TP002: The RealiMation Scene Graph

Introduction

At the core of the RealiMation system is its scene graph, which is a collection of different object types and their interconnections. This document describes the scene graph, pointing out how to use the API to manipulate it, and the features provided by the various object interconnections.

A scene graph is built up using RealiMation API (Application Programming Interface) calls, and once data has been sent through the API, the scene graph is known as a *RealiBase*. The RealiMation API provides functions for saving and loading a RealiBase to and from hard disk.

RealiMation Objects

There are eleven different object types in the RealiMation API:-

Type	Function
Server	Distributed network rendering
Channel	Represents display hardware
View	A container of 3D viewable objects
Geometry	Represents the shape of an object
Instance	Instantiates geometry for building articulated object hierarchies
Light	Illuminates a scene
Atmospheric	Contains lights and fogging
Camera	Represents the 3D to 2D projection
Path	Moves objects over time
Image	Textures, backgrounds, photo-realistic depth buffering
Material	The light reflectance characteristics of a shape

Each of these object types has its own object specific properties (such as colour and transparency for a material). They can be uniquely named, and have application specific properties attached to them.

At run time, each object is assigned a unique ID (identifier), which is used as a handle to access particular objects. These handles can either be queried by name or by using the various query functions supplied by the API. For example, to get the ID of an object called "View 1", simply call

```
RTGetIDFromName ("View 1");
```

The return value of this call can then be used to access the data of the object, attach it to other objects, or simply call one of the functions that processes it (e.g. `RTDisplayView()` will draw the 3D view).

The next section shows how these different object types work together to generate real-time multi channel 3D output.

Object Interconnections - The Scene Graph

The diagram on the following page shows how different object types in RealiMation interconnect, and the features that result from each link. As well as showing the features, the actual API calls used to make the connection are displayed. Notice how many different features are implemented by a small set of API calls - see the next section for more information.

The diagram also shows 1-to-1 and 1-to-many relationships between objects. A view can reference a single camera, and contain lots of instances, for example.

What the diagram does not show, however, is the fact that one instance can belong to a number of different views. In general, all RealiMation objects can be referenced from multiple other objects, which is another way of saying, for example, that one material can be shared amongst many geometry objects. The diagram shows the relationship between a *single* object of a given type, and the other object types. The arrows between object types are meant to help, as they indicate the direction of reference. For example, a view can reference a background image, but putting the arrow in means that the link cannot be interpreted as meaning that a background image can only exist in one view.

Object Handles for a Compact API

The diagram shows that many different object links are implemented by just a small subset of API calls, and thus keeps the API very compact, yet still feature rich, since functions are defined by *type linkage*, rather than by special functions. For example, `RTSetObject()` does all of the following:

- Terrain following
- Make an instance move
- Make a camera move
- Make a light move
- Define complex hierarchical embedded motions
- Define a view's lighting and fogging
- Define a view's 3D->2D projection matrix

This works because all objects, irrespective of type, have a unique ID. `RTSetObject()` takes two ID arguments - the container and the object to be contained. All of the above features can be uniquely defined just by the type of the arguments supplied. For example, a view's 3D to 2D projection is defined by containing a camera object within a view.

