with the following messages.

Listing 11. Communication

```
table SignalInvoke {
    id   : SignalID (required);

    // if the two below are not set, it is on the document
    on_object : ObjectID;
    on_table  : TableID;
    on_plot   : PlotID;

    signal_data  : AnyList;
}

table MethodException {
    code    : int64;   // required
    message : string; // optional
    data    : Any;    // optional
}

table MethodReply {
    invoke_ident     : string (required);
    method_data      : Any;              // optional
    method_exception : MethodException; // optional
}
```

Either only `on_object` or `on_table` or `on_plot` or none must be set, to indicate context. Signals may NOT be invoked on a context that does not have them attached.

Method replies must have a previously given method invocation identifier (see Section 7.3). If the method could not be executed, an exception field is filled instead of data.

Code	Message	Description
-32700	Parse Error	Given invocation object is malformed and failed to be validated
-32600	Invalid Request	Given invocation object does not fulfill required semantics
-32601	Method Not Found	Given invocation object tries to call a method that does not exist
-32602	Invalid Parameters	Given invocation tries to call a method with invalid parameters
-32603	Internal Error	The invocation fulfills all requirements, but an internal error prevents the server from executing it

Table 3. Error Codes

In an exception, the code should represent either one of the predefined error codes in Table 3, or a code in the defined user-code region. A short message should be provided for users; additional data may also be provided for things like nested errors. Given the differences in clients, however, it is likely that such data would be flattened to a string.

Reserved error codes are provided in Table 3 and are designed to match the XMLRPC and JSONRPC codes. Error codes -32768 to -32000 are reserved by the spec.

6.11. **Document.** The document is implicit, and always exists. It can be modified with the following messages.

Listing 12. Document Messages

```
table DocumentUpdate {
    methods_list  : [ MethodID ];
    signals_list  : [ SignalID ];
}

table DocumentReset {
    padding : bool; // these things cannot be empty, so...
}
```

The document may be updated with `DocumentUpdate`, to modify the current methods and signals. It may also be completely reset. The reset message, by quirk of Flatbuffers, may not be empty; ignore any fields within. When a document is reset, all components and objects are to be deleted at that point.

7. Client Messages

In this section we discuss the messages sent by a client.

7.1. **Root Message.** Client messages are defined as the following root type.

Listing 13. Client Message Root

```
union ClientMessageType {
    IntroductionMessage,
    MethodInvokeMessage,
    AssetRefreshMessage
}

table ClientMessage {
    content : ClientMessageType (required);
}

table ClientMessages {
    messages : [ ClientMessage ] (required);
}

root_type ClientMessages;
```

7.2. **Introduction.** The client introduces itself to the server with the following message.

Listing 14. Introduction Message

```
table IntroductionMessage {
    client_name : string (required);
    version : uint32 = 0;
}
```

The name of the client must not be empty, and should identify a client; host names can be used. The version integer represents the major version of the specification that the client supports. Version 0 implies that the client is a pre-version-1 client.

7.3. **Method Invocation.** The client asks to invoke a method with the following message.

Listing 15. Method Invocation

```
table MethodInvokeMessage {
    method_id : MethodID (required);

    // if any of the below is not set, it is on the document
    on_object : ObjectID;
    on_table  : TableID;
    on_plot   : PlotID;

    invoke_ident : string (required);
    method_args  : AnyList;
}
```

The message must have an invocation identifier; the asynchronous reply will carry that identifier. Identifiers must not be reused.

Either the `on_object` or the `on_table` or `on_plot` or none should be set, to indicate the context of the invocation: on an object, on a table, or on a plot or on the document, respectively. The method can only be called on a context on which it is attached.

The arguments to the method must match the documented method signature.

7.4. **Asset Refresh.** The client may ask to receive missing buffer content with the following message.

Listing 16. Buffer Refresh

```
table AssetRefreshMessage {
    for_buffers : [ BufferID ] (required);
}
```

8. Semantics

8.1. **Tables.** Tables are a way of exposing record data to clients so that they can either provide an alternative representation of that data or to allow command line clients access to the data. An example of an alternative representation would be a 2D chart that could be provided for a lightweight 2D client instead of a 3D plot. Another approach would be to allow a visual representation to provide a link to details of a certain data point.

