

# Caligari Software License Agreement

This License Agreement is your proof of license. Please treat it as valuable property.

This is a legal agreement between you (an individual or an entity), the end-user, and Caligari Corporation. If you do not agree to the terms of this Agreement, promptly return the disk package and accompanying items (including written materials and packaging) to the place you obtained them for a full refund.

## License

*1. Grant of License.* This Caligari Software License Agreement ("License") permits you to use one copy of the Caligari software product ("Software") on any single computer, provided the Software is in use on only one computer at a time. If you have multiple licenses for the Software, then at any time you may have as many copies of the Software in use as you have licenses. The Software is "in use" in a computer when it is loaded into the temporary memory (i.e. RAM) or is installed into the permanent memory (e.g., hard disk, CD-ROM or other storage device) of that computer. However a copy installed on a network server for the sole purpose of distribution to other computers is not "in use." The Software may be executed from a common disk shared by multiple computers provided that one authorized copy of the Software has been licensed for each computer executing the Software. If the Software is permanently installed on the hard disk or other storage device of a computer (other than a network server) and one person uses that computer more than 80% of the time it is in use, then that person may also use the Software on a portable or home computer. If such person's authorization to use such computer ceases for any reason (e.g., termination of employment), then such person's authority to use the Software on a portable and home computer shall terminate.

*2. Copyright.* The Software is owned by Caligari or its suppliers and is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the Software like any other copyrighted material (e.g., a book or musical recording) except that you may either (a) make one copy of the Software solely for backup or archival purposes, or (b) transfer the Software to a single hard disk provided you keep the original copy solely for backup or archival purposes. You may not copy written materials accompanying the Software.

*3. Other Restrictions.* This license is your proof of license to exercise the rights granted herein and must be retained by you. You may not rent or lease the Software, and you may not transfer this license to any other recipients. You may not decompile, or disassemble the Software.

*4. Multiple Media Software.* If the Software is provided to you on multiple media (e.g., CD-ROM, DVD or Download Install File), then you may use only the media appropriate for your single designated computer or network server. You may not use the other media on another computer or computer network, or loan, rent, lease or transfer them.

## Limited Warranty

*Limited Warranty.* Caligari warrants that the Software will perform substantially in accordance with the accompanying written materials and will be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of receipt. Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

*Customer Remedies.* If a defect in the Software appears during the warranty period, Caligari's entire liability and your exclusive remedy shall be, at Caligari's option, either (a) return of the purchase price or (b) repair or replacement of the Software that does not meet Caligari's Limited Warranty and that is returned to Caligari with a copy of your receipt. This Limited Warranty is void if Caligari determines that the failure of the Software has resulted from accident, abuse or misapplication. Any replacement Software will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. Some of these remedies and product support services offered by Caligari may not be available outside the United States of America.

*No Other Warranties.* Caligari disclaims all other warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the Software, the accompanying written materials and any accompanying hardware. Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you.

*No liability for Consequential Damages.* In no event shall Caligari or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use or inability to use the Software, even if Caligari has been advised of the possibility of such damages. Caligari's cumulative liability to you, or any other party, for any loss or damage resulting from any claims, demands, or actions arising out of or relating to this License shall not exceed the license fee paid to Caligari for your use of the Software. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

## U.S. Government Restricted Rights

The Software and documentation are provided with Restricted Rights. Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause as DFARS 252.227-7013 or subparagraphs (c)(1), (2) and (3) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19 as applicable. Contractor/manufacturer is Caligari Corporation, 1959 Landings Drive, Mountain View, CA 94043.

This agreement is governed by the laws of the State of California without regard to conflict of laws principals, and venue shall be in Santa Clara County, California. For more information about Caligari's licensing policies, please call Caligari Customer Service at (650) 390-9600, email [custsrv@caligari.com](mailto:custsrv@caligari.com), or write to: Caligari Customer Sales and Service, 1959 Landings Drive, Mountain View, CA 94043.

No part of this document or software may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, and information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the express written permission of Caligari Corporation.

### Disclaimer of Warranties and Limitation of Liabilities

This documentation and the associated software were prepared by Caligari Corporation, and are subject to change without notice. While the authors feel that reasonable care has been taken in developing the program and preparing the documentation to ensure its accuracy, Caligari Corporation assumes no liability resulting from any inaccuracy or omissions contained herein or from the use of the information or programs contained herewith.

Caligari Corporation makes no expressed or implied warranty of any kind with regard to these programs or the supplemental documentation in this manual. In no event shall Caligari Corporation be liable for the incidental or consequential damages in connection with or arising out of the furnishing, performance, or use of this program or documentation. This disclaimer includes, but is not limited to, any loss of service, loss of business, or anticipatory profits, or consequential damages resulting from the use or operation of this software.

Caligari, trueSpace7 trueSpace6, trueSpace5, trueSpace4, trueSpace3, trueSpace2, trueSpace, trueSpace/SE, true - Clips, gameSpace and iSpace are all trademarks of Caligari Corporation. All other trade names and trademarks are the property of their respective owners.

Software Concept and Design: Roman Ormandy

Key Developers: Vladimir Sisolak, Martin Kusnier, Michal Valient, Pavol Elias, Anton Mateasik, Peter Malcovsky, Roman Ruckschloss, Tomas Bujnak, Martin Bujnak

Documentation Editor: Carrie Perez  
Assistance: Shengche Hsieh, Jeff Wall

Additional Chapters: Roman Ormandy, Norm Fortier, Tom Grimes  
Production: Bibiana Ormandy

We appreciate the invaluable assistance of our beta testers, the members of our user groups and TSML, and would like to thank the following individuals in particular: Paul Ashington, David Bokon, Saul Greenberg, Joseph McPeak, Paul Woodward, Marcel Barthel, Mike Harris, Anthony Ware, Stan Slaughters, John Logan, Richard Bolman, Anthony Owen, David Froude, Pete Massaro, Stefan Giurgiu, Roland Strålberg, Clinton Reese, Christiaan Moleman, Andy Davis, Mikko Ronkainen, Oliver Röhe, Rick Notaro, Zac Mansfield, Riccardo Scanu, and Lars Nilsson

Caligari Corporation      Phone    650-390-9600  
1959 Landings Drive      Fax      650-390-9755  
Mountain View, CA 94043  
USA

Sales Email:      [sales@caligari.com](mailto:sales@caligari.com)

Support Email:      [support@caligari.com](mailto:support@caligari.com)

# **trueSpace7**

## **Developer Guide**



# Contents

|                                           |            |
|-------------------------------------------|------------|
| <b>1 DESIGN PRINCIPLES</b>                | <b>1</b>   |
| 1.1 INTRODUCTION – METAPHORS WE LIVE BY   | 1          |
| What is Rosetta?                          | 2          |
| What is a Context?                        | 4          |
| 1.2 ROSETTA DIMENSIONS                    | 7          |
| Aspects of Interaction                    | 10         |
| 1.3 CONCEPTS & DEFINITIONS                | 12         |
| Concepts & Design                         | 12         |
| Rosetta Implementation                    | 18         |
| User                                      | 21         |
| User Interface                            | 22         |
| <b>2 WIDGETS</b>                          | <b>29</b>  |
| 2.1 LOW LEVEL ASSEMBLY                    | 29         |
| Description of widget package nodes       | 61         |
| <b>3 MATERIALS</b>                        | <b>75</b>  |
| 3.1 ADVANCED MATERIALS IN TRUESPACE       | 75         |
| Building Advanced Materials               | 75         |
| trueSpace Material Types                  | 76         |
| 3.2 USING THE MATERIAL EDITOR             | 82         |
| DirectX Materials in the Material Editor  | 82         |
| 3.3 WRITING SHADER SCRIPTS                | 99         |
| Writing DirectX Shader Scripts            | 99         |
| Writing Super LightWorks Material Scripts | 121        |
| <b>4 SCRIPTING</b>                        | <b>131</b> |
| 4.1 SCRIPTING IN TRUESPACE                | 131        |
| 4.2 SCRIPT COMMANDS                       | 139        |
| 4.2 SCRIPT OBJECTS                        | 155        |
| <b>5 FILE FORMATS</b>                     | <b>167</b> |
| 5.1 DX IMPORT                             | 167        |
| 5.2 DX EXPORT                             | 168        |
| 5.3 CONITEC A6 MDL EXPORT                 | 171        |



# Chapter 1

## Design Principles

---

### 1.1 Introduction – Metaphors we live by

Over many years and after much design work on countless generations of trueSpace, I have developed a mistrust of traditional object oriented (OO) software design techniques. I have observed the everyday experiences of our real-life interactions and have found them to be far richer than the idealized fixed hierarchy of the C++ abstract classes that today's software designers live by. Programmers always try to interpret the rich variations of meaning in all the different human contexts as some sort of aberration, an exception to the perfect order of their C++ hierarchical semantic trees.

What has also bothered me is that all of these hierarchical designs put observers outside of the hierarchy itself, or rather, outside of the “system” as the software designers will often say. In doing so they give themselves god-like powers and an independent “objective” observer status, as though they themselves and their beliefs (and they hold many beliefs) are not also part of the context. If you argue with them they will inevitably claim that the arguments they hold are “logical”; elevating their beliefs over yours to the coveted status of “fact”.

I am convinced that humans and our belief systems can be a part of any “reality” that we choose to live in. Our beliefs are inseparable from the contexts within which we act out our roles, whether the context is a theater play, broadcast TV news, a mass, or a scientific debate.

What “reality” ultimately becomes is a consensus of the beliefs of participants within a specific human context. This context is always established by a human founder who somehow convinces his disciples about the “truth” of his vision, whether it is a religious prophet or the founder of a new scientific theory.

There is no link between human concepts and objective things existing “out there” or outside of human existence. This does not mean that I deny a world “out there” exists, but rather that any structure and semantics attributed to this “outside world” is always a creation of the human mind.

Do not be afraid that the human mind will invent an arbitrary absurd reality. After all, our minds have evolved similarly over millions of years of collaboration together in shared contexts. The human mind can create wonderful and complex contexts such as religions, arts, sciences, economies, games, even the internet. We can also move between different contexts and in doing so we understand that we have to behave differently, obeying the rules of each context we enter.

We acquired (invented?) language, which is fundamental to many aspects of human culture and without which so many contexts would not be possible. Language gives us a fundamental tool to move from one context to another and to even create new contexts out of existing ones. This is called Metaphor.

When we say “Time is money”, we take an existing concept (Time) and give it a new dimension which in this particular case will become a new prototype in a specific (western) culture. Once established, this new prototype changes context in important ways. For example, we can say that “we do not have time” or that “time is being wasted”.

Metaphors (much preferred by myself over class-like definitions) create prototypical objects, which when accepted by context participants can be used with minor variations over and over. We call copies delegates. The relationship between prototypes and delegates in trueSpace is similar to the relationship between classes and instances in traditional object oriented (OO) design but with an important distinction: both prototypes and delegates are concrete objects which live in a concrete context, unlike disembodied classes which live in a platonic heaven (together with their software creators).

The most important feature of prototype/delegate design is that it does not assume any pre-ordained order. It is not based on “objective reality”, nor is it hierarchical. It evolves as participants expand their contexts with new meanings and create new contexts.

This parallels the architecture of trueSpace which aims to facilitate navigation between user created contexts, expansion of these contexts, and the creation of new contexts.

These are my beliefs and I fervently hope that they will convert many of you into disciples. If on the other hand reading the section above seems like wasted time, do not let that deter you from pressing on.

Roman Ormandy

## What is Rosetta?

Rosetta is a code name for the next-generation architecture for trueSpace, one on top of which future versions of trueSpace will be built. It is a distributed, object oriented collaborative online platform, which over time will enable many 3D applications, hundreds of application developers, and thousands of content creators like yourself to collaborate over the web in real time photorealistic 3D space. In order to deliver these features, Rosetta provides a robust message-passing kernel designed to take advantage of upcoming multi-core CPU's connected via Internet. This kernel is already employed as a transfer mechanism for trueSpace's new network physics engine which actually runs faster as more participants enter shared space.

Let us discuss some of the more important Rosetta concepts:

## Objects

The closest analogy for a Rosetta object is that of a living cell comprised of a membrane enclosing proteins inside the cell's cytoplasm and DNA inside the cell's nucleus. A Rosetta object is a **Container** (membrane), which encapsulates **Methods** (DNA) and **Attributes** (proteins). Some of the attributes are exported to the outside of the object container, thus they comprise the object's **Interface**, i.e., the part that other objects, including human participants, interact with. The only way to communicate with a Rosetta object is to send a message to one of these exported attributes, or **Connectors**.

Rosetta objects can be joined together with **Links** to create complex object assemblies designed to perform specific tasks. Such assemblies can be encapsulated, named, and their attributes exported from the inside to provide an interface for interaction with the outside world. Once created, this new object becomes a prototype of a new category but behaves exactly like any other object. In this way very complex object systems can be created.

## Dependency Graph

A Dependency Graph is a live representation of the structure of Rosetta objects and links. It is a dimensionless graph of nodes and links which is maintained by the kernel.

## Commands

Rosetta provides a message-oriented system for object-to-object communication and user-to-object communication. Each operation requested by the user (participant) in Rosetta is processed by a unified mechanism. This mechanism allows for the logging and queuing of user requests, their synchronization over a network, and the implementation of general support for unlimited undo and redo, even for 3<sup>rd</sup> party developers and scripts.

User requests are encapsulated in objects called Commands. Rosetta provides a basic set of these objects which can be thought of as Action requests. New commands can be incorporated into the system by 3<sup>rd</sup> party developers in packages or can be created by scriptwriters. Also, new commands can be defined visually in the Link Editor by combining existing commands into more complex structured commands and activities.

## Activities

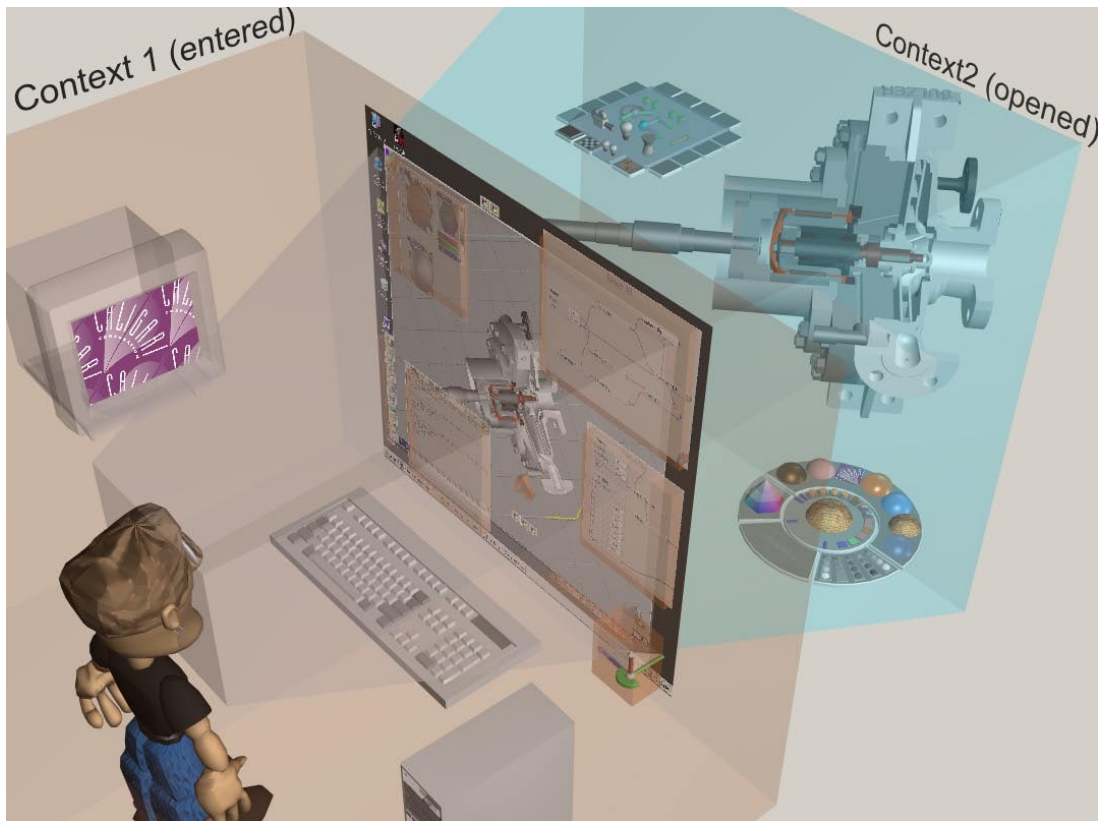
An Activity is a building block for all Rosetta based applications. In Rosetta, activities are sequences of actions with a duration in **time**. Unlike animations (even procedural animations), activities have a **goal**, which the activity participants try to accomplish. If the participants are human users then the activity can be a task like painting a texture, playing a simple game, or a collaborative modeling effort with other trueSpace users.

The Rosetta Activity Editor (based on the Link Editor) provides flexible editing tools that allow you to create and modify activities in a convenient way. Activity support allows for the creation and modification of behaviors applied to smaller or larger contexts, such as game rules, physical laws, and manipula-

tion access. You can also create behaviors associated with specific objects (models) like driving a vehicle, walking an avatar, or creating an engine simulation. Also, the Activity Editor will assist you in constructing compound and scripting commands. Combining behaviors with objects is a powerful new way to create complex procedural animations and interactive behaviors. It gives trueSpace objects life.

## What is a Context?

The trueSpace concept of a scene is extended in Rosetta to the concept of a **Context**. A context is a state of an **Activity** that usually includes a space where one or more users interact with objects and each other together with a set of rules governing this interaction. This is similar to a video game level and the particular role an individual player may enact. The **context** is an object just like any other Rosetta object; the main difference being that one or more users are **inside** of it, much like players in a particular video game level.



Everything in Rosetta is context dependent and the **meaning** of every action may change as the user moves from context to context. For example, you can control a tricycle object in a street context by rotating its pedals and steering its handlebars. By right-clicking on the tricycle you can exit the “street”

context and enter a “repair-shop” context where you can modify the tricycle by adding, for example, a bell to its handle bars.