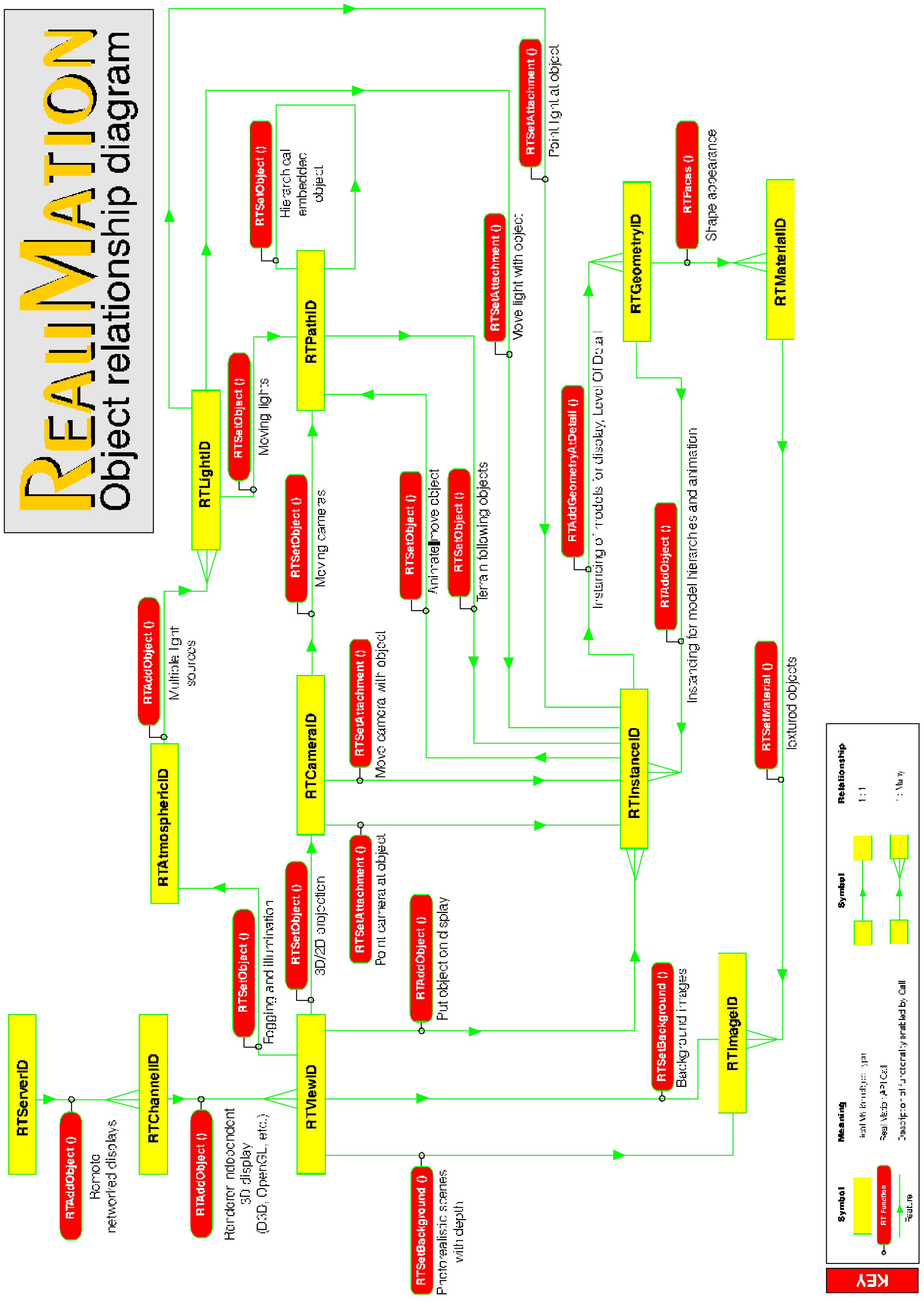
The corresponding `RTInqObject()` means that only two functions are needed instead of fourteen. As long as the developer understands the inter-object linkages, the number of functions to remember is vastly reduced, improving both API learning curve and application development time.

A further advantage is that if a new linkage is developed (for example, letting an instance have a material), the same API's can be used if they show the same semantics.

RealiMation splits up which functions are used based on the following factors:

- Does the simple fact that two ID's are linked provide all the information necessary to make that linkage meaningful?
- Are two ID's linked by a 1-to-1 or a 1-to-many relationship?

The first factor decides if one of the general linkage functions is used (such as `RTSetObject()`), or whether more information is required to put the link in context. For example, `RTAddGeometryAtDetail()` is used rather than `RTAddObject()` because, to make level of detail meaningful, the position of the geometry object in the set of geometry's is required. A table of which calls are used when is shown following the diagram:


REALIMATION
Object relationship diagram
RTServerID
RTAddObject ()
Remote networked displays
RTChannelID
RTAddObject ()
Renderer independent 3D display (D3D, OpenGL, etc.)
RTViewID
RTSetBackground ()
Photo realistic scenes with depth
RTAtmosphericID
RTSetObject ()
Fogging and illumination
RTSetObject ()
3D/2D projection
RTSetAttachment ()
Point camera at object
RTAddObject ()
Put object on display
RTSetBackground ()
Background images
RTImageID
RTCameraID
RTSetAttachment ()
Move camera with object
RTSetObject ()
Moving cameras
RTLightID
RTAddObject ()
Multiple light sources
RTSetObject ()
Moving lights
RTPathID
RTSetObject ()
Hierarchical embedded object
RTSetObject ()
Animate/move object
RTSetObject ()
Terrain following objects
RTSetAttachment ()
Move light with object
RTSetAttachment ()
Point light at object
RTAddGeometryAtDetail ()
Instancing of models for display, Level Of Detail
RTAddObject ()
Instancing for model hierarchies and animation
RTInstanceID
RTGeometryID
RTFaces ()
Shape appearance
RTMaterialID
RTSetMaterial ()
Textured objects
KEY
Symbol
Meaning
Real World object type
RT Function
Realimation API Call
Feature
Description of functionality enabled by Call
Relationship
1:1
1:Many