Tables consist of columns (with unique names) and rows. Rows are identified by a `Key`, which is an integer. Keys are assumed to be monotonically increasing, starting from 0, that is, new insertions into the database are given a new key larger than any key seen before.

Another useful abstraction is the `Row` type; a row is either an `AnyList` or a `RealList`. A `Column` is the same.

A commonly used notion is the concept of a selection within a table of data. Listing 17 shows, in a JSON-like way, the definition of a Selection object as encoded in a NOODLES Any.

Listing 17. Selection object definition. Note that the `to` field in the row ranges is exclusive. The `row_ranges` list *must* have an even number of elements.

```
{
    "rows" : [Key, ...], // must be an IntegerList
    "row_ranges" : [
        Key from, Key to,
        ...
    ] // also must be represented as an IntegerList
}
```

8.1.1. *Methods & Signals.* To query table information, signals and methods are used. These names are restricted and cannot be used by the user application. Note, indexes are all zero-based. Tables 4 and 5 list the data related methods and signals a table can support. The server should not send any data or signals to the client for a given table *unless* a client has expressed interest by calling the subscribe method. This is to avoid stressing clients that have no table interface and to reduce unnecessary network traffic. Further it is up to the server to honor these methods; should the server not support modification, for example, requests will return an exception.

Method Name	Description
`TblInit tbl_subscribe()`	Subscribe to changes in the table, receiving initial table state. The client will then receive signals.
`void tbl_insert([Column])`	Request to add rows of data to the table, as a pack of column segments.
`void tbl_update([Key], [Column])`	Request to update many rows of data to the table, as a pack of column segments.
`void tbl_remove([Key])`	Ask to remove a list of keys.
`void tbl_clear()`	Ask to remove all rows of the table.
`void tbl_update_selection(/*snip*/)`	Ask to update a selection in the table.

TABLE 4. Table Methods summary

Signal Name	Description
`void tbl_reset()`	Reinitialize the table. Sent if the table is cleared or reset in some way.
`void tbl_updated([Key], [Column])`	Rows were updated in the table.
`void tbl_rows_removed([Key])`	Rows in the table were removed.
`void tbl_selection_updated(/*snip*/)`	A selection has changed.

TABLE 5. Table Signals summary

Subscribe. This allows the client to receive signals from the table. Without this, no signal should be sent by the server regarding the table. When this call is made, the server will reply with a `TblInit` object. The full object definition is as follows:

```
TblInit : {
    "columns" : [ string ], // 1
    "keys" : [ Key ], // 2
    "data" : [ Column ], // 2
    "selections" : [ [string, SelectionObject] ] // 3
}
```

Part 1 is a list of columns. This establishes a column order that is used to interpret and pack data values later in other calls and signals. The second elements provide the current data that is in the table, as well as the keys used to identify rows. The third is a pack of the current selections that are available in the table; this is provided as a list of pairs, where the first part of the pair is the string identifier of the selection and the second is the selection object that defines the selection.

Reset. Should the server issue the `tbl_reset` signal, this would imply that the table has been reset, with no data, and no selections, but with the same header.

Insertion. Data may be inserted into the table through both the row and many versions of the call. Note the key cannot be specified. The row length should be equal to the length of the header, and supplied in header order. The many version simply takes a list of rows to be inserted. Insertion success is demonstrated through reception of the `rows_inserted` signal; this signal provides the data inserted along with the keys that were assigned to that row, i.e. the full row of data for all columns.
Update. Data can be updated through both the row and many versions. In this case, as opposed to the insertion functions, the full row, including the key column, is specified in column order, so that the correct row may be updated. Success is indicated through the corresponding update signal.
Removal. Data can be removed by specifying a list of keys to delete. Success will be indicated through the corresponding signal for all clients.
Selection. Data selections can be made through the `update_selection` call. The full signature of the call is as follows:

```
void tbl_update_selection( string, SelectionObject );
```

The first argument denotes the selection to update or add, and the selection object defines what that selection should be updated/initialized to. A selection object that is empty, i.e. specifying no rows or ranges is considered the empty selection and denotes that the selection should be deleted from clients.