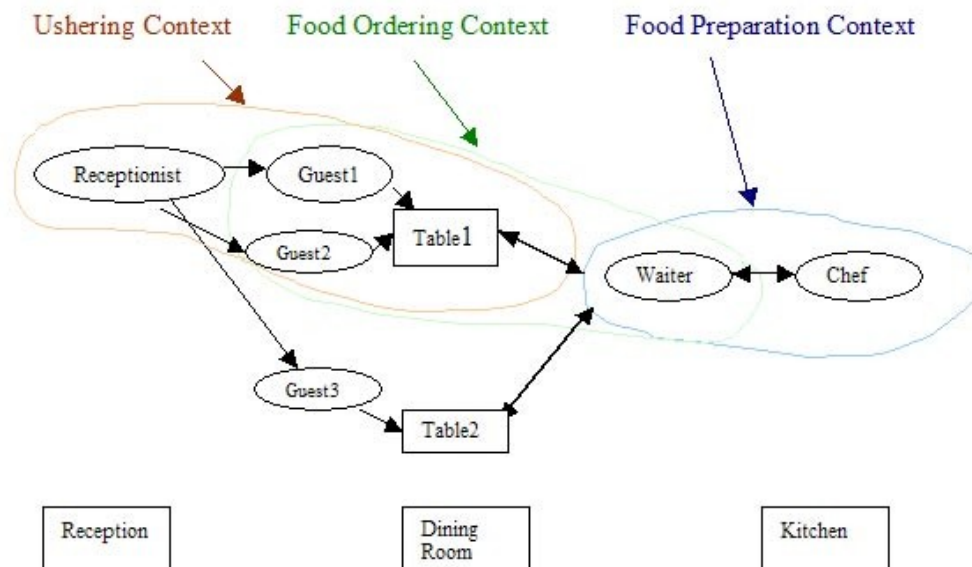
Let us look at an example of concrete, real world contexts and how they participate in a concrete real-life scenario:

## Restaurant Scenario

### *Time Structure:*

Guests arrive at a restaurant. A receptionist take their coats, ushers them to a dining room and seats them at a table. A waiter hands them a menu and takes their order. The guests talk to each other when the waiter leaves. The waiter goes to the kitchen and conveys the order to the chef. The chef cooks the meals in the oven from ingredients and when finished, hands the plates to the waiter. The waiter serves the guests their meals. The guests eat (and talk) and when finished, the waiter brings the bill. The guests pay the bill and go to reception. They pick up their coats from the receptionist and leave.

### *Dependency structure:*



Please note that each of the 3 contexts above includes a space (sometimes more than one) and a selection of participants (actors), in this case guests, receptionist, waiter, and chef. As we will see from the following examples, human participants and their roles are very important when it comes to establishing the meanings of objects and actions within a context.

## Attributes and Methods Depend on Context

Most object oriented software has an absolute distinction between object attributes (outside interface) and methods (inside structure). In trueSpace this distinction is relative to a context, specifically the roles of participants within contexts.

Let us take a knife and scissors objects as an example. They are similar objects but serve different purposes and thus they have different attributes and structure. Actually, I would hazard a guess that scissors evolved as a symbiosis of two knives.

The knife is one of the oldest of man-made objects. Even ancient hand axes carved out of stone have two separate attributes, handle and blade. Let us take a simple kitchen knife: I am a Cook, I grab the handle with my right hand (“handle-hand”) and press the blade against an onion lying on the kitchen counter to cut it in half. The handle is an object attribute which is the same as a connector and an interface. At least with a knife, it is clear that it is the same thing, with software objects though, the situation can be much more confusing,

As a Cook I do not care how my knife is made, I do not care about its inner structure. As a Knife Maker I see that its metal blade extends into the handle which is actually two wooden pieces on each side connected with three rivets. The rivets go through holes in the blade and hold everything together, and I as a knife maker would use some machine tools to put the knife together from its parts in my workshop. The cook could say that the knife maker is interested in the knife’s structure. But it is all relative. Take scissors as another example. A seamstress uses scissors to cut a piece of cloth. She puts the fingers of her right hand into two opposing holes in the scissor handles, puts cloth between the blades and skillfully cuts a nice pair of pants out of the cloth. She does not care about the structure of the scissors, but she does care about the structure of the pants which she puts together with other tools (like a needle and thread). Scissors Designers see right away that scissors are nothing more than two knives with opposing blades connected in the middle by a 1D rotational joint. Over time, both the blades and handles of “half scissors” knives have had the value (shape) of the original knife like attributes modified to better serve their new purpose of cutting cloth rather than onions.

So what is the moral of the story? On one hand we see that each object has outer attributes to interact with and an inner structure which shows how it is put together. Through editing (of structure) we create complex objects out of simple ones. On the other hand, the difference between Attributes (Use) and Structure is not absolute and depends on the context and role of the participants within it. The Seamstress’ structure is the Scissors Designer’s use. If you take participants and their roles within contexts away (like in a dictionary definition or hierarchical classification), the distinction between structure and attributes seems to be absolute.

The role of a cook or seamstress may not seem relevant here, but it is easy to substitute roles for a Modeler, Surfacers, Rigger, Animator and Renderer with the same result: the Modeler builds and edits a shape structure, encapsulates it, exports some attributes which will become elements for Surfacers who edit the shape’s materials. A rigger adds a skeleton to surfaced objects and passes it on to an Animator. For a Rigger, the skeleton is the structure (a method) to be edited, for an Animator it is just a bunch of attributes

(UI controls) that he can use to create “real” content, i.e. animations, which are structure to him but not to the Rendering guy and so on....

In my experience, most of the problems in everyday human collaboration arise when participants of one context assume that the meanings they attribute to objects in their local context are the same in other contexts and for different participants with different roles. Have you ever wondered why developers of some software applications have such a hard time understanding users of their own software? The answer lies in the very different roles they assume with regard to a seemingly identical object.

---

## 1.2 Rosetta Dimensions

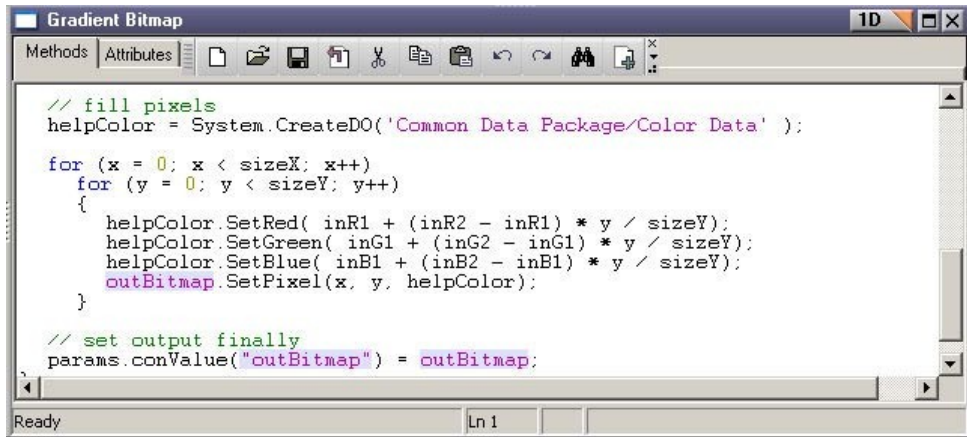
We perceive our world as 3D, but 2D and 1D representations play an important role in many contexts. Artists prefer a 2D user interface (UI) for image and vector based graphics tools, and developers prefer 1D text editors for writing code. Today, most authoring tools hardwire a particular dimension for most, if not all user interactions. For the first time, trueSpace gives you, the user, control over what dimension you deem best for a particular task. In principle, every truespace object can present a UI aspect in any dimension you choose. All trueSpace editors (Script Editor, Link Editor, and 3D Editor) allow for a uniform and explicit way to move in, out, and between objects; effectively leaving and entering contexts at will and without getting lost.

On the outside, each Rosetta window has a 2D border simply because it is part of standard MS desktop. On the inside, a Rosetta window can display its content in 1, 2 or 3 dimensions. Dimensions are a fundamental part of all aspects that Rosetta objects can display.

Let us look at all 3 dimensions of Rosetta editors:

### 1D Script Editor (SE)

This is another name for the textual (symbolic) representation of an object. Here the text cursor can only be moved in one dimension, from the beginning to the end of a text string.

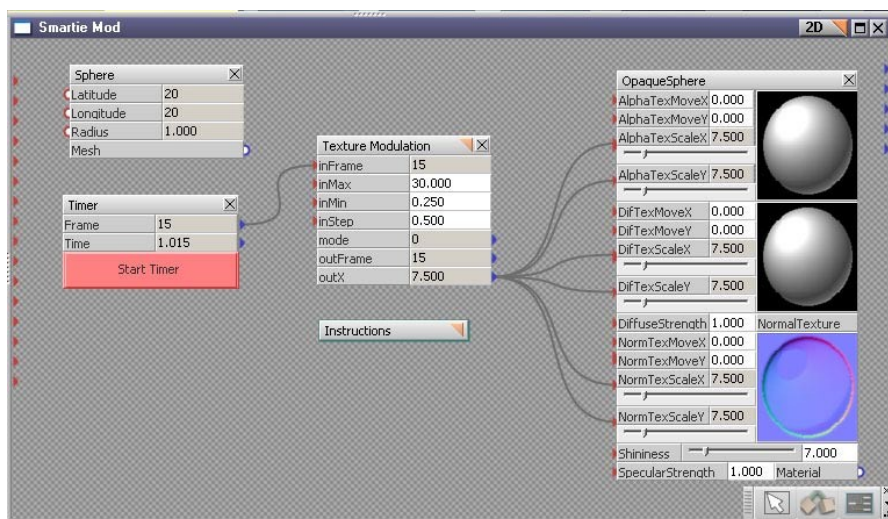


This is how program objects were traditionally designed. In Rosetta, a developer can create a new object in C, C++, JAVA or any other language using a favorite programming environment and then compile it into a binary COM Rosetta object. In addition to this, users can create script objects at run time using a scripting language of their own choice, such as VB, JavaScript, LUA, and HLSL. In principle there is no difference between script objects and compiled objects except for speed. They are all first class Rosetta citizens. Similarly, there is no real difference between objects created by in-house 1st party Caligari developers and 3rd party outside programmers.

## 2D Link Editor (LE)

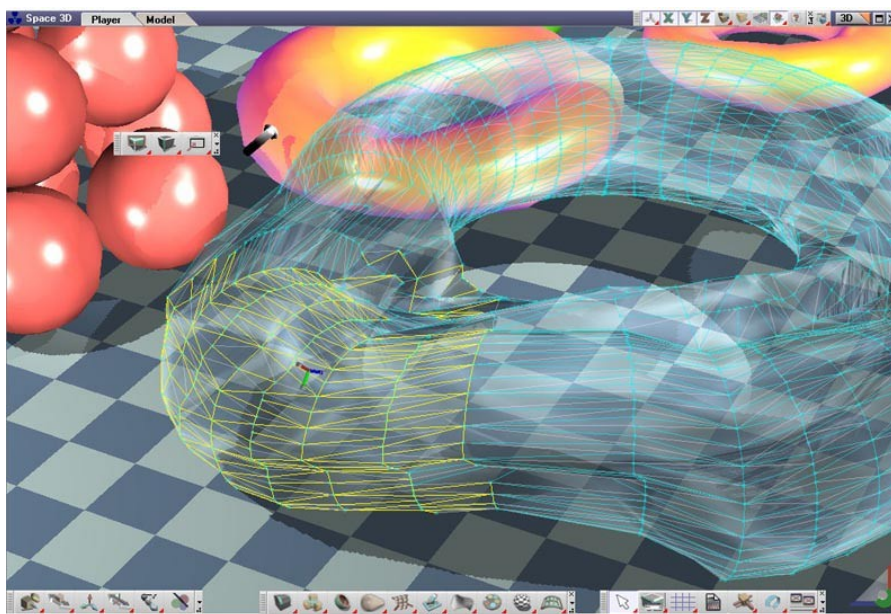
If programming or scripting is not your cup of tea, you can still create first class complex Rosetta objects, including procedural and interactive objects, in the 2D Link Editor by simply dropping existing objects from libraries into a 2D LE window, linking them with visual links, and then encapsulating this 2D structure as a new object with a new name and a new set of interfaces. In this way you can create any kind of object, shape, new material or interactive behavioral object. Objects created in this manner are regular Rosetta objects and can be opened and edited (in most cases) not only in the 2D LE but also in the 1D text editor or the 3D Modeling Editor.

In the 2 link editor, the cursor can move in two dimensions, usually denoted as X and Y.



### 3D Modeling Editor

Though best suited for shape modeling, the 3D Modeling Editor does not prevent you from linking in - teractive 3D objects using widgets, joints, and other constraints into more complex 3D assemblies and encapsulating them, just like you would do in the Text Editor or the Link Editor.



All of this is possible because the **object and its interface** (including the User Interface) in Rosetta **are separate**. Rosetta objects can in fact possess a number of different interfaces, or **aspects**. You as a user can decide which particular aspect you wish the object to display based on the current context and your current needs or purpose. For example, a **shelf** object display can be changed from a **1D menu** to a **2D Toolbar** with a single click. The choice is yours. You can even decide to create an entirely new User Interface aspect and add it to an existing object at runtime, using the **Panel Editor (PE)**.

In addition to the Link Editor, whose primary purpose is to edit the structure of “noun-like” objects (things), there is also a **2D Activity Editor (AE)** whose purpose is to edit “verb-like” objects (behaviors) in a **time** dimension. Currently the Activity Editor shares the Link Editor window. You can differentiate its structures by the green links.

## Aspects of Interaction

Objects do not exist in a vacuum. In Rosetta, it is always you, the user-participant and your interaction with an object that determines how the object appears. Without your presence, your purpose, your seeing eye, and touching hand, the object could not exist. As in real estate, location is everything. You can either be outside or inside an object.

When you are **outside** of an object, all you can see are the attributes that have been exported in the form of a particular **interface**. You choose the interface suitable for your present **purpose** from the available aspects of an object.

We say that an object is in a **Minimized** state when it only displays its name. In 1D this is a text string. In 2D it is an icon. We say an object is **Expanded** if all attributes of the current aspect are visible.

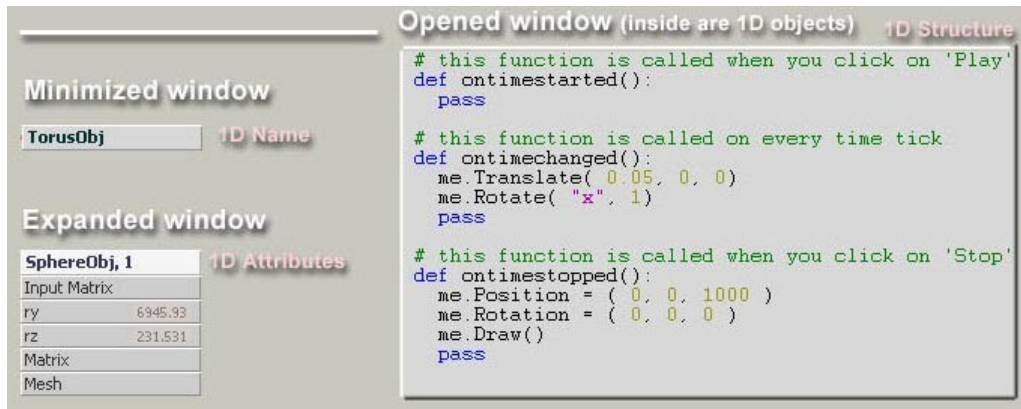
You, the user, also do not exist in a vacuum. In fact you must always be located **inside** of some Rosetta object. That object, together with its current state, is called a **Context**. Context is not static though. You can always decide to exit the current context and enter another one, such as when you open the door in your living room and step through it into the kitchen.

In Rosetta you can open or enter an object, often simply by right-clicking on it. This simple action has profound consequences though. If you **Open** or **Enter** an object, you are, in fact, stepping inside of it and changing your context.

Something interesting then happens. Once inside an object, you no longer have to interact with the object’s interface. You can see and interact directly with the internal object **structure**. To change this structure, Rosetta provides you with powerful editors such as the Link Editor or Script Editor. Depending on the dimension, these editors will assume a different form.

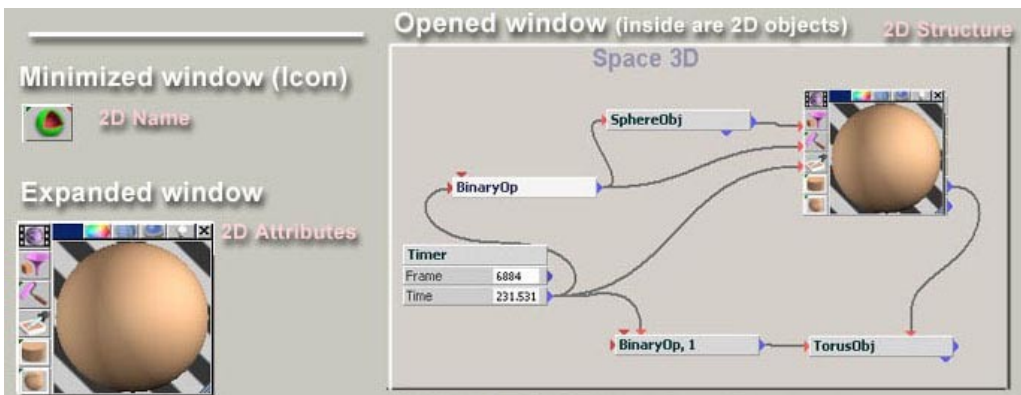
## 1D Interaction Aspects

The **Minimized** aspect in 1D is simply a text string denoting the name of an object. The **Expanded** aspect adds a list box with the names of all attributes for its current interface. The **Opened** aspect in 1D is a **Script Editor** that allows you to edit an object's internal methods (scripts).



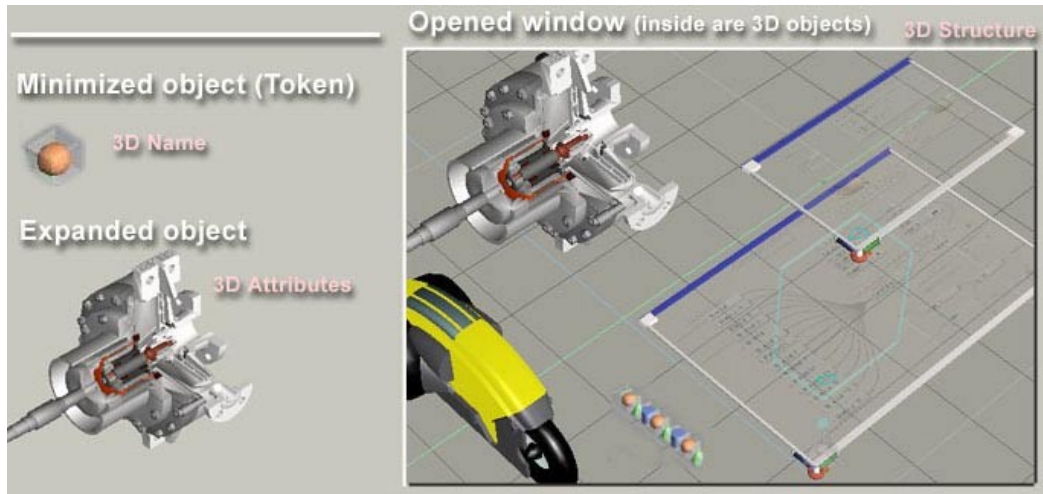
## 2D Interaction Aspects

The **Minimized** 2D aspect is a bitmapped Icon. The **Expanded** 2D aspect can be a 2D control panel or maybe a library. The **Opened** 2D aspect can be a **Link Editor** or **Activity Editor**.