API Function	Usage
`RTSetObject()`	Handles generic 1-to-1 mappings (e.g. camera attached to a view)
`RTAddObject()`	Handles generic 1-to-many links (e.g. a number of instances attached to a view)
`RTSetAttachment()`	Handles 1-to-1 mappings, but where some extra meaning is required, such as a path through the scene graph for point-at/move-with operations (e.g. always point a camera at the turret of a tank)
`RTAddGeometryAtDetail()`	Handles specific case of adding multiple geometry's to an instance for the creation of level of detail models for display (see note below).
`RTFaces()`	Handles specific case of attaching faces (which reference materials) to a geometry object to define shape. Since a geometry object can have multiple faces, and each face can have its own material, this is effectively a 1-to-many relationship between geometry and material objects.
`RTSetMaterial()`	Handles specific case of attaching images to materials to provide texture effects when displayed. A material can have multiple images.
`RTSetBackground()`	Handles the specific case of attaching visual and depth backgrounds to views. Each view can have one visual background and one depth background.

Note: All geometry's attached top instances are treated as level of detail models, even if there is only a single geometry (and therefore level).

There is also a "fast wrapper" function that can be used in place of multiple calls to `RTAddObject()`, for use when adding lots of objects to a single container. This function, `RTAddObjects()`, is very useful when adding lots of instances to a geometry object, for example.

Breaking Object Links

Often applications want to remove object linkages. For example, an aircraft may be carrying a missile as a sub-instance, but when the missile is fired it needs to exist independently of the aircraft.

The following table shows the removal counterpart functions to the creation methods mentioned above:

Creation Function	Removal mechanism
`RTSetObject()`	`RTSetObject()`, supplying a null ID for referenced object
`RTAddObject()`	`RTRemoveObject()`
`RTAddObjects()`	`RTRemoveObjects()`
`RTSetAttachment()`	`RTSetAttachment()`, passing in a null object array
`RTAddGeometryAtDetail()`	`RTDeleteDetailLevel()`
`RTFaces()`	`RTSetFaces()`, `RTEmptyGeometry()`
`RTSetMaterial()`	`RTSetMaterial()`, supplying a null ID for images
`RTSetBackground()`	`RTSetBackground()`, supplying a null ID for images

Querying the Scene Graph

There are several ways of interrogating the scene graph to get handles of specific object. Once an application has the handles, it can change anything in the RealiBase. An important feature of the API is that *anything* that goes into the RealiBase can come out again. This means that anything that has been defined (whether by other applications such as the STE, or the application program itself), can be modified, deleted, or added to.

The following table lists the query mechanisms for the various linkage creation functions listed earlier:

Creation Function	Query mechanism
`RTSetObject()`	`RTInqObject()`
`RTAddObject()`	`RTGetNextObject()`
`RTSetAttachment()`	`RTInqAttachment()`
`RTAddGeometryAtDetail()`	`RTInqGeometryAtDetail()`
`RTFaces()`	`RTInqFaces()`
`RTSetMaterial()`	`RTInqMaterial()`
`RTSetBackground()`	`RTInqBackground()`

All objects exist “standalone”, in that just because they are in the RealiBase they do not have to be linked up to other objects. This means that there is always a direct access mechanism to objects, without having to traverse a whole graph and all its linkages. The function used to iterate through the ID’s of each different object type is `RTGetNextID()`.

The ability of objects to exist by themselves without having to be referenced is extremely useful. For example, objects can exist in memory that are swapped in and out dynamically under application control, without having to be recreated or located on hard disk again. Another example that is often used is to create a view object containing collision models (normally simpler versions of the actual visual ones), and use this for raytesting and collision detection of just those objects required. A view can be processed and accessed even without displaying it.

Another way of traversing the scene graph is by using the `RTWalkObject()` function, supplying your own callback function. This takes in any object type, and calls the supplied function for each scene graph object (or node) visited.

When building scene graphs, particularly ones that support object instancing, and important feature is the ability to test if adding one object to another would result in an invalid cyclic dependency. For example, having object A reference object B which references C which references A again would cause infinite loops and program crashes. The RealiMation API provides a function `RTIsSubObject()` to help applications guard such an occasion, without imposing a performance overhead of checking every time an object is added into the scene graph.