This shall trigger the selection update signal. The full signature of the signal is as follows:

```
void tbl_selection_updated( string, SelectionObject );
```

This mirrors the update call, and denotes which selection has changed, and what to change it to.

8.1.2. *Tables Metadata.* Tables are also capable of synchronizing metadata for other purposes. This is exposed as a JSON object.

8.2. **Plots.** To facilitate 2D plot synchronization, multiple optional mechanisms are present. Plots expose a simple definition system, and a URL system.

8.2.1. *Simple.* In the `SimplePlot` object, there is a single member `definition`. This is a JSON encoded object, containing one of several formats.

The first format provides a simple encoded approach:

Listing 18. Table Metadata for Plot Sync

```
definition : {
    "simple_plot" : SimplePlotInfo ,
    <other keys>
}

SimplePlotInfo : {
    "plot_name" : "name",
    "columns" : [ ColumnInfo ]
```

```
}

ColumnInfo : {
    "column_name" : "<name>",
    "prefers" : "x" | "y",
    "color" : "#rrggbb",
    "range" : [from, to]
}
```

In Listing 18, the definition JSON will contain a key called `simple_plot`. This key is a listing of named plots; each plot describes how each column of the table should be treated in an arbitrary simple plot.

Complex 2D Plot Sync: More advanced plotting facilities are forthcoming, but planned to follow a system like: `http://docs.juliaplots.org/latest/attributes/`.

Web. Another option is to directly expose a URL for web access. This allows for complex server-based or other peer to peer 2D synchronization tools.

8.3. **Objects.**

Objects may carry the logical operations.

For simplicity, in this section, we let `vec3 = RealList` and `vec4 = RealList`. When used as arguments, the three component and four component vectors require the exact number of components to be supplied in the list. Otherwise the server will consider that to be malformed, and can reject the call.

8.3.1. *Activator.* For clients, this could be when the user clicks on an object, or presses an interaction button when a wand is over an object.

```
void activate(string)
void activate(int)
list<string> get_activation_choices()
```

It is up to the server application to decide how to handle this ‘activation’. Activation is either in the string or integer form. Activation names can be obtained through the API (example: ‘Click’, ‘Clear Options’). An activation can be triggered by the string, or by an integer index. It makes sense to thus tie the order of names to priorities; a 0 is a primary click, 1 is an alternate click action, etc.

8.3.2. *Options.* Options are is conceptually the same thing as a combo-box widget.

```
list<string> get_option_choices()
string get_current_option()
void set_current_option(string)
```

A list of choices can be presented for an object, and an option can be set.

8.3.3. *Movable.* Movable objects allows the user to request to change the position of an object.

```
void set_position(vec3 p)
void set_rotation(vec4 q)
void set_scale(vec3 s)
```

Positions, rotations and scales are in the coordinate system of the parent object. The rotation is to be provided as a quaternion, with w being the last component.

8.3.4. *Selection.* Regions of an object can be 'selected'. What this means is up to the application.

```
void select_region(vec3, vec3, int)
void select_sphere(vec3, real, int)
void select_half_plane(vec3, vec3, int)
void select_hull([vec3], [int], int)
```

The selection API allows for a number of different selection tools. Others can be forged through the use of the movable API, and activators. All coordinates provided are in the object-local coordinate space.

For `select_region`, the selection region is supplied as an axis aligned bounding box, and an option for either additive selection (> 0), deselection (< 0) or replacement ($= 0$). For `select_sphere`, a position and a radius is supplied. For `select_half_plane`, a point and a normal is provided. For `select_hull`, the client provides a list of 3D points, and an index list interpreted as a mesh hull.

To support multiple selections, consider adding options and activators to your object.

8.3.5. *Query.* Objects can be probed to obtain a data value or annotation.

```
[string, vec3] probe_at(vec3)
```

The location (object local coordinates) to be probed is supplied in the argument. As a return value, a revised position is returned (in case the server desires to snap the probe to a different location) and a string containing the data to display.