## 3D Interaction Aspects

The **Minimized** 3D aspect is a simple polyhedral token. The **Expanded** 3D aspect reflects position, shape, and surface attributes such as diffuse color etc. The **Opened** 3D aspect is our familiar 3D **Modeling Editor**.



### Open/Enter/Exit/Close Metaphor

In order to edit an object in Rosetta you have to open and enter it first, then after you do some editing, you exit and close the object. When you enter an object, it appears “maximized” i.e., it zooms to fill the active window, but it is important to realize that the object itself did not change. It only appears maximized because you the user changed your location.

The power of this metaphor stems from the fact that you can move from one context to another context based on your current purpose alone. There is **no predefined hierarchy** in which contexts are organized. As the author of a particular object in Rosetta, you can create links to whatever other contexts you choose, just like links between Web sites.

The remainder of this chapter is a collection of some of the most important Rosetta definitions and concepts.

---

## 1.3 Concepts & Definitions

### Concepts & Design

These are most important terms for Rosetta’s design. They explain what Rosetta consists of.

#### What is Rosetta?

##### Object (Encapsulator)

A container with connectors (attributes); the container specifies the boundary between the inside and outside of an object.

Only attributes are visible from the outside, to see its internal structure one must enter inside of the object. Interestingly, the English term “outside” is derived from “Out of Sight” and “inside” from “In Sight”. There are two kinds of objects: things (nouns, like a cell, knife, chess piece), and activities (verbs like dance, battle, traffic).

**Attribute (connector)**

A Visible property with a value produced by one of the object’s methodS, which other objects can interact with (cell’s proteins).

Some common attributes are location, shape, (knife handle) and behavior (participant’s character). Attributes are grouped together and are accessible to the user via interfaces (aspects) based on user purpose. The term Connector is used for LE representations outside of specific object aspects. The art of interface design is based as much on what the interface hides as in what it reveals.

**Link (Medium, channel)**

A Medium delivers signals from a sender object to a recipient object.

Medium is a physical matter with a single force (wire with electron flow, space with light, cell with molecular flow, neuron’s axon with ion+ charge flow).

**Structure**

The visible (non-encapsulated) network of objects and links connecting them; equivalent to a method

**Method**

A procedure (function) with input and output messages (attributes). Methods are equivalent to an object’s inner structure (cell’s DNA, gene).

**Signal (Data Object)**

A Data Object (sign) is sent from a sender to a recipient.

**Message**

The meaning of a signal sent from a sender object as understood by a recipient object

**Event**

The generation of a message by an object’s method at a point in time

**Time**

A sequence of events observed by a participant from a viewpoint in the context

**Object State**

The values of an object’s attributes at a point in time.

**Action**

Execution of a method and state change with negligible time length; can not be subdivided.

(An object is selected or copied → face turns red)

### **Goal**

The state which an object is motivated to reach)

(A door wants to be closed, an organism trying to reach a sufficient level of liquids, John wants to be a president)

### **Activity (Behavior)**

A sequence of object actions with a measurable time duration and aimed at reaching a goal.

Among simple activities are gestures (mouse LClick and Drag), medium level skills (shooting, walking) and complex activities (Open the door, enter store, buy milk, drink it).

### **Social Activity**

Activities of multiple motivated (purposeful) interacting participants, coordinated in time and with a beginning and end. This is what Rosetta is all about.

(Dinner conversation, paying a bill, game playing, flirting, 100 meter dash)

### **Role**

The goals and rules of behavior for an object participating in an activity

(In soccer, a forward tries to score the goal, he can only kick the ball. The goal keeper tries to save the goal; he can kick the ball and catch it with his hands)

### **Actor**

An object with a role and a purpose (motivation) in an activity

(The ball in a soccer game, enemy bot in a video game)

### **Participant (user)**

A human actor (buyer/seller in a transaction, student/teacher in class, judge/plaintiff/defendant in court, pirate/sailor in a movie)

### **Scene**

An object entered by a participant where an activity occurs; a place of interaction. It is a user presence which defines a scene. The scene is the focus of a participant's attention, i.e. the end point (termination) of their senses (this may be different from a participant's physical location).

### **Context**

The state of an activity; it encompasses the state of a scene and the state of participants as well as the rules of behaviors for all of the actors.

Context is the present reality for participants. Thus reality, the rules of behavior, the meaning, and the truth change with the context also.

(A CPU and the values of its registers, an egg cell with DNA from both parents and proteins from the mother, an animal in the forest, a factory with workers, tools, and parts, a classroom with trainer, students, and learning objects, a design studio with customers, designers, and prototypes, an MS Word editor with story writer and document, the trueSpace workspace with modeler and objects in 3D space)

Augmented participant senses (X-ray, telescope, TV) define context boundaries. A participant with X-ray vision can see and inhabit different objects compared to a participant with normal vision.

### **Language**

Human communication/speech patterns in the form of sounds or written script

### **Conversation**

A social language activity where two or more participants speak or write to each other

A large portion of, if not all, everyday conversation serves to create and reinforce the reality of current context in the minds of participants.

### **Purpose (motive, intent)**

The use of a specific aspect of an object by a participant to obtain some goal motivated by its role in the current activity

(For the purpose of paying a bridge crossing toll a moped is the same as a motorcycle, for the purpose of vehicle insurance, it is not)

### **Culture**

A belief system adopted by participants justifying the purposes for all roles in a context

(Buddhism, Christianity, Islam, western rationalism, quantum physics, programming, environmentalism, free market conservatism, communism, feminism, surrealism. A hero saves a child because it belongs to his context. A terrorist hurts the child because she does not. One culture's terrorist is another culture's hero.)

### **Community**

The participants of a context sharing a common culture

(A parish, skinheads, quantum physicists, trueSpace users, Rosetta programmers, Palestinian villagers, A CNN reporter talking with them is not a member of their community)

### **Founder**

A participant, (prototype) who creates a new belief system which other participants then follow

He must convert other people to this new belief. (Jesus Christ and his disciples, Charles Darwin and his followers, Albert Einstein, inventor of baseball, a company founder)

### **History**

A sequence of past events leading to current the object state; an object's memory

(Construction history in modeling context)

### **Scenario**

The recorded history of a social activity

(Actual dining experience, sporting event, story, a movie)

### **Story**

A scenario created (recorded) by a context author for an audience of passive participants. Story usually reinforces the primary context (where the protagonist resides).

## **Why Rosetta Works**

Unlike traditional Object Oriented design, Rosetta does not rely on classes (abstract templates) but rather, prototypes (real objects) where semantics is based on the actual use within a specific context. (Wittgenstein, Lakoff).

### **Base Object (element, brick)**

A building block object (defining element), in a given context

In this context it can not be entered (though it does have a structure called *gestalt*) because context rules prohibit it.

(Joints in kinematics modeling, a chessboard in a game of chess, atoms in physics, molecules in chemistry)

### **Inner Object (Contained object, Sub-object)**

An object which is inside of another object, encapsulated within a linked structure and thus invisible from the outside

It becomes visible when we enter the context of the outer object. (Matches become visible when we "open" the matchbox).

### **Outer Object (Container, Owner)**

An object encapsulating inner objects

### **Part**

A visible object with permanent links to other objects within a context; parts of a structure can be grasped in one glance by a participant.

(A knee is a part of a leg, a leg is a part of a body, a blade is a part of a fan)

### **Prototype (parent)**

An encapsulated (novel) structure of two or more objects; an object which serves as a template for new objects in a given context

An object which is the “first”, or “best” representative of a new category; a founder or central member that is used the most by context participants and “stands for” the entire category.

(First car of a new production line, Bush stands for republicans, Xerox stands for a copier)

In an organic world, new prototypes are created by a **sybiosis** of two formerly separate organisms which become so dependent on each other that they can not exist separately (sym-bio-genesis).

(Mitochondria inside an eukaryotic cell, bacteria inside human intestines, a caterpillar and butterfly are symbiotic genomes sharing one body in time)

### **Delegate (child)**

An object derived from a concrete prototype within the same context. In some cases it can delegate by aggregating parts or all of its functionality to its prototype.

This is Darwin’s “Descent with modification” which says that all organisms living today are mutated (modified) descendants surviving through a process of “natural selection” (**bio-genesis**).

### **Inheritance Link (Aggregation)**

A link between a prototype (parent) and a delegate (child)

### **Category**

A prototype and its delegates, linked with inheritance links within a context.

A category is not a traditional class, as delegates are not identical to prototype and are always relative to context. Category members are more like family members. They resemble each other (they have very similar internal structures) but do not share a single common set of attributes.

“Bird” = eagle (best representative) → chicken → ostrich → penguin (worst representative)

### **Metaphor (analogy)**

An inheritance link where prototype and delegate are in different contexts; mostly used in language contexts

A metaphor creates a new prototype; *This* is like *That* but with *these* new features.

(A car is “a horseless carriage”, an argument is like “war”, this is the “Cadillac” of brushes, a fiscal train wreck)

Language activities are linked to each other with metaphors (Context interaction).

(Intel created new network based communication platform and called it PCI Express to help make the transition easier for engineers used to the term PCI bus)

### **Metonymy**

A metaphor where a part stands for a whole (“I have a new set of wheels” means “I got a new car”), or one thing stands for another (The Times hasn’t arrived at the conference yet - meaning the reporter from the Times).

### **Collaboration (symbiosis)**

A social activity in which two or more objects (actors) achieve their mutual (non-competing) goals through interaction; the participants give up some autonomy and achievement of part of their goals in exchange for benefits related to membership in the social group

(Volvox colonies, slime mold, religious communities, minority religions with terrorist suicidal bombers, project teams or sport teams, states with citizens, a company with employees and customers)

### **Competition**

A social activity in which participants of overlapping contexts are trying to reach the same goal

(Soldiers fight for land, girls fight for a mate, companies fight for market share, predators fight for prey, politicians fight for power, Quake players and hunters fight for the pleasure of the kill, soccer teams fight for victory, participants fight in a verbal argument)

## **Rosetta Implementation**

### **How Rosetta works**

This section describes the Rosetta system and its three parts: the Dependency Graph (DG), the User and the UI (viewers) that link the user and the DG.

### **Node**

The System metaphor for a Rosetta object

A node is represented by a COM object that implements a special communication interface (IRsUnknownNode and aggregates RsBaseNode object) allowing it to connect with other nodes. Rosetta nodes are maintained by the Dependency Graph.

**Link**

An ordered pair of node connectors with messages flowing from one to another

**Encapsulator**

A system metaphor for a Rosetta container

An Encapsulator is a node that hides some attributes of a group of connected nodes.

**Manager**

An object which maintains node context, i.e., a lifecycle of node states from its creation to its death.

**System (kernel)**

The Rosetta messaging kernel and database which manages an object's life cycle

The system connects all nodes (including participants), sends global messages, creates new object types, and serializes them.

**Dependency Graph (DG)**

A System metaphor for Rosetta structure maintained by the System; a dimensionless graph of nodes and links

**Link Editor (LE)**

A 2D View of some part of the DG

**Space3D**

A DG node encapsulating all 3D objects

**Scene (SceneGraph)**

A DG of the inside of a 3D object entered by a participant

It is the user presence which defines the scene, and without the participant's presence a scene can not exist.

**Connector**

A System metaphor for a Rosetta attribute with links to other connectors

Connectors are used to create and manage dependencies (links, connections) among attributes of different nodes. Interfaces define an object's Type.

**Aggregation (Delegation)**

One object taking advantage of the services offered by another object; causing this second object to appear as a natural part of the first object.

Aggregation means that the containing (outer) object creates the contained (inner) object as part of its creation process and the interfaces of the inner object are exposed by the outer object.

**Exported Connector**

A connector that is visible (belongs to an encapsulator like all other connectors), but its methods (GetValue, SetValue, Invalidate) are redirected to the same type of connector of an encapsulated inner object (the object which is exporting the connector)

**DataObject (signal)**

An object used to transfer signals between two linked node connectors

(Matter moved by a force, electron flow in a metal wire)

**Command (message)**

An encapsulated request (message) to an object for an action; also the name of the corresponding method executing the action

A command is an object which can connect to the DG, make a request, and disconnect from the DG. A command can come from the user or from other objects.

An object which knows how to execute an action, must also know also how to execute the inverse action.

**Command Generator**

A Node which can create Commands

**Direct Communication**

Two objects visible to each other through a single medium link

**Indirect Communication**

Two objects not visible to each other directly, but linked through exported connectors

**Import**

Conversion from a file external to Rosetta into a Rosetta object

**Export**

Conversion of a Rosetta object into an external file compatible with another application format

**Transport**

Transfer of a Rosetta object or library between two different Rosetta contexts

**Instance (clone)**

A delegate which aggregates all of its interfaces from the prototype; it only differs from the prototype in attribute values

**Binary object**

An object whose inner structure is frozen at compilation time and can not be displayed or edited at run-time

**Package**

A collection of Nodes, DataObjects and Commands with a special interface (IRsUnknownPackage) that simplifies the creation of objects in one container; a name space

It is also useful for installing efficiently in the system, but limits the ability to move the objects between contexts by the user.

**Scene**

A stored context

**User****User (Participant)**

A human actor with input connectors (Sensors) and output connectors (Effectors)

**Sensor**

A user input connector (Eyes, Skin, Ear); objects can also have sensors (proximity sensor)

**Effector**

Generates an action, i.e. visible behavior (muscles of eyes, hand, mouth)

**Sense**

A user's bi-directional interface of touch, sight, and sound; human specific transducers, which convert signals between two media, one of them inside of human body and the other outside

(Skin/hand, eyes, ears/mouth)

Senses interact with each other, an Eye-In/Hand-Out link provides important feedback for a participant (eyes tell you what to touch; to shoot someone "from behind" is cowardly). Rosetta simulates touch, sight, and to some extent sound, but only passes speech between participants.

- Touch detects mechanical collision (weak force, gravity). Touch is an extension of skin, hand, arm, etc. (mouse, direct manipulation), and communicates the collision of objects such as button clicks and slider drags used for manipulating objects (pet the dog, cut the wood).
- Sight transfers electromagnetic waves (light) in a participant's eyes to electrons in the brain, communicating shape and location of solid objects. Eyes (or a mouth) can also select objects i.e. send an outgoing message.
- Sound (hearing and speaking) transfers modulated mechanical airwaves (by weak force) from a mouth to an ear, communicating speech behaviors and the location of objects.

**Control**

A visible but view independent (2D or 3D) object that receives events from an I/O device (transducer) connected to a participant (mostly by hand) and re-directs them to the input of another object or generates a command (action).

- **Button:** Starts a one time action, usually on release of a mouse button.
- **Toggle (radio button, checkbox):** Changes an object's state, usually switching back and forth between two states.
- **Slider:** Starts a one time parameterized action with values between Min and Max as visualized on the slider.
- **Continuous Slider:** Calls an action repeatedly with different parameters, usually read from a Drag (dX, dY) during the input gesture. Continuous sliders are useful for real time direct manipulation. Slider control objects may control one or two real time inputs simultaneously (trueSpace 3D NAV widgets are mostly racks of sliders).
- **Brush:** Calls an action repeatedly during a mouse drag. Unlike a continuous slider, it reselects its target (under the mouse cursor) each time the drag event (dX, dY) is sent (e.g. tS 3D Paint).
- **Text Field:** Calls an action parameterized by a text string input from a sequence of keyboard events.
- **Number Field:** Like a text field but the string is interpreted as a number by the action parameters.

### Gesture

A user muscle activity combining I/O device events over time and sent to the surface of a control object as one message

Control objects can invoke different actions based on different gestures received on the same surface. Examples of mouse gestures are LClick, RClick, LPress, LHold, LRelease, Drag, and Double LClick. Gestures can be further modified with Keyboard events such as CTRL LClick etc.

Actions generated by control objects can be one time actions, parameterized one time actions, and continuous (real time) actions.

## User Interface

The Rosetta user interface consists of a collection of viewers and editors for displaying DG objects inside of standard MS Windows. They allow the user to move between contexts and to display each context in different aspects. Viewers can show 1, 2, or 3 dimensions and each DG object can display multiple aspects (interfaces) inside a viewer based on the user's current purpose.

We perceive the difference between the “normal” use of objects when we interact with an object’s interfaces in some (outer) context (riding a tricycle) and “editing”, when we change the object’s inner structure and enter its context by doing so (repairing a tricycle). We must always keep in mind that the difference is relative to context, i.e. a “rigging artist” creates a new rig (skeleton controls) by “editing” and then an animator “uses” the same rig for making the character walk. Each uses however, a different interface, i.e. aspects of the same object. (See the Attributes and Structure paper in the Sharepoint Concept folder).

## **Viewers**

### **Desktop**

This is the highest context in Rosetta. It is a collection of free floating or docked MS Windows each displaying a Rosetta viewer through which the user can navigate to all of the other contexts.

### **Window**

A 2D frame on a desktop inside of which a viewer displays a view of a particular context

### **View (Portal)**

The rendering of a context inside of a window

Context which is displayed in an active window is the active context where the participant’s focus is located at any given time. From this context other contexts may be visible and the user can navigate to enter them.

(In my living room in addition to 3D objects like a globe or a lamp, I have a large TV screen and a big picture window. They both are views or portals to a different context. I also have a framed painting on the wall, magazines on the table, books, and even a binocular on the bookshelves. A physician’s X-ray machine has a separate 2D view. The physician must move his eyes away from the patient and towards this view window to see a patient’s inner organs. There are also doors and drawers (connectors to other contexts) in my living room, which require that I physically transport all or part of my body through them, thus entering a new context).

### **Viewer**

A 1D, 2D, or 3D rendering manager that displays some aspect of a particular context inside an MS Window

### **View Aspect**

Views can have 1, 2, or 3 spatial dimensions and one time dimension. Views for each dimension can have several aspects, for example in 3D view, objects can be shown as solids or wire-frames and in 2D view links can be either visible or hidden.