Two useful utility functions are `RTListGeometryInstances()` and `RTListInstance References()`. The former allows an application to find all instances of a particular geometry object, while the latter returns all geometry objects that contain a particular instance.

Object Properties

Each object type has its own set of intrinsic properties. For example, a camera has position, orientation, and field of view as some of its unique properties that, in effect, define it as being a “camera”. Similarly, an image has its own set of image-specific properties such as pixels, mip mapping mode etc.

In addition to these intrinsic (sometimes called “stock”) properties, applications can add their own property values to any ID. In other words, an application can append its own data structures to any object. Almost any RealiMation application can benefit from using this feature. This ability to add application defined information to an object, in conjunction with objects being able to exist independently, means that the developer does not have to duplicate the RealiMation scene graph to create their own records in complex applications. All they need to store is object handles, and let RealiMation look after the structure.

The API provides a set of common properties that are useful at run rime, such as ID locks and hotlinks. The RealiMation STE makes extensive use of properties for many of its processing operations. For example, object edit boxes (those that appear when picking) and their original objects are tied together using properties, as are the graphical representation of lights and the lights they represent.

A Trip through the Display Pipeline

Given a scene graph, what happens when an application calls `RTDisplayView()` to draw a picture on a screen?

If the view is running on a remote network server (i.e. in a RealiNet environment), the server issues an instruction for it to process the view remotely. The local machine (the one on which the application is running), then returns immediately to the application. If not running remotely, `RTDisplayView()` carries on processing as below.

The camera transformation matrix and clipping information is updated as necessary to allow for any changes in time (time is set by calling `RTSetViewTime()` before calling `RTDisplayView()`). This allows for cameras that are on paths, moving with or looking at instances. Any lights in the atmospheric attached to the view are then updated if on paths, moving with or looking at instances.

The view background is then cleared (although this can be disabled) by either filling in with a solid colour or by using a background image. If the image has been defined as a scrolling image (i.e. it moves based on camera direction), then this is taken into account here. If required, the depth buffer is also cleared. In cases where there is a depth buffer image, this image is used to set the initial z values for all pixels. If no depth image is being used, then the depth is cleared to the maximum allowable value.

Once the projection, lighting, and background has been taken care of, the instances in the view are iterated through one at a time.

An instance has its position and size evaluated according to the current time (which may be overridden), which is then used to evaluate the level of detail models attached to the instance. Any models (i.e. geometry objects) that are candidates for display after this test are checked to see if they will have a visible effect on the view (based on view projection etc.).

When the object is accepted for further processing, the faces are optionally back faces culled and illuminated (this phase can be skipped - geometry objects can be pre lit or use fixed colours). The points are transformed into camera space and the final polygons dispatched to the renderer. Note that some RealiMation drivers (such as for OpenGL), do this culling / lighting / transformation stage themselves, allowing for hardware geometry acceleration.

Once all the faces have been drawn for the geometry object, the display process loops through any instances attached to the geometry object.

When the last instance in a view has been processed, the renderer is given the opportunity to flush any buffered polygons (for example, any transparent polygons should be left until last), and any per-frame tidy ups required.

That is the end of `RTDisplayView()`, but we still have not got an actual picture on the display. This final stage is executed by called `RTSwapPage`, supplied the channel handle. This executes the command to take the off-screen rendered 3D image and make it visible.

This is not part of `RTDisplayView()`, as a channel can have several views attached, and so will want to do several calls to `RTDisplayView()` with only one call to swap the page. Another reason for not incorporating the swap page in the display view call is so that 2D overlays (text, cockpit displays etc.) can be drawn in the offscreen buffer, and then everything made visible in one step.

One important point is that instances and geometry objects can have their own set of display overrides (e.g. pre-illuminated, shading mode, hidden surface removal, time, etc.). These are handled automatically as part of the display pipeline by effectively pushing and popping a state stack, giving a huge amount of control over the speed and visual effects possible with RealiMation just by altering object data.

Summary

The RealiMation scene graph - "RealiBase" - provides a simple yet powerful way of managing real time 3D databases in a multi channel, possibly multi-machine, environment. The data driven handle based interface minimizes API calls and allows for easy extensions in the future, both by adding new connections and new object types.

The RealiMation display pipeline traverses only those parts of the scene graph that will have an effect on the final display, and uses data contained within instances of objects to affect performance and visual effects.