Note that more complex actions may take place; a user can build their application to add more functionality (or use a different activator), which can instantiate objects for all users to see.

8.3.6. *Annotation and Attention.* The object may request user attention, through the following signals.

```
void signal_attention()
void signal_attention(vec3)
void signal_attention(vec3, string)
```

Multiple overloads are provided. If the signal has no data, the whole object would like attention. If there is a position, a specific object-local coordinate would like attention. If there is a string in addition to that, a message should be displayed at that point.

To attract attention, sounds, client-specific graphical adornment can all be used. For some clients, changing the camera view to include the point of attention can also be done.

8.3.7. *Object Tags.* Objects may be given tags. They are a list of strings. These allow the client to discover capabilities of the Object, or classify an object. Some tags imply the presence of certain methods or signals. Tags prefixed with `noo_` are reserved for use by the system.

Tag	**Description**
`noo_user_hidden`	On lists of objects or tree-views, this object should be hidden. Other objects should be visible[2]

8.4. **Scene Semantics.**

8.4.1. *Reporting.* Clients may inform the server of areas of 'interest' of the given scene through reporting methods attached to the document.

```
void noo_client_view(vec3 direction, real angle)
```

Note that 'interest' is different for different clients. As an example, a desktop client may wish to signal interest via a mouse. An AR system may consider an eye-tracking based approach. For an Immersive VR environment, head direction might be used.

This method, if it exists, should not be called very often; as we are sampling the user, view information can be provided at a human scale, on the order of a second or more.

9. Operation & Lifecycle

9.1. **Websocket Messages.** The server side shall send the `ServerMessages` message, while clients are restricted to sending `ClientMessages` message.

9.2. **Connection.** Upon the connection of a websocket, the client first sends an introduction message. Any other message is ignored by the server until the introduction is provided.

The server will then send a list of creation messages to build the scene. This could pose a problem; large mesh or texture assets could take a significant amount of time to transfer and attempting to send those all at the start could cause issues ranging from the server being blocked, the client being overwhelmed, or de-synchronization, depending on implementations. In order to avoid this, the server may send creation info of buffers without the data. The client can use placeholder assets, and use the asset refresh mechanism to request asset updates with full information, which it can then use to update the graphical representation.

From this point onward, the client can invoke methods, and the server can send signals and other messages.

[2] This approach (hidden-specified) is chosen, because in a visible-specified, it is difficult to know when to hide the other objects.

Appendix A. Common Flatbuffer Specification

```
// Generate with
// flatc --scoped-enums --reflect-names --gen-mutable -c noodles.fbs

namespace noodles;

// Identifiers == These are tables, due to poor language support for
    structs.

table ObjectID {
    id_slot : uint32;
    id_gen  : uint32;
}

table PlotID {
    id_slot : uint32;
    id_gen  : uint32;
}

table TableID {
    id_slot : uint32;
    id_gen  : uint32;
}

table SignalID {
    id_slot : uint32;
    id_gen  : uint32;
}

table MethodID {
    id_slot : uint32;
    id_gen  : uint32;
}

table MaterialID {
    id_slot : uint32;
    id_gen  : uint32;
}

table GeometryID {
    id_slot : uint32;
    id_gen  : uint32;
}

table LightID {
    id_slot : uint32;
    id_gen  : uint32;
}

```

```
table TextureID {
    id_slot : uint32;
    id_gen  : uint32;
}

table BufferID {
    id_slot : uint32;
    id_gen  : uint32;
}

union AnyIDType {
    ObjectID,
    TableID,
    SignalID,
    MethodID,
    MaterialID,
    GeometryID,
    LightID,
    TextureID,
    BufferID
}

table AnyID {
    id : AnyIDType (required);
}

table MapEntry {
    name  : string (key);
    value : Any;
}

table Text { text : string; }
table Integer { integer : int64; }
table IntegerList { integers : [int64]; }
table Real { real : double; }
table RealList { reals : [double]; }
table Data { data : [byte]; }
table AnyList { list : [Any]; }
table AnyMap { entries : [MapEntry]; }

union AnyType {
    Text,
    Integer,
    IntegerList,
    Real,
    RealList,
    Data,
    AnyList,
    AnyMap,
    AnyID
```

```
}

table Any {
    any : AnyType;
}

// Misc Types
    ================================================================

struct Vec2 {
    x : float;
    y : float;
}

struct Vec3 {
    x : float;
    y : float;
    z : float;
}

struct Vec4 {
    x : float;
    y : float;
    z : float;
    w : float;
}

struct Mat4 {
    // for javascript compat, we have to expand this.
    // components : [float : 16];
    c1 : Vec4;
    c2 : Vec4;
    c3 : Vec4;
    c4 : Vec4;
}

struct BoundingBox {
    aabb_min : Vec3;
    aabb_max : Vec3;
}

table BufferRef {
    id    : BufferID;
    start : uint64;
    size  : uint64;
}
```