### **Object Aspect**

A specific interface which an object displays inside a Rosetta view

An object can and typically will have more than one interface. It depends on the user's purpose in a given context and is set by the user. These are the most important aspects:

- **Minimized, (Closed, Icon):** Shows only the location and name of an object; useful in the Link Editor.
- **Expanded:** Shows all the visible attributes of an object.
- **Default:** "Out of the box" layout
- **Custom:** User created aspect

### Snapping

Snapping occurs when 2 objects are attached along their boundary. In 2D views, panels can be attached edge to edge, or corner to corner.

### Docking

Docking occurs when an inner object is snapped to the boundary of its outer object.

### Tool

An object used by a participant for the purpose of reaching the goal of an activity in a context.

(The paint tool changes the color of the surface, a knife can manipulate a weak force to split a solid object, the seductress was a mere tool in his diabolical scheme).

### Editor

A viewer plus a set of tools whose purpose is to change the value of an object's attributes or structure (methods themselves)

### Application

A collection of tools with the same overall purpose (e.g. Polygon tools) encapsulated and distributed in a package. Rosetta tends to dilute the importance of large legacy applications and break them into smaller groups of tools that are easily stored inside a library.

### 1D Viewers

Render phonetic text.

Even though actual letters are 2D shapes, strings of letters are linear structures and resemble 1D more than 2D.

### Script Object

An object whose inner structure can be displayed and edited at runtime in the script editor

### Script Editor (SE)

A 1D view displaying the text editor for the editing of an object's methods (structure) at runtime in a text format

## 2D Viewers

Viewers brought upon us by a printing press

### 2D Control Object Structures

Groups of control objects categorized by common semantics

- **Icon:** Minimized object aspect used to represent a control object visually.
- **Shelf:** A simple 2D container for storage of un-linked icons. All icons are of the same size and they stick to each other on the edges. Shelf containers always “snap” tight around the icons visible inside of them.
- **Library (Category):** A simple or complex shelf with icons that load objects into the current context or that store objects from the context in a library. Common two level layouts are Tabbed Panels and Task Panels, both of whose outer containers can be oriented horizontally or vertically. Pop-up - a library whose icons include a behavior (activated by LHold) which opens another library. Pop-up libraries can have icons displayed in one row or column only.
- **Toolbar:** A shelf with icons executing some action
- **Menu:** A 1D Toolbar
- **Panel:** A complex 2D container where inner objects can be of a different type and can include other panels or shelves. An example is a Control panel with tools (called a Rack in 3D aspect).

Panels and Racks in Rosetta employ the principle of Direct Manipulation. This means that the shape, material, location, and orientation of the control elements indicate by their visual metaphor the object's expected behavior (affordance).

### Link Editor (LE)

The 2D viewer part of the Dependency Graph; also a set of tools for editing the structure (links, methods) of objects by direct manipulation, as well as for creating new prototypes by linking existing objects into novel structures

### Panel Editor (PE, Interface editor)

A 2D viewer for creating and editing 2D control assemblies; each panel can have several aspects.

### Material Editor (ME)

A 2D viewer for creating and editing materials, shaders, and UV spaces

### Activity Editor (AE, Scenario Editor)

A 2D viewer for creating and editing activities in time

### 3D Viewers

Viewers simulating the physiology of human vision and experience by incorporating the physical forces of light (EM force), gravity and collision (weak force).

#### Light

Objects can emit photon messages. These bounce from object to object until they arrive in a participant's eye input connector.

#### Eye (Camera)

An object with a directional photon sensor; sensing photons emitted from lights that bounce around the scene and finally arrive in the eye

#### Solid Object

An object with a 3D surface that reflects light

It receives and outputs photon messages. The transport of photon messages is handled by a render engine.

Each Solid object has 3 attributes:

- **Shape:** The boundary attributes (2D or 3D) of an object (NURBS, meshes, curves)
- **Material:** The function specifying the surface reaction to impinging light (color, texture)
- **Transform (matrix):** The location, orientation, and size attributes of an object within a scene

#### Renderable object

An object with transform matrix attributes and shape attributes but no reflectance function. The output of photon messages is dependent on input messages.

(Orientation vector, joints, forces, wireframe widgets)

#### Constraint Object

An object capable of receiving and sending messages relating to the position, material, or shape attributes of solid objects.

(All joints, point to path constraint, lookAt)

#### Physical Object

An object capable of receiving and sending electromagnetic, collision, and gravitational messages

The Transport of collision, gravitation and electrostatic forces is handled by the physics engine.

(Tricycle with mass, speed, acceleration, friction)

#### Widget

An object located in 3D space that transduces mouse events that arrive at its inputs (control surfaces) to other objects

**Animation Object**

An object that encapsulates behaviors (interactive animation methods)

(Walk, grab, explode)

**Modeling Editor**

A 3D view for editing an object's attributes such as shape, materials, physics, or behavior; the structure of an object can also be edited in 3D.



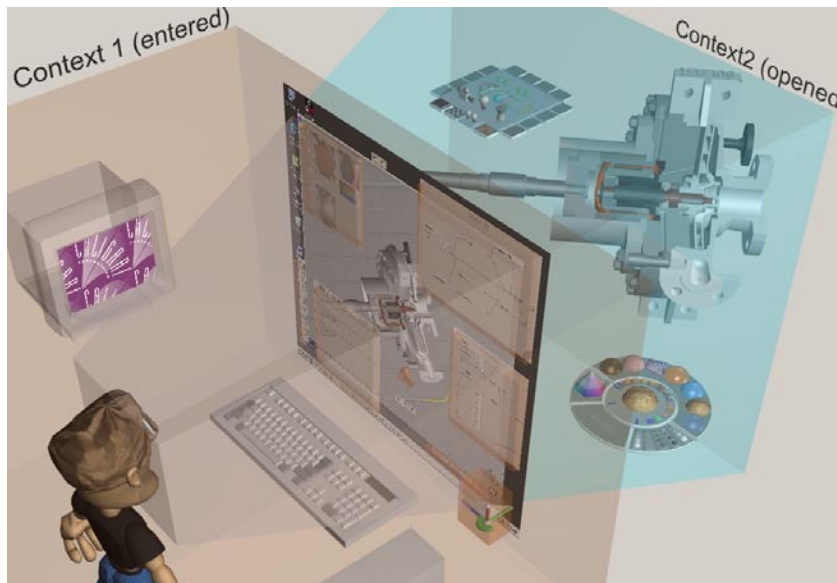
# Chapter 2

## Widgets

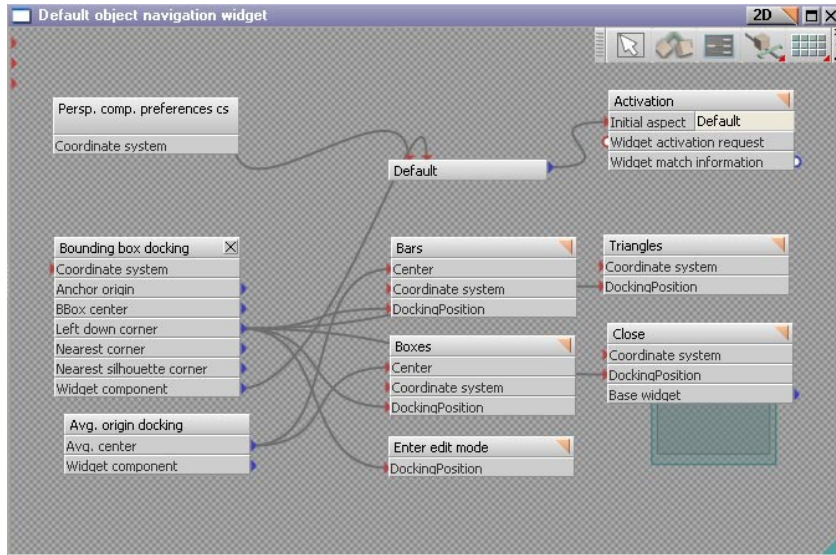
### 2.1 Low Level Assembly

#### Widget prototypes

Prototypes for active widgets are stored by default in the **widget subsystem node** in [Desktop]/Widgets encapsulator. This node contains all automatically managed widgets scanned during automatic activation. (For details, see Widget activation section). The following image shows the Widgets subsystem node with three widget prototypes: **Nav Widget** for object navigation and **ViewNav Widget** for camera navigation and **Spot light widget** for spot lights navigation. It also contains widgets for modeling and NAV toolbar widgets and other nodes for widgets runtime management.



Every widget prototype consists of widget management nodes ( **Widget aspect**, **Automatic activation node**, coordinate systems nodes , docking position nodes), and a set of nodes adding functionality to a widget, forming Base widgets (visualization controllers, widget action controllers , **Gesture node**, and actions). In following image, these additional nodes are encapsulated together for clearer widget design. For more detailed description of all nodes, refer to the end of this chapter.

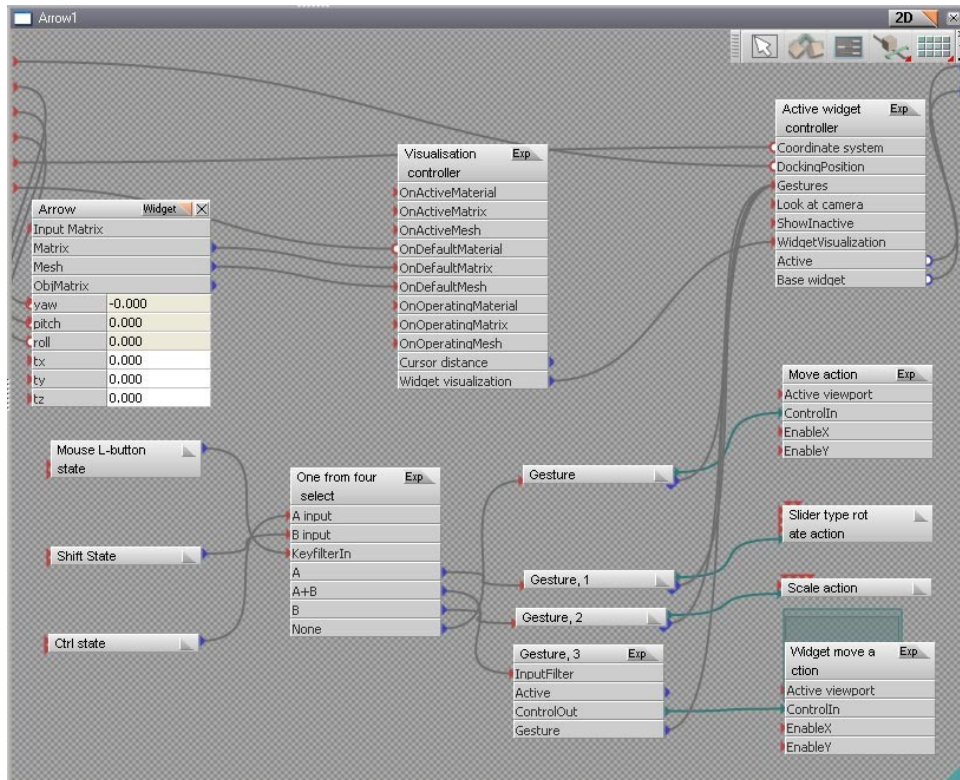


## Simple widgets

Widget actions are defined by base widgets. These can or need not to have control surface. A Base widget with control surface can be split into three logical units: *shape*, *gesture* and *action*. Base widget without control surface is missing the *Shape* part. *Shape* defines what the base widget looks like, for example, an arrow. *Shape* also defines the area where it can be dragged or pressed. Shape-less base widgets are activated by clicking on the screen. *Gesture* defines the combination of keyboard keys and mouse buttons for *action* and *actions* define the operation that will be performed when the user activates the base widget. *Action* can be either a command or a widget tool. Command base widgets are treated as buttons (can for example Select) with the command being executed after button release. Widget tools perform some continuous action – for example moving current selection. If the base widget becomes active then the widget tool will receive all user inputs.

Because it is possible to link more actions to one shape (for example LDrag will activate Move, RDrag Rotate), it is suitable to group base widgets according their control surfaces (shapes).

Here is an example of the “Arrow” base widget. It contains four actions: *Widget move action*, *Slider type rotate action*, *Scale action* and *Move action*.

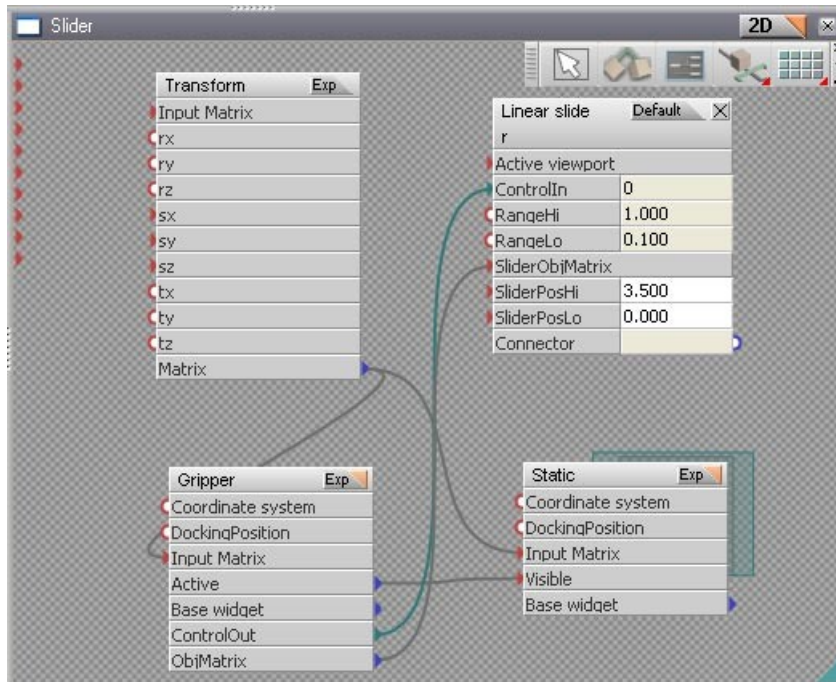


The Arrow node defines shape and consists of shape and Euler transformation nodes. The *Visualization controller* handles the selection of the current shape according widget state. The *Active widget controller* handles base widget management (matrices setup, tool messages dispatching and gestures management).

*Gesture* handles tools activation and deactivation according the user defined activation conditions. In this example the base widget is an arrow that when LClicked will perform moving, LClick+SHIFT will perform scaling, LClick+CTRL will perform rotation, LClick+CTRL+SHIFT will move widget. The coordinate system, in which base widget operates, is defined by shape yaw, pitch and roll Euler angles input.

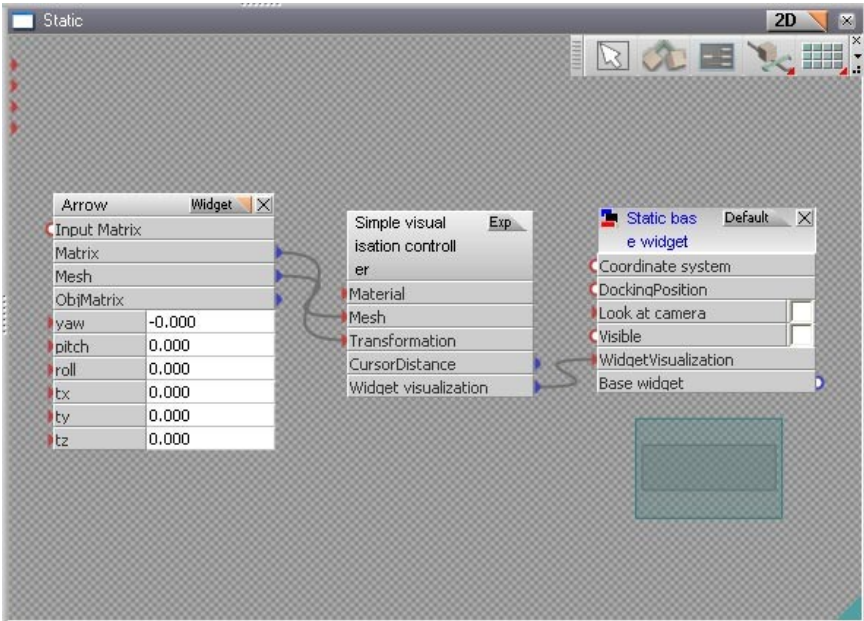
## Widgets with decorations

Not all widget surfaces need to be active. An example for widget with some inactive shapes may be a slider control.

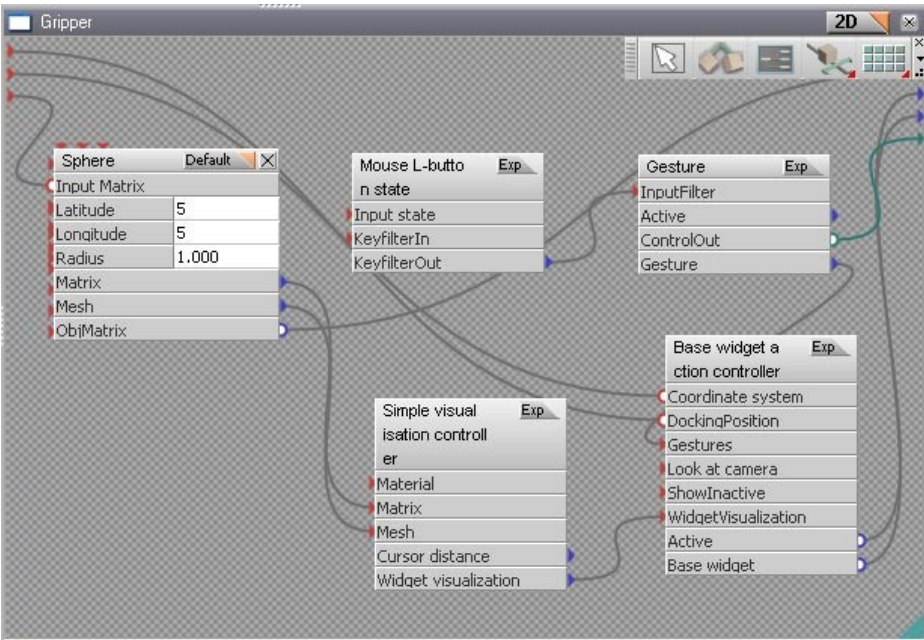


The Slider control contains a Static part (the one without linked actions and a gripper) that can be dragged and moved. The Active part is connected to *Linear slider* node, which is widget tool.

The Static part consists of a shape, visualization controller, and a *Static base widget* node that is a simplified version of *Active base widget*.

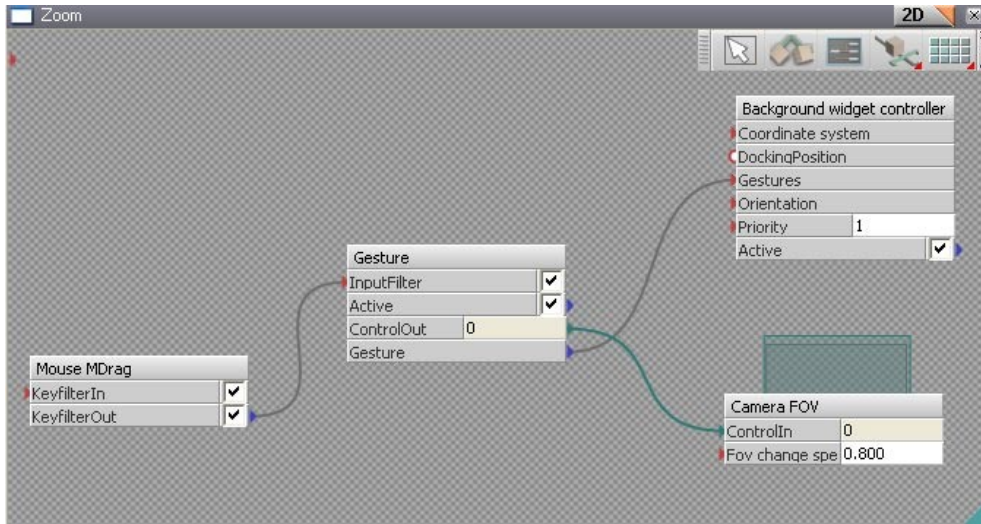


The Active part resembles simple widgets. Note, however, that they export *Active* connector, so the static part during activation knows if base widget active control is active or not.



## Background widgets

Widget subsystem supports also another widget controller node: Background widget controller. It can be used to define base widget without a control shape. These base widgets use whole screen as control surface and thus behave like tS toolbar tools. Using this base widget, users are able to customize behavior of Rosetta toolbar tools using the same interface and nodes that are used for 3D widgets. Here is an example of Eye camera zoom, which can be activated by MDrag (dragging mouse with middle button pressed) on screen.



Orientation for action is for background widgets determined by **Orientation** input connector. Gesture priority within active tool stack is defined by **Priority** connector. Background widgets are fully compatible with existing mouse tools and can also extend functionality of ordinary widgets with control surfaces.

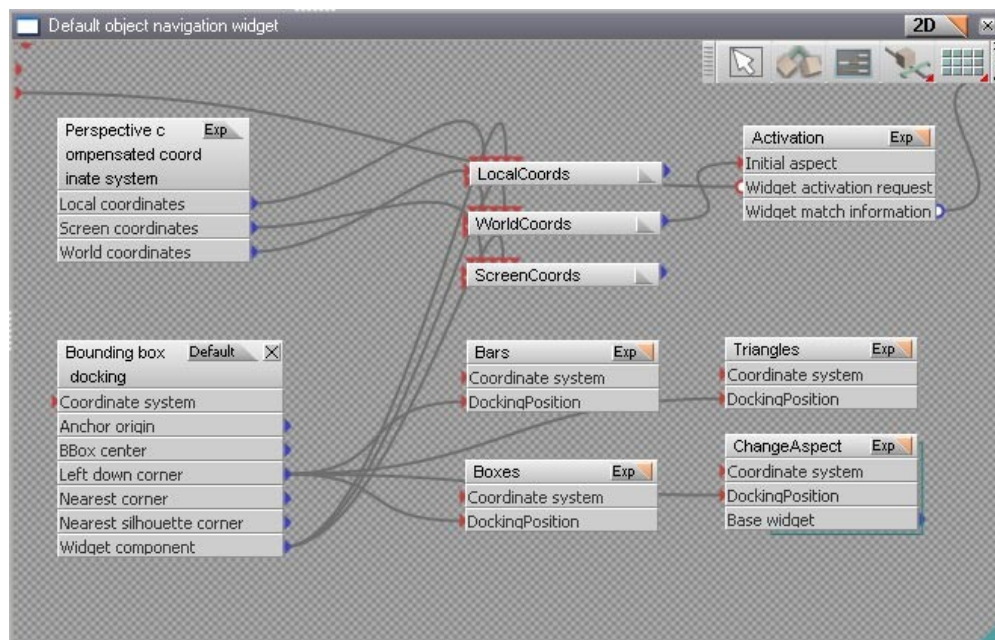
## Designing widgets

As mentioned before, you can create either a widget template or directly active widget. To design a whole widget template, you need to understand the widget activation process so that the newly created widget could be used by widgets subsystem. First, let's start with much simpler task: creating new aspects and base widgets. Then, we will go through design of a widget template. Finally, we will create a simple widget.

### Designing new aspects

Widget aspects are specified by **Widget aspect nodes**. To create a new aspect, add this node to the Widget prototype, specify its name, and then connect the requested coordinate system and base widgets (via docking nodes) to it. An example of a multi-aspect widget is the NavWidget. It contains 3 aspects: local, world and screen. Similarly as in this node, each node can contain multiple aspects. The Widgets package

provides actions to switch between multiple aspects. To add this control, drag and drop the ChangeAspect base widget from widgets library, and connect it to desired docking position.



## Designing new base widgets

To create a new base widget, you can drag and drop an existing base widget from a widgets library and modify it. You can change shapes or actions.

When changing gestures you need to be careful in designing activation conditions. You set gesture by connecting activation filters to gesture controller. When setting more complex gestures on multiple-action surfaces you have to take care that both actions will not be activated at once (if this is not intended). For example, when you specify LClick and CTRL+LClick on the same shape, then pressing CTRL+LClick will also activate the LClick action, and both actions will become active.

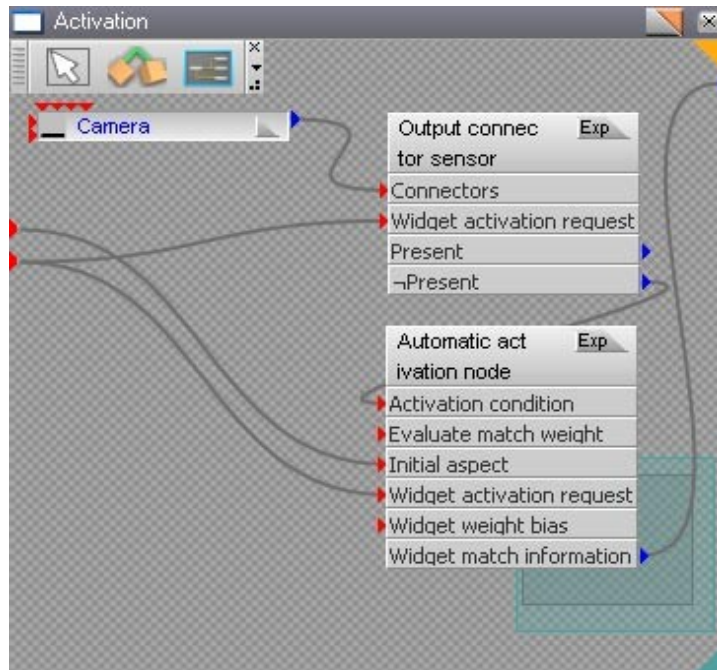
The Widgets library currently provides only limited number of different actions, whose descriptions are available at the end of this chapter. However, you are free to create your own widget tools using scripts or by creating new Rosetta packages.

## Widget activation

For widgets to become functional, they have to become active. The activation process was selected in order to simplify widget design and to reduce tons of links. There are two forms of widget activation.

### 1: Automatic activation

The widget is activated by node selection (by LClick on it in LE, or by the Pick tool). The widget subsystem deactivates the widget for last selected object. Then it searches all available widgets and activates the widget that is most appropriate for the selected node. This appropriateness is determined by **Automatic Activation Node**. Suitability is detected by activation condition, which is built with connector scanner nodes and logical operators. If the activation condition is true for the widget and selected node, the widget is scanned for all actions. Widget suitability is determined according number of active actions. This further suitability evaluation can be disabled for performance purposes by un-checking **Evaluate match weight** checkbox. Widget activator searches for the widget with highest probability of appropriateness and creates an active widget from it.



This example shows the usage of an Output connector sensor and Automatic activation node. The widget contained in this node will only be activated if the target node does not contain a Camera connector.

### 2: Manual activation

A widget is activated by the activation command or from a script. Command parameters directly specify widget prototype to activate, the anchor node, and the controlled objects. This type of widget is not automatically deactivated. Currently, it is used to activate ViewNav and Background widgets. To activate widgets from a script, write

```
Widgets.ActivateWidget('widgetPath', 'Aspect', 'Objects', 'Anchor');
```

to command prompt or to script. The meaning of parameters is as follows:

- *widgetPath* specifies position of widget prototype in dependency graph (link editor)
- *Aspect* is name of initial aspect
- *Objects* is object path, for which the widget is being activated, if the object name is empty, current selection is taken.
- *Anchor* is object path that will serve as anchor for widget

`ActivateWidget` script command activates widget to default unmanaged group. If you wish to activate widget into `ToolWidget` group, use

```
Widgets.ActivateToolWidget('widgetpath', 'Aspect');
```

The meaning of the parameters is similar to above. `ActivateToolWidget` script command is used to activate background tool widgets from View3D toolbar. `ActivateToolWidget` does not contain objects and anchor, because this should be updated in runtime by **Synchronize widget with selection** node. If you create your own background widget, you can use this script command to deactivate old tool from tool group and activate your newly created.

To activate widget into specified group, you can use

```
Widgets.ReplaceWidget('groupId', 'widgetPath', 'Aspect', 'Anchor', 'Objects');
```

where meaning of parameters is similar to above. *groupId* specifies GUID of widget group. Currently, there are three widget groups defined.

- Unmanaged, with ID {1ED6A591-1DF7-4e2c-9333-41216D589C27}
- 3D widgets with ID {5C9008D4-B6B3-4359-9E63-18D2FC228A6E}
- Toolbar tools with ID {88839603-7F98-41f0-96AC-BE94E801CFF3}

You can add as many groups as you wish.

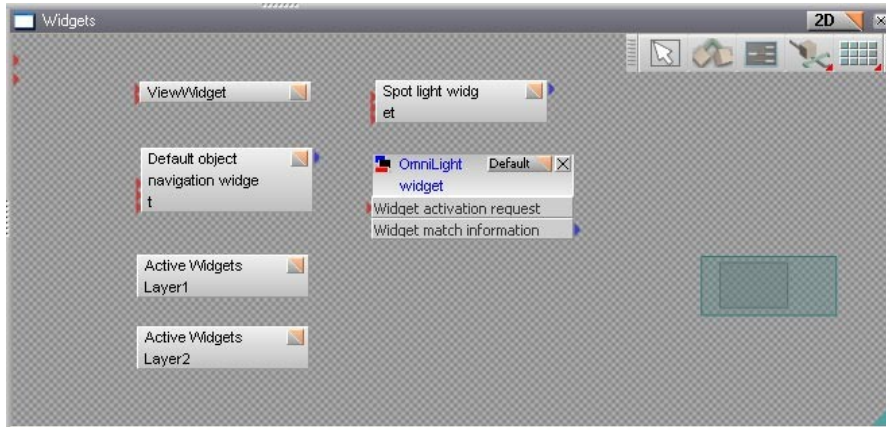
## Designing new widgets

You are free to create new widgets. You can start by copying and modifying existing widgets, or you can create a new widget by adding a `Widget` node with management nodes to the `Widgets` subsystem encapsulator. You can create either automatically activated widgets (when you add `Activation` node with appropriate connections), or manually activatable widgets. Then, you need to write activation commands into script window, or create a tool button with the activation script.

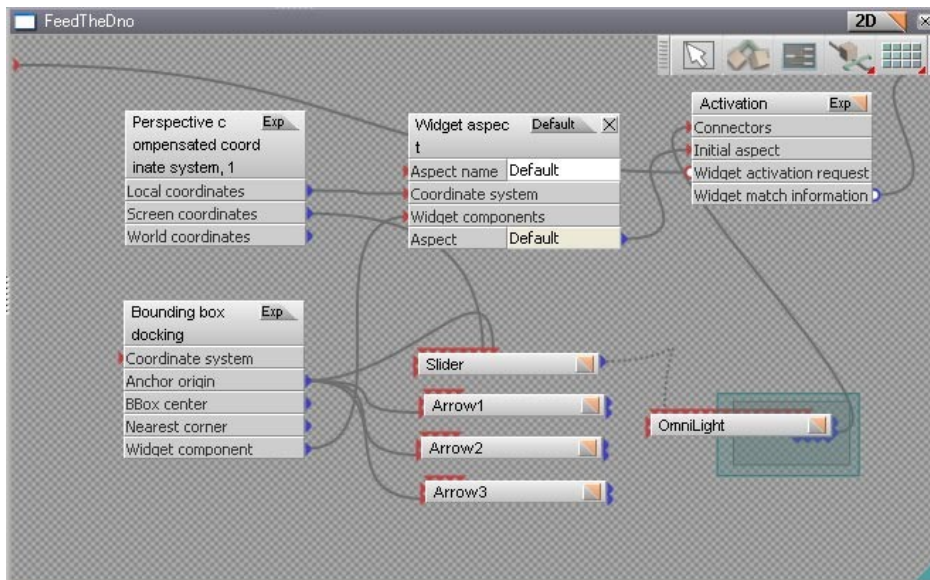
## Feed the dino widget example

In this example, we create a feed the dino widget from an old Omni light widget.

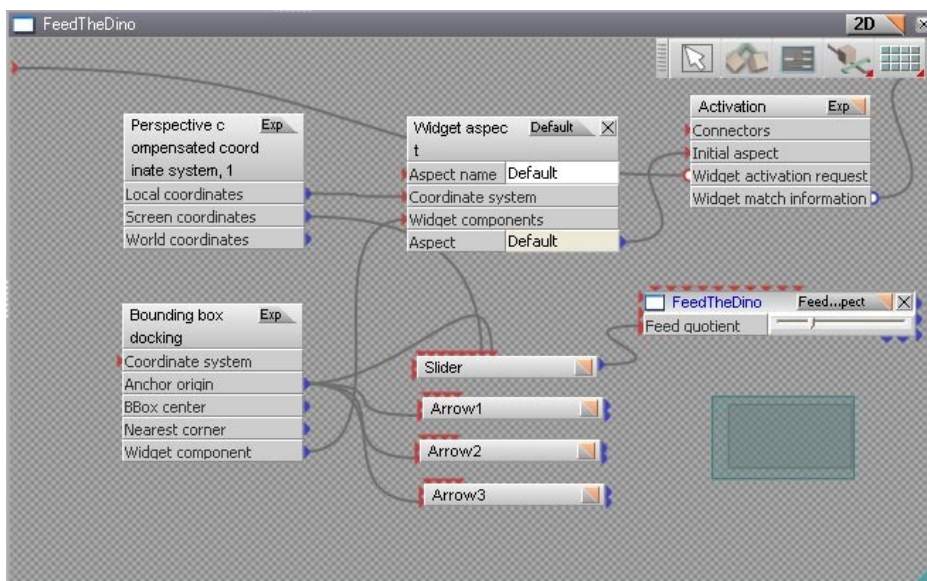
First, locate `OmniLight` widget in widgets library and drop it to `Widgets` subsystem encapsulator. The `Widgets` encapsulator can be found in Desktop.



Rename the OmniLight widget to FeedTheDino, and open it.



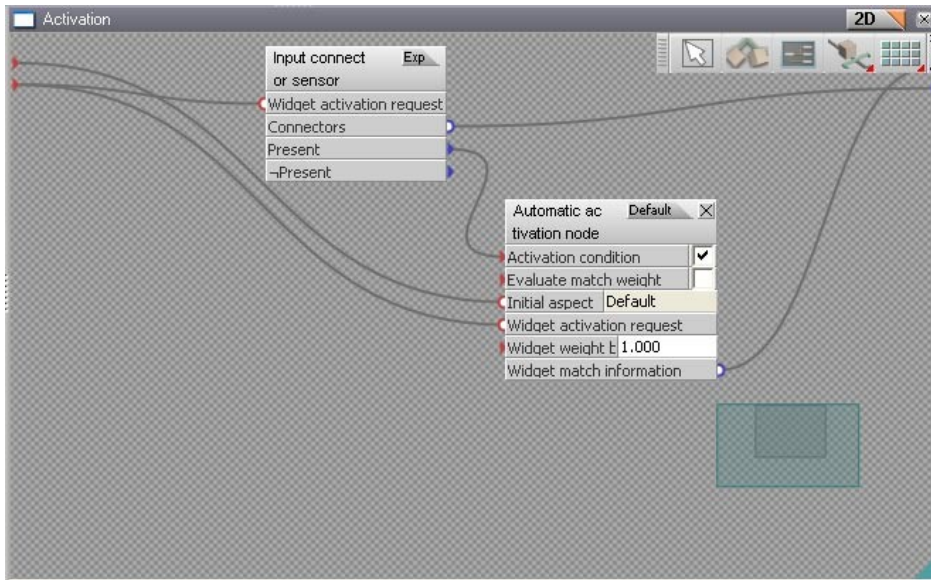
Locate the *OmniLight* node and close it. From scripts objects library, take *FeedTheDino* and drop it to in *FeedTheDino* widget. Link the *Slider Connector* output to *FeedTheDino* quotient.



With this connection, we have provided Slider node with the identification of the connector we wish to control.

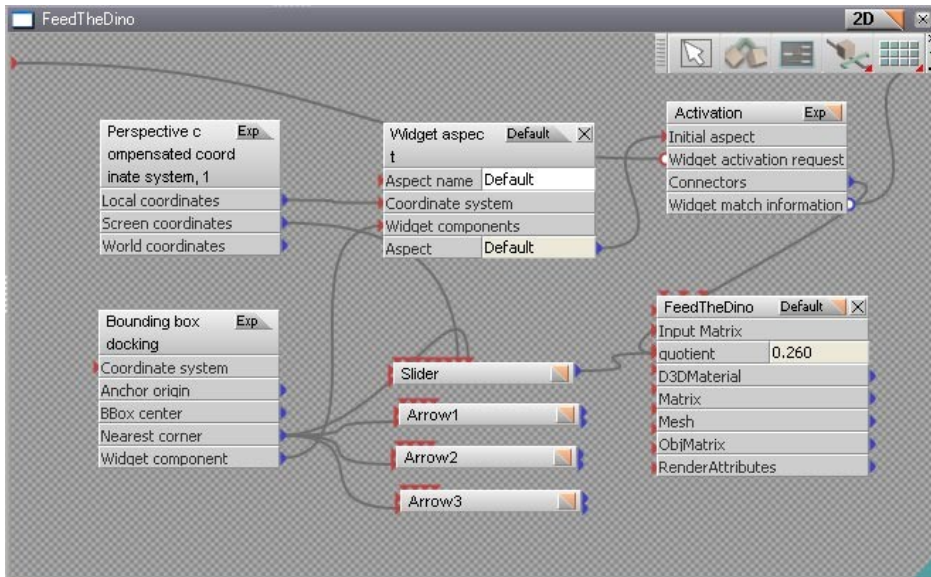
Now, we have to update the Activation node so that the widget subsystem can properly use our new FeedTheDino widget. We will identify the FeedTheDino script object by its quotient input, because if other nodes contain the same Mesh Shrink Filter as dino contains inside, our new widget will operate properly – all its controls will be functional.