Appendix B. Server Message Flatbuffer Specification

```
// Generate with
// flatc --scoped-enums --reflect-names --gen-mutable -c noodles_server.fbs

include "noodles.fbs";

namespace noodles;

// Server Messages ==============================================================

table MethodArg {
    name : string;
    doc  : string;
    hint : string;
}
table MethodCreate {
    id            : MethodID (required);
    name          : string;
    documentation : string;
    return_doc    : string;
    arg_doc       : [ MethodArg ];
}

table MethodDelete {
    id            : MethodID (required);
}

// ==============================================================================

table SignalCreate {
    id             : SignalID (required);
    name           : string;
    documentation  : string;
    arg_doc        : [ MethodArg ];
}

table SignalDelete {
    id             : SignalID (required);
}

// ==============================================================================

table EmptyDefinition {
    padding : bool = false; // cannot be empty
}

```

```
// atomic update semantics
table TextDefinition { // Text plane, normal +z, up is +y, center: obj origin
    text   : string (required); // String to render
    font   : string (required); // Approximate font to use (e.g. Arial)
    height : float; // The height of the text plane.
    width  : float = -1; // Optional width of text, infer from height if < 0
}

// atomic update semantics
table WebpageDefinition {
    url    : string (required);
    height : float = .5;
    width  : float = .5;
}

// atomic update semantics
table RenderableDefinition {
    material    : MaterialID (required);
    mesh        : GeometryID (required);
    instances   : [Mat4]; // optional
    instance_bb : BoundingBox; // optional override for instanced object culling
}

union ObjectDefinition {
    EmptyDefinition,
    TextDefinition,
    WebpageDefinition,
    RenderableDefinition
}

struct ObjectVisibility {
    visible : bool;
}

// nonatomic update semantics
table ObjectCreateUpdate {
    id           : ObjectID (required);
    name         : string;
    parent       : ObjectID;
    transform    : Mat4;
    definition   : ObjectDefinition;
    lights       : [LightID];
    tables       : [TableID];
    plots        : [PlotID];
    tags         : [string];
    methods_list : [MethodID];
    signals_list : [SignalID]; // Dont use "signals" to avoid Qt conflict.
```

```
    influence    : BoundingBox;
    visibility   : ObjectVisibility;
}

table ObjectDelete {
    id        : ObjectID (required);
}

//
    ==========================================================================

table SimplePlot {
    definition : string (required);
}

table URLPlot {
    url : string (required);
}

union PlotType {
    SimplePlot,
    URLPlot
}

// non-atomic update semantics
table PlotCreateUpdate {
    id           : ObjectID (required);
    table        : TableID;
    type         : PlotType;
    methods_list : [MethodID];
    signals_list : [SignalID];
}

table PlotDelete {
    id        : ObjectID (required);
}

//
    ==========================================================================

table BufferCreate {
    id       : BufferID (required);
    bytes    : [byte]; // This could be empty
    url      : string; // This could be empty too
    size     : uint64; // Size of buffer

    // If both are empty, data is coming, just hang on for another message
        of
```

```
    // this kind with the bytes. you may have to request a refresh message
}

table BufferDelete {
    id    : BufferID (required);
}

// =============================================================================

// non-atomic update semantics
table MaterialCreateUpdate {
    id            : MaterialID (required);
    color         : Vec4;
    metallic      : float;
    roughness     : float;
    use_blending  : bool;
    texture_id    : TextureID;
}

table MaterialDelete {
    id            : MaterialID (required);
}

// =============================================================================

// non-atomic update semantics
table TextureCreateUpdate {
    id        : TextureID (required);
    reference : BufferRef (required);
}

table TextureDelete {
    id        : TextureID (required);
}

// =============================================================================

enum LightType : byte {
    POINT = 0,
    SUN,
}

// non-atomic update semantics
table LightCreateUpdate {
```

```
    id         : LightID (required);
    color      : Vec3;
    intensity  : float;
    light_type : LightType; // after being set once, updates ignored
}

table LightDelete {
    id        : LightID (required);
}

// =============================================================================

table ComponentRef { // All elements required, but not expressable in this spec
    id     : BufferID (required);
    start  : uint64;
    size   : uint64;
    stride : uint64;
}

table GeometryCreate {
    id         : GeometryID (required);

    extent : BoundingBox;

    positions  : ComponentRef; // Vec3
    normals    : ComponentRef; // Vec3
    tex_coords : ComponentRef; // Vec2
    colors     : ComponentRef; // U8Vec4
    lines      : ComponentRef; // U16Vec2
    triangles  : ComponentRef; // U16Vec3
}

table GeometryDelete {
    id : GeometryID (required);
}

// =============================================================================

// non-atomic update semantics
table TableCreateUpdate {
    id           : TableID (required);
    name         : string;
    meta         : string;
    methods_list : [ MethodID ];
    signals_list : [ SignalID ];
```

```
}

table TableDelete {
    id      : TableID (required);
}

// ==============================================================================

table DocumentUpdate {
    methods_list  : [ MethodID ];
    signals_list  : [ SignalID ];
}

table DocumentReset {
    padding : bool; // these things cannot be empty, so...
}

// ==============================================================================

table SignalInvoke {
    id  : SignalID (required);

    // if the two below are not set, it is on the document
    on_object : ObjectID;
    on_table  : TableID;
    on_plot   : PlotID;

    signal_data  : AnyList;
}

table MethodException {
    code    : int64;  // required
    message : string; // optional
    data    : Any;    // optional
}

table MethodReply {
    invoke_ident     : string (required);
    method_data      : Any;              // optional
    method_exception : MethodException; // optional
}

// ==============================================================================