First, we have to remove old *Output connector sensor* because it was scanning input from the omni light output connector. We need to put the *Input connector sensor* into activation node and connect it to dino's quotient.



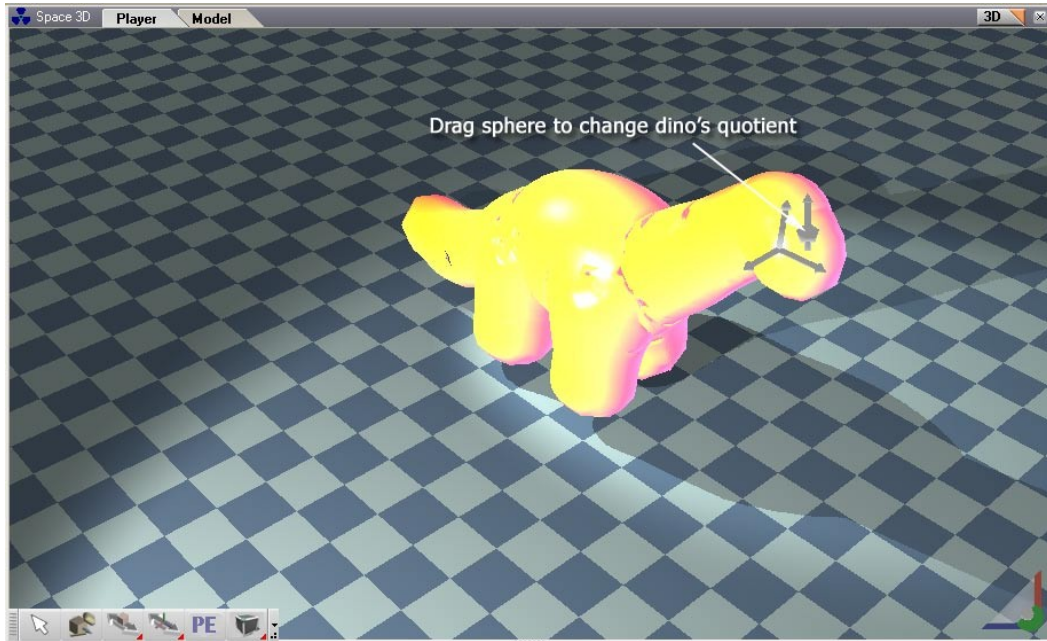
Note, that we have left *Evaluate match weight* disabled, because the quotient connector is genuine and no other object (excluding objects having MeshShrinkFilter inside) can have this connector exported.

Finally, go back to FindTheDino widget and connect *Connectors* from *Activation* node to FeedTheDino's quotient.



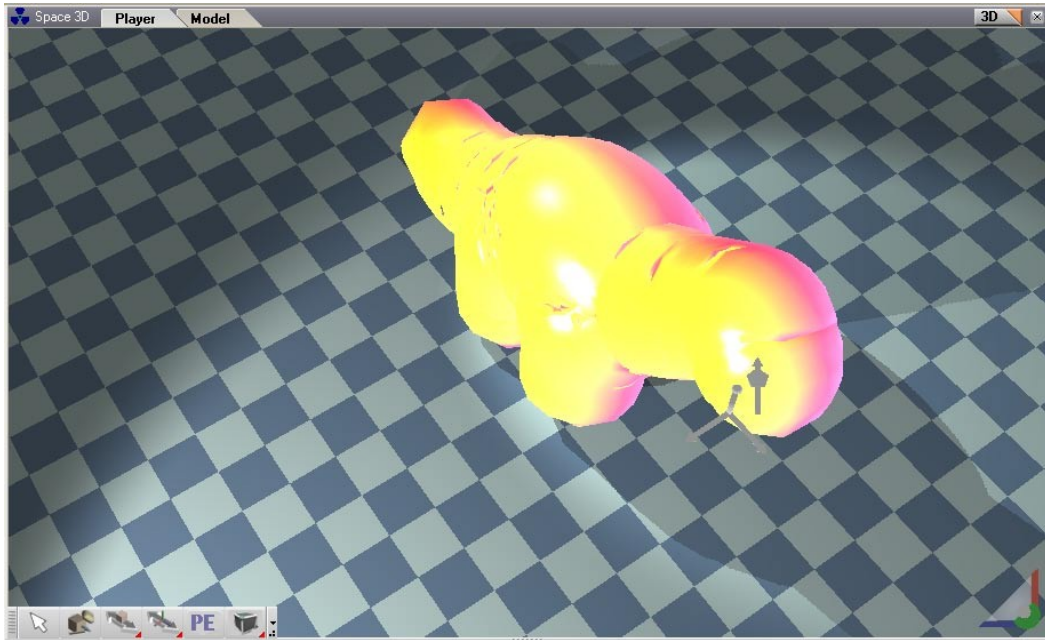
You can also change docking position from Anchor origin to Nearest corner by reconnecting all links from *Bounding box docking* Anchor origin connector to Nearest corner connector. You can specify slider range by RangeLo and RangeHi inputs of Slider node. The good values for FeedTheDino are -1 and 1.

Now, go back to Space3D and Drag&Drop FeedTheDino from the objects library to space 3D. Your new widget should be activated.



By dragging and moving the sphere, you will change quotient connector of a *FeedTheDino* node in space 3D.

If you somehow cannot create new widget, FeedTheDino widget is provided in widgets library, so you can drop it into Widgets encapsulator and study it.



You can add material to your FeedTheDino widget. You can extend your widget with additional actions, for example for rotation, or scaling. Check AdvNav Widget or nodes description for guidance on how to use scaling, moving, and rotating and other nodes.

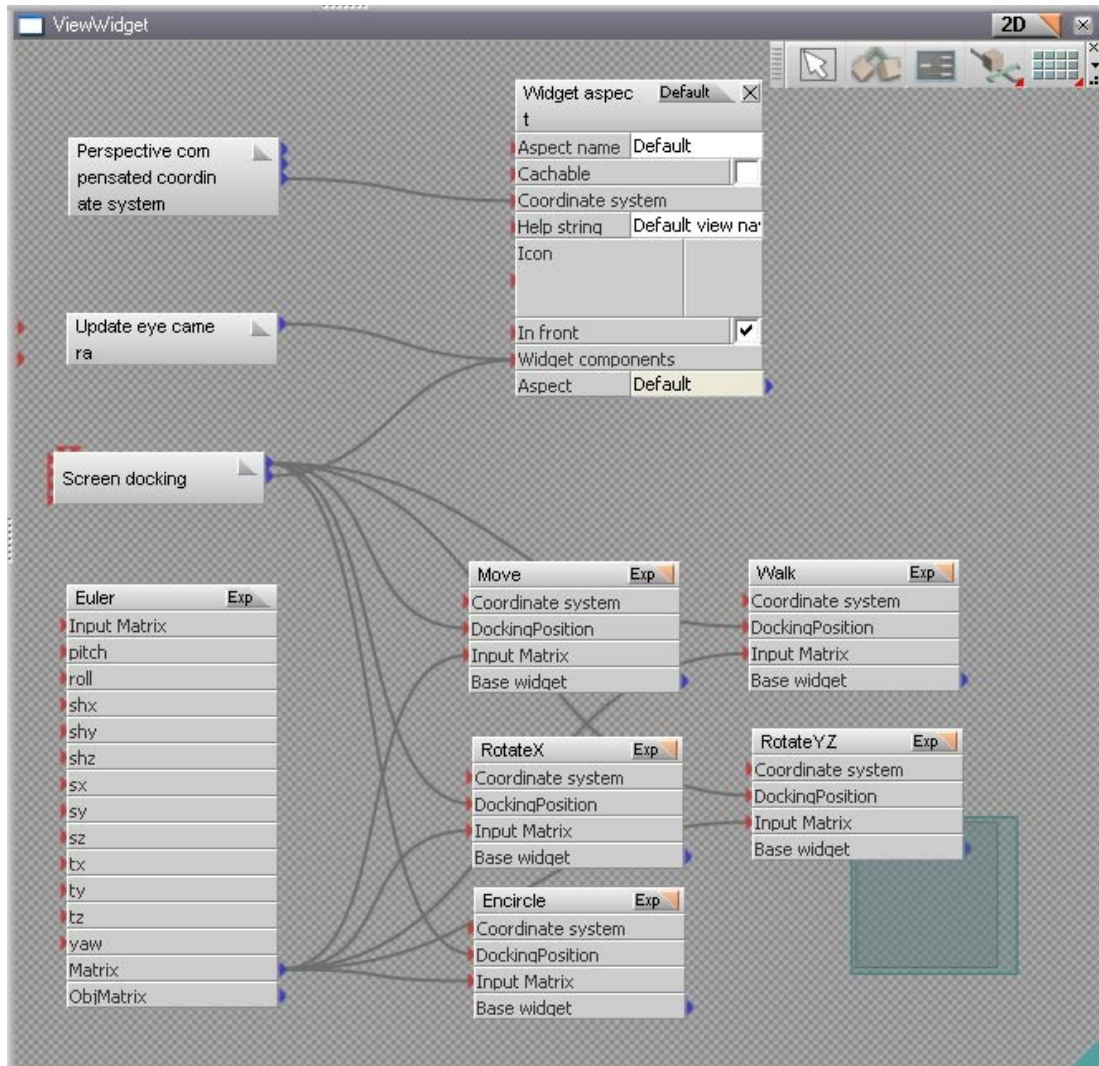
### Changing action activation logic (gestures) example

In this example, we will change the action activation logic by adding new action: Walk to view widget. Walk base widget contains two gestures. LClick moves in XY, and RClick in Z. We will add Camera Walk, which will be activated by SHIFT+LClick.

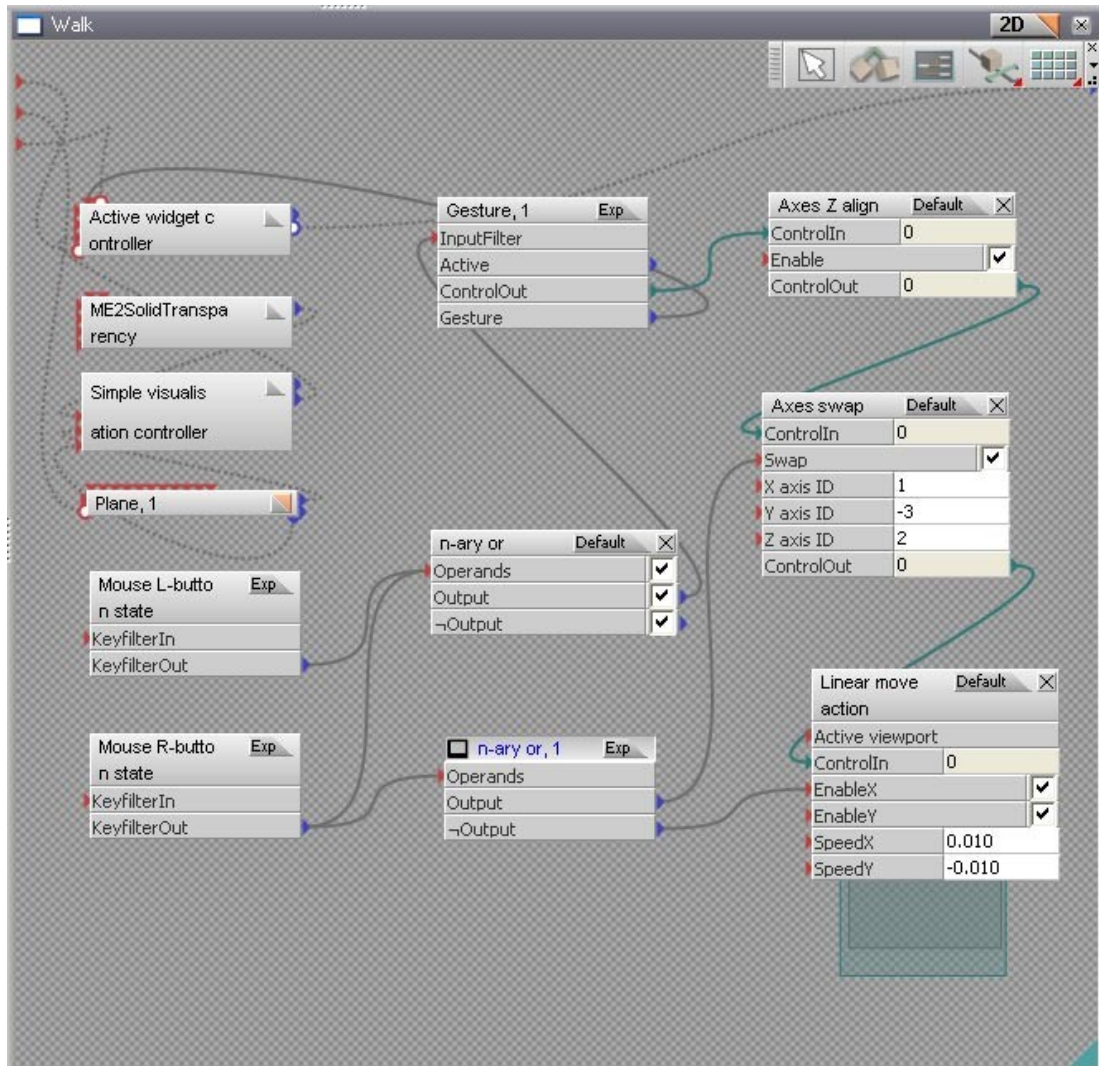
Walk action starts moving the camera forward at the specified speed, while the mouse can be used to look around. This style of movement is well known from computer games.

The View navigation widget is created from a script, and it is not activated automatically, so after we add Walk action, you will have to delete old View navigation widget from the Active widgets Layer2 encapsulator, and use command prompt window to manually activate the new widget.

First, locate ViewWidget in the widgets subsystem node (Desktop/Widgets/ViewWidget), and open it.



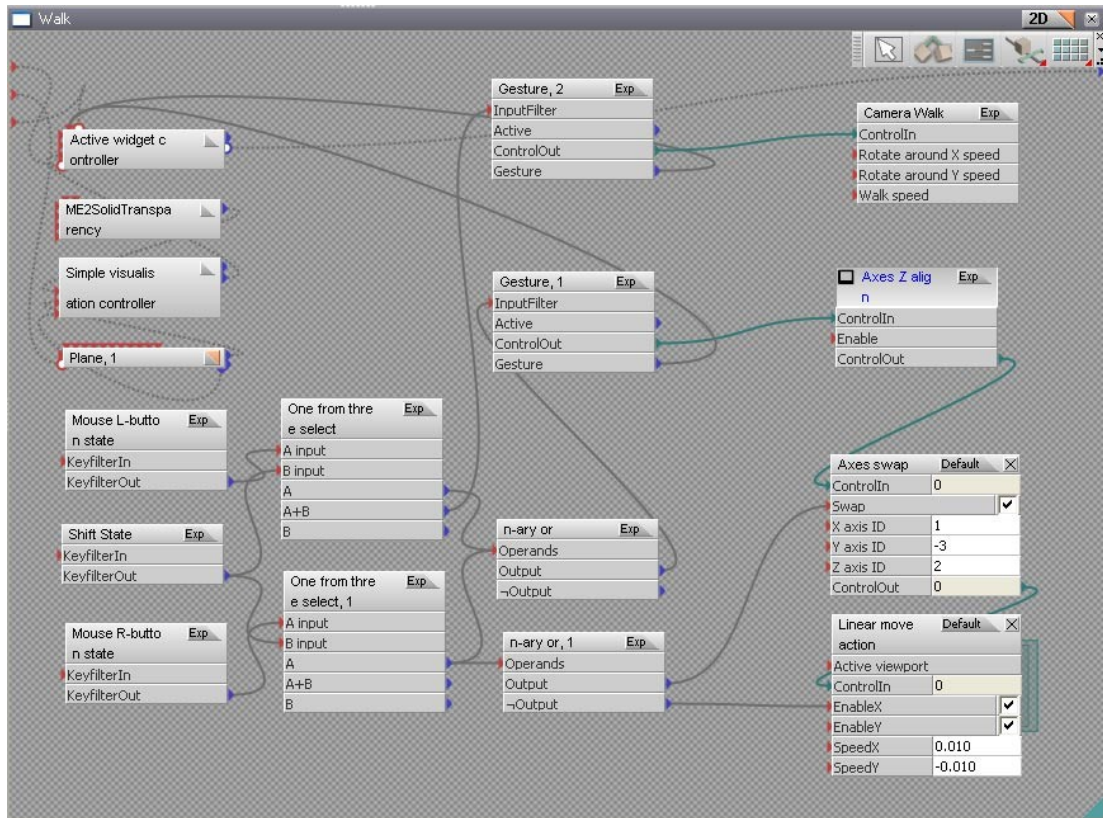
We will modify the Walk base widget, so open it.



As you can see, gesture and action attached to the *Walk* surface is a little complex. It contains one *Linear move* action, but this is modified with Mouse R-button state. When you press mouse L button, linear move action uses Z-aligned (axes are rotated so, that Z points up) camera transformation matrix to move in X and Y coordinates. When you press mouse R button, the *Axes swap* node swaps Y and Z axes, so Move will operate in YZ space, and movement along X (camera Y) space is disabled, so the camera moves only up/down.

We want to activate Walk on SHIFT+LClick, so we need to modify action activation logic. Add *Shift state* and two *One from three* select nodes. Also add one gesture node and a Camera walk node. Reconnect Mouse L-button state to *One from three* select input A, Mouse R-button state to another *One from*

three select input A, and connect Shift to both *One from three* select inputs B. Connect other connectors as in the following image:



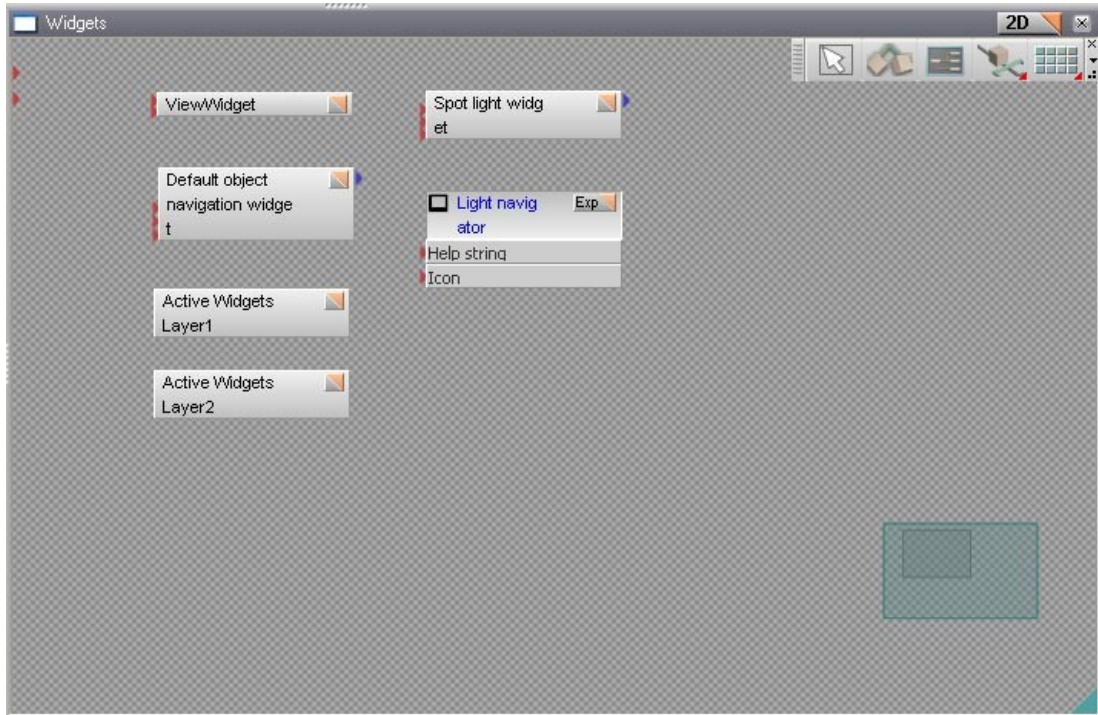
Now, when you press mouse L-button, *One from three select* sets 1 to A output, and only linear move action is activated. When you press R-button, *One from three select, 1* sets 1 to A and everything goes as before. When you hold SHIFT with LClick, *One from three select* sets 1 to A+B output connector and activates Camera Walk. Holding SHIFT key with R-click does nothing, because *One from three select, 1* activates only A+B output, which is not connected.

Because ViewWidget was active during modifications, all changes should reflect to all views. LClicking on the transparent triangle of View Widget should trigger Camera walk action and camera should start moving forward.

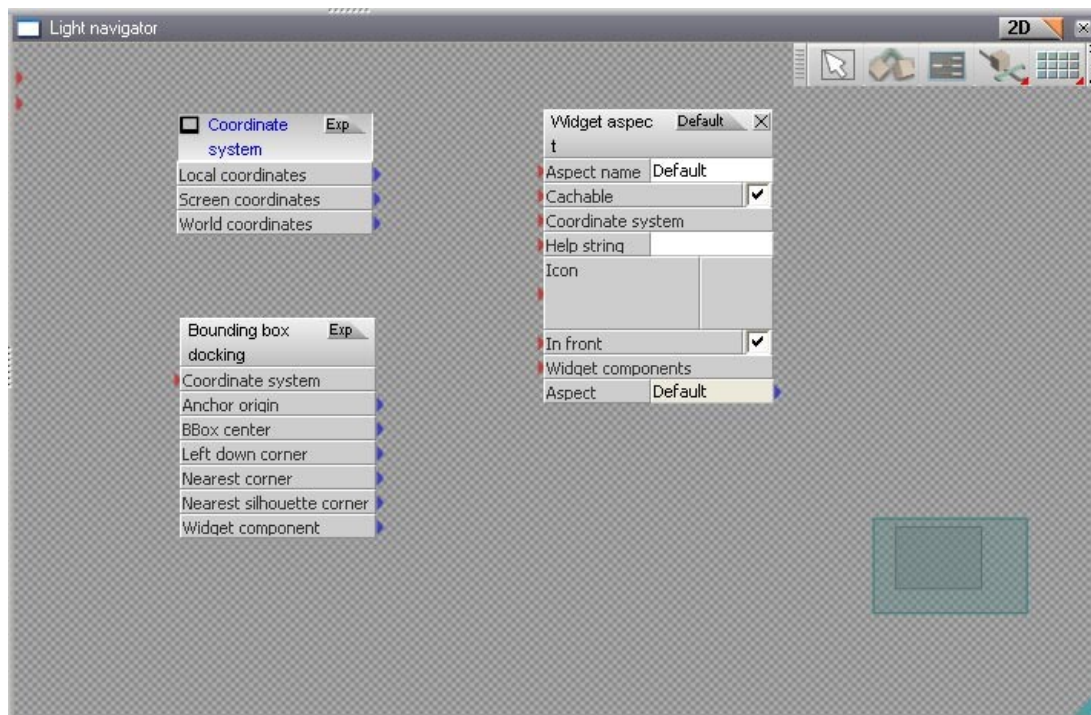
## Simple Light Navigation widget example

In this example, we will create a simple navigation widget step-by-step, which can be used to change the position of lights. All components will be created from scratch, so we can explain widget functionality in depth. Some of created elements can be found in Widgets library.

To create the widget, we need to place the **Widget node** of our new widget into widget subsystem. We can name that node for example Light Navigator.



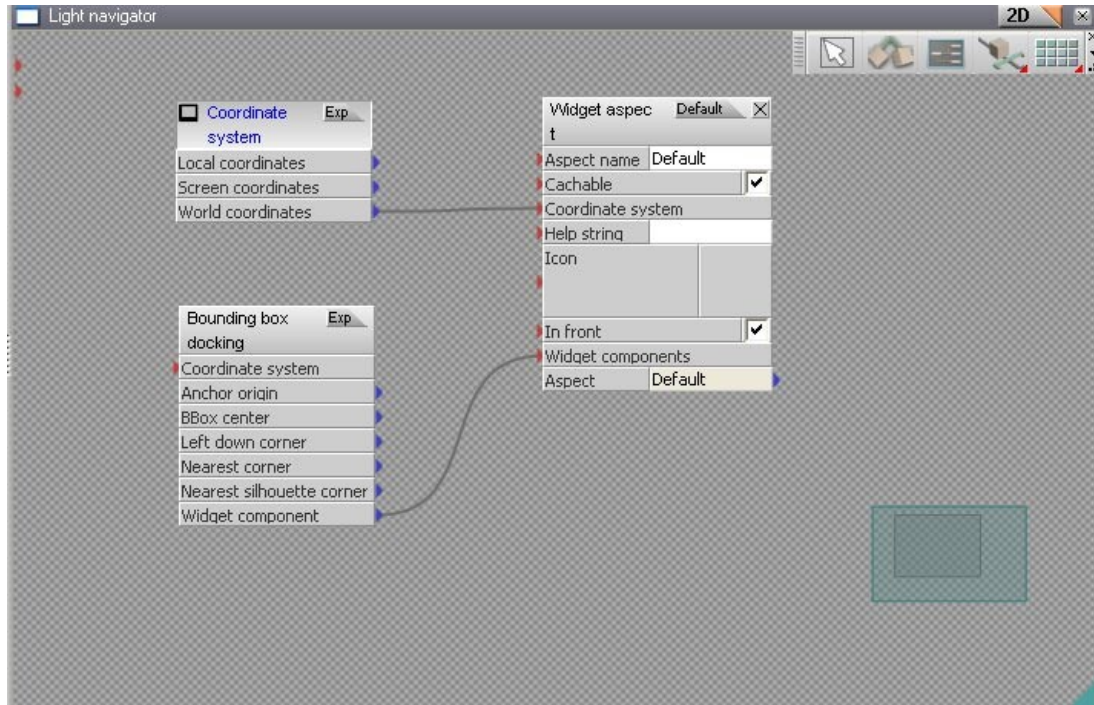
The first thing we need to do for our widget is to add all nodes that are required for widget management and rendering. These nodes are: Widget aspect that specifies widget aspect name and a starting position for widget activation, activation nodes, docking node (**Bounding box docking**) so we will have docking positions to which we can dock base widgets, and a coordinate system ( **Coordinate system**) in which should widget operate. If you want to have your widget's size not dependent on camera settings (position, FOV, etc.), use **Perspective compensated coordinate system**.



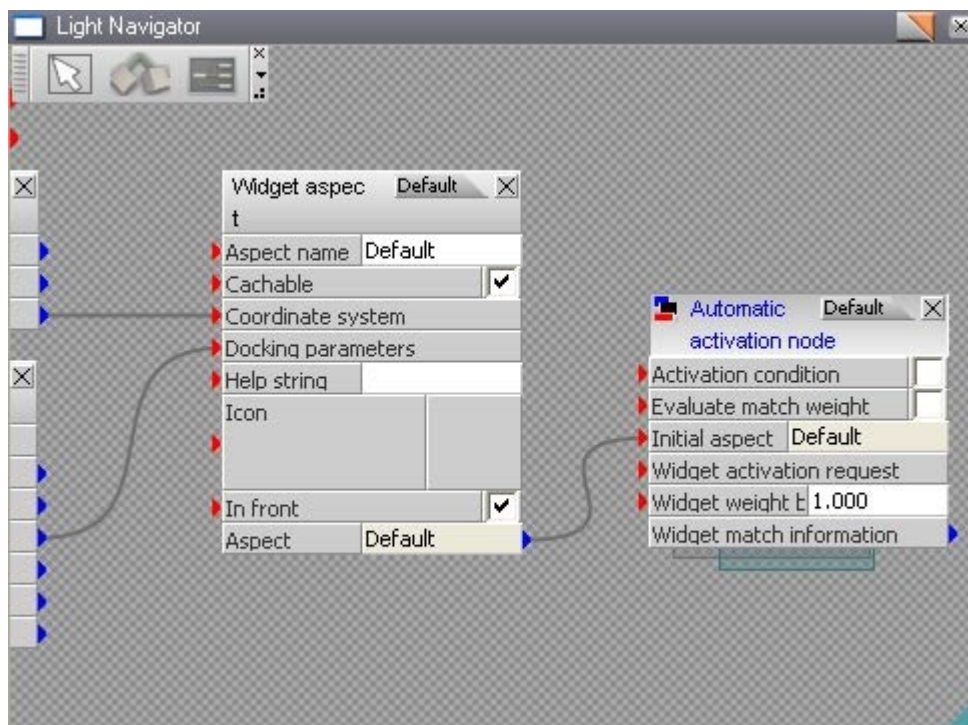
As you can see on widget aspect node, you can specify caching and enable/disable overlapping with scene geometry (with *In front* checkbox). Widget caching takes some memory, but significantly speeds up widget activation and deactivation. Global widget caching strategy can be modified in Widgets encapsulator's panel directly in LE root

The Widget aspect also contains inputs for the coordinate system. This will become the default coordinate system for nodes which do not have their coordinate system connected. In this way, the total number of links is greatly reduced, because almost all nodes will operate on default coordinate system. Widget components input serves as starting position from which activation will spread. Only nodes connected through various links to this connector will be activated in final active widget

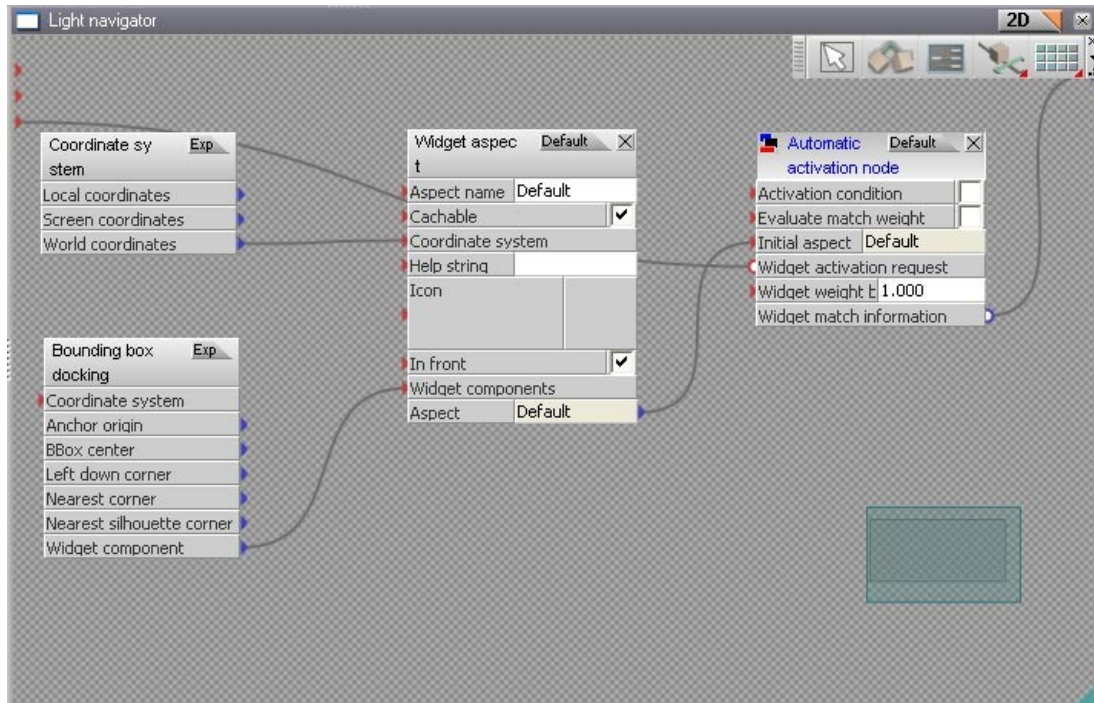
By connecting World coordinates from Widget coordinate system, we say that we want to use world coordinates as default, and by connecting Docking parameters we specify that the default aspect will contain all nodes docked to Bounding box docking.



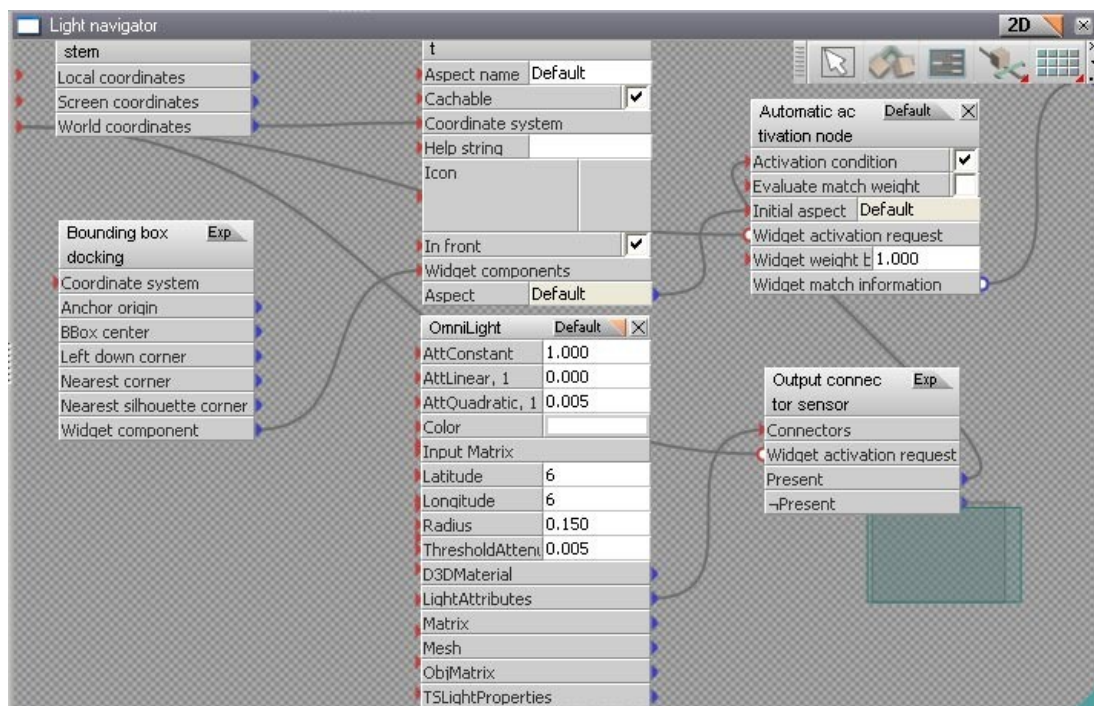
Now, we need to tell the widget subsystem that we want automatic activation of our widget only for lights. This can be performed by adding Automatic activation node to our design.



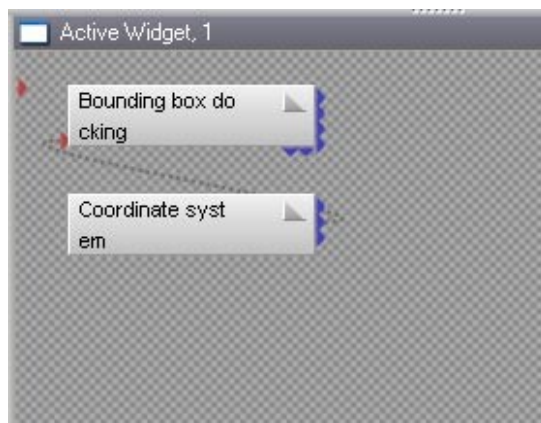
After Rosetta changes selection, the widget subsystem is notified and searches Widgets subsystem nodes for Widget match information. If it finds this connector, the widget can be automatically activated. Widgets subsystem sets Widget activation request and reads back Widget match information. Automatic activation node then scans for activation condition, and if it is true and selected node contains anchor information, widget can be activated. So both Widget activation request and Widget match information need to be exported from our Light Navigator widget. We need to export Widget match information and Widget activation request. Activation process is in described in depth in the Widget activation section.