```

```
union ServerMessageType {
    MethodCreate,
    MethodDelete,
    SignalCreate,
    SignalDelete,
    ObjectCreateUpdate,
    ObjectDelete,
    BufferCreate,
    BufferDelete,
    MaterialCreateUpdate,
    MaterialDelete,
    TextureCreateUpdate,
    TextureDelete,
    LightCreateUpdate,
    LightDelete,
    GeometryCreate,
    GeometryDelete,
    TableCreateUpdate,
    TableDelete,
    DocumentUpdate,
    DocumentReset,
    SignalInvoke,
    MethodReply
}

table ServerMessage {
    message : ServerMessageType;
}

table ServerMessages {
    messages : [ ServerMessage ];
}

root_type ServerMessages;
```

Appendix C. Client Message Flatbuffer Specification

```
// Generate with
// flatc --scoped-enums --reflect-names --gen-mutable -c noodles_client.fbs

include "noodles.fbs";

namespace noodles;

// Client Messages
    ============================================================

table IntroductionMessage {
    client_name : string (required);
```

```
    version : uint32 = 0;
}

table MethodInvokeMessage {
    method_id : MethodID (required);

    // if any of the below is not set, it is on the document
    on_object : ObjectID;
    on_table  : TableID;
    on_plot   : PlotID;

    invoke_ident : string (required);
    method_args  : AnyList;
}

table AssetRefreshMessage {
    for_buffers : [ BufferID ] (required);
}

union ClientMessageType {
    IntroductionMessage,
    MethodInvokeMessage,
    AssetRefreshMessage
}

table ClientMessage {
    content : ClientMessageType (required);
}

table ClientMessages {
    messages : [ ClientMessage ] (required);
}

root_type ClientMessages;
```