We now need to specify activation condition such that our widget will be activated only on lights. We can use the Output connector scanner node to achieve this because all lights can be identified by searching for LightAttributes connector. The Output connector sensor scans target nodes for connector presence. Connector is specified by providing connector example - linking **Connectors** connector of Output connector sensor to any connector. We want to activate the widget if this connector is present on target node, so we connect Present connector from Output connector sensor to Automatic activation node's Activation condition. Also, we want to use "Default" aspect after selecting the light, so we need to connect Aspect from Aspect node to Automatic activation node's Initial aspect connector. Note that we also need to connect the Output connector sensor's Widget activation request to outer encapsulator, so sensor node gets information about nodes for which widget is being activated.

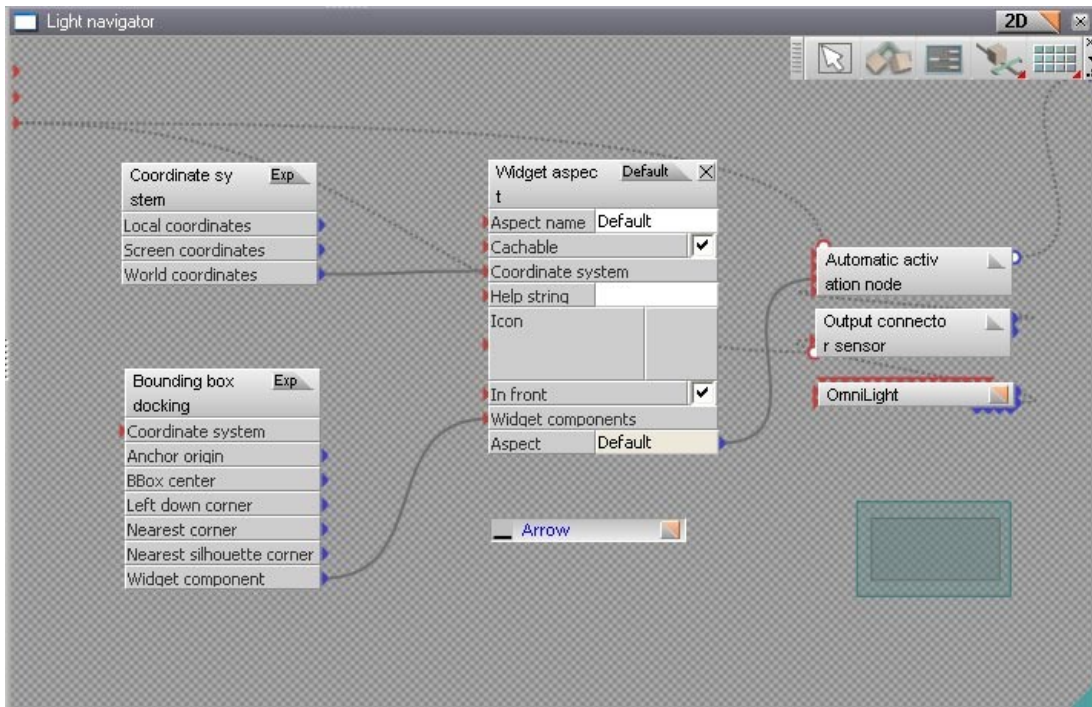


From this point, every time you select a light, Light navigator should be activated. This can be verified by selecting light and checking [Desktop]/Widgets/ Active Widgets Layer2/Active Widget,1. As our widget contains only two active nodes (coordinate system and docking node), active widget will contain only two nodes.

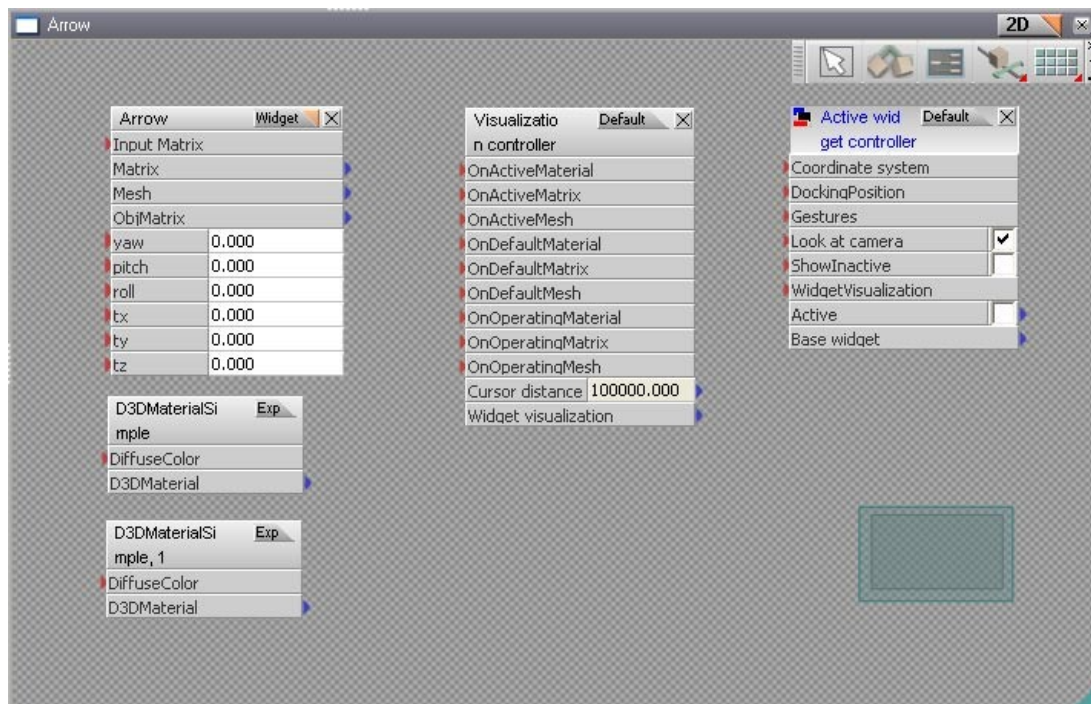


Now, we start to add functionality to our widget. First, let's create a move arrow. After we do this, we will copy it three times to get three movement axes.

To make widget design clearer, we will use encapsulation. So, first place Object from System/Kernel package into the widget and rename it Arrow.

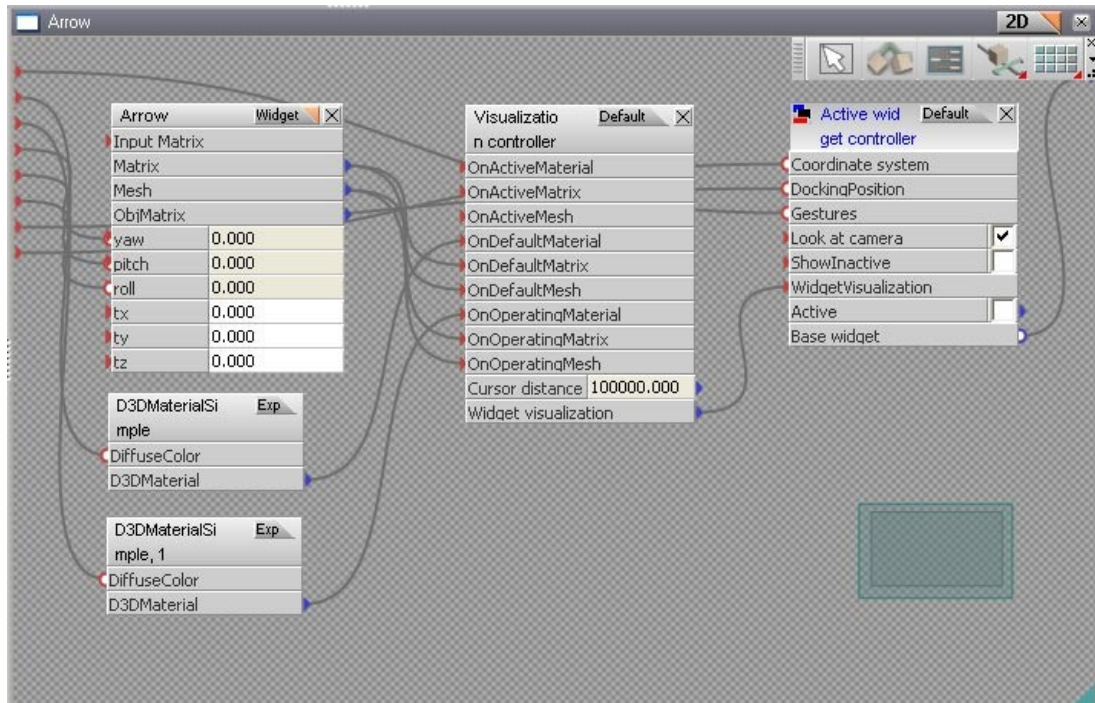


As you have read before, every base widget consists of shape, gesture and action. We add Mesh, Visualization controller, D3D material and Active widget controller nodes into the Arrow encapsulator. As a mesh, you can use a shape you have modeled in tS, but for this example, we will use the Arrow from ui/widgetslib library. When you want to edit your own shape, remember that it needs to be oriented in +X direction, since widget actions operate on the shape's transformation matrix. We want to specify different materials for different widget states (dragged arrow will turn yellow, other surfaces will hide). This is allowed by Visualization controller node. Simple Visualization controller uses the same shape, material and matrix for all base widget states. For a material we use D3DMaterial Simple from DX/D3DView Library. We also put Active widget controller into Arrow node. This way we tell the D3D renderer that it should not render shapes inside, but it should treat the node as a widget.

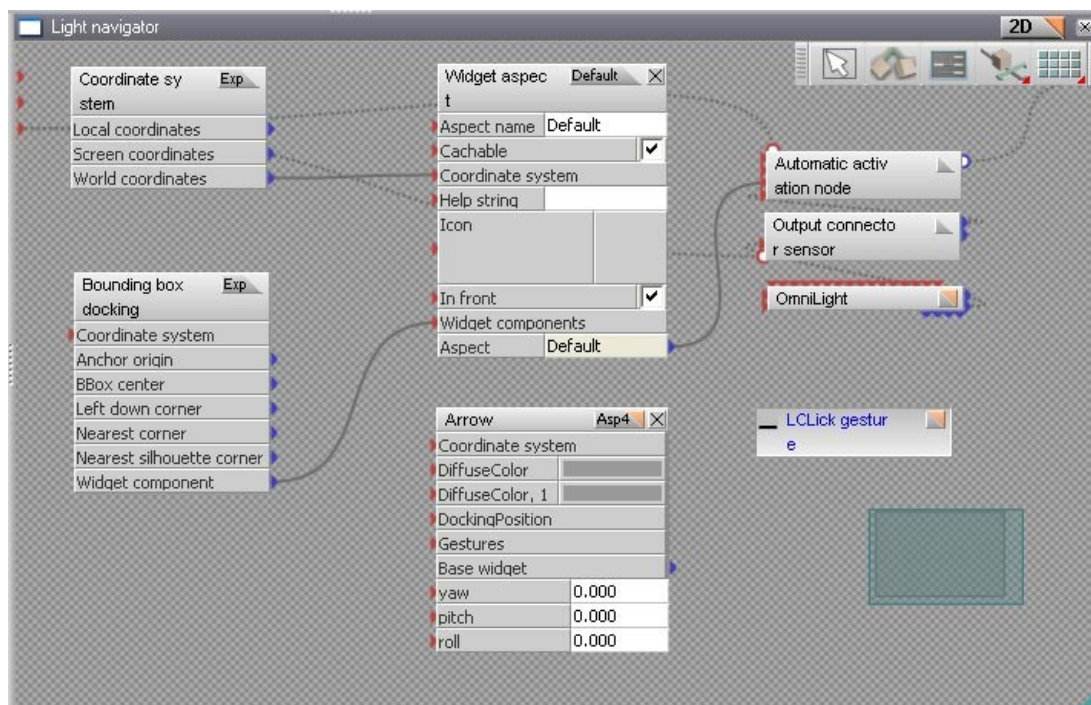


First, we need to export Coordinate system, Docking position, Base widget and Gestures from Base widget action controller. Then, since we want to specify axes from outside of the node, we can export yaw, pitch and roll from Arrow. Also export DiffuseColor from both D3D material nodes.

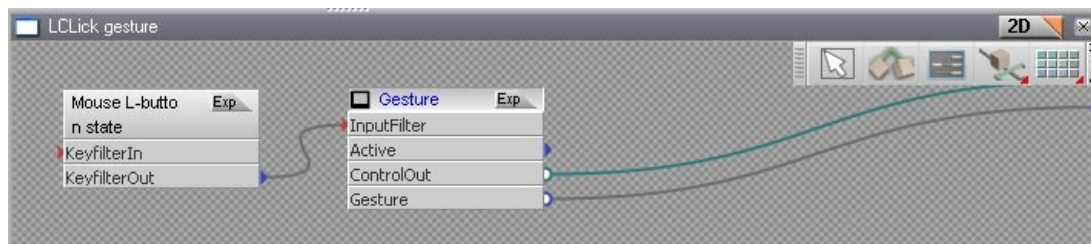
Now we need to connect connectors to create the widget arrow. So, connect Widget visualization from Visualization controller to WidgetVisualization input of Active widget controller. Connect Arrow's mesh to Visualization controller's OnDefaultMesh and OnOperatingMesh. Connect Matrix to OnDefaultTransform and to OnOperatingTransform. Also, connect D3DMaterial from D3DMaterialSimple to OnDefaultMaterial, and D3DMaterial from D3DMaterialSimple, 1 to OnOperatingMaterial.



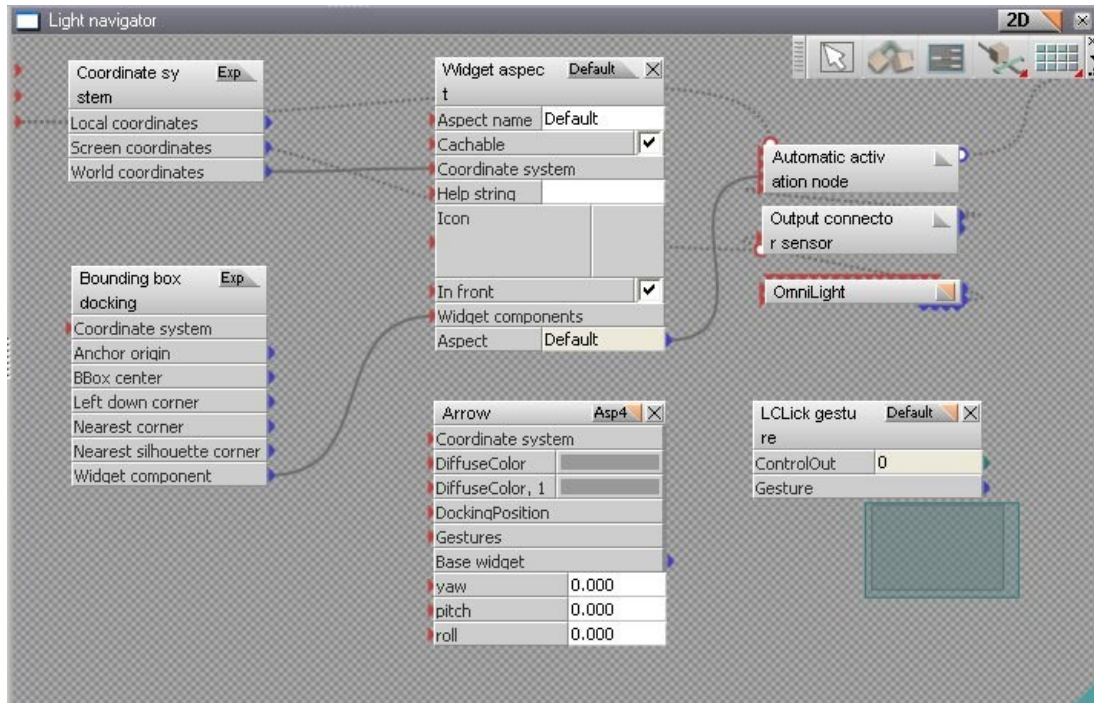
We have just created active arrow. Somehow Rosetta ignores exported yaw, pitch, roll connectors. We need to enter panel editor mode and add these connectors manually. We now need to create gesture and add action to it. Place Object from System/Kernel library to Light navigator widget and rename it to “LClick gesture”.



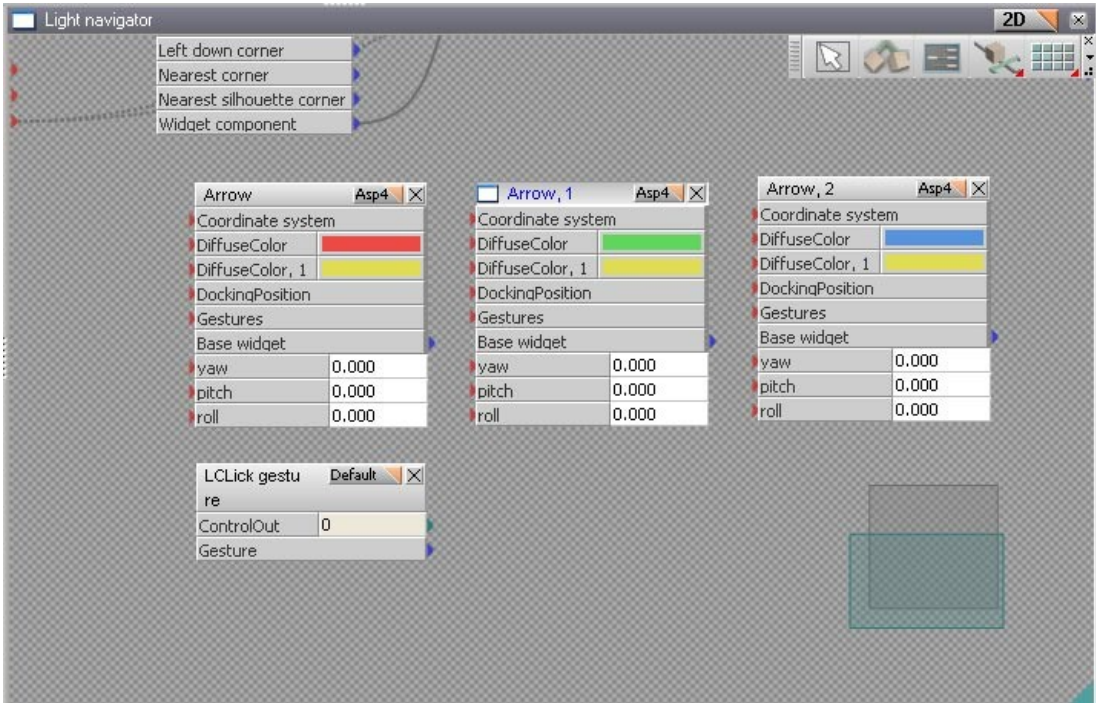
Open it, and place Gesture and Mouse L-button state nodes. Switch nodes to default aspect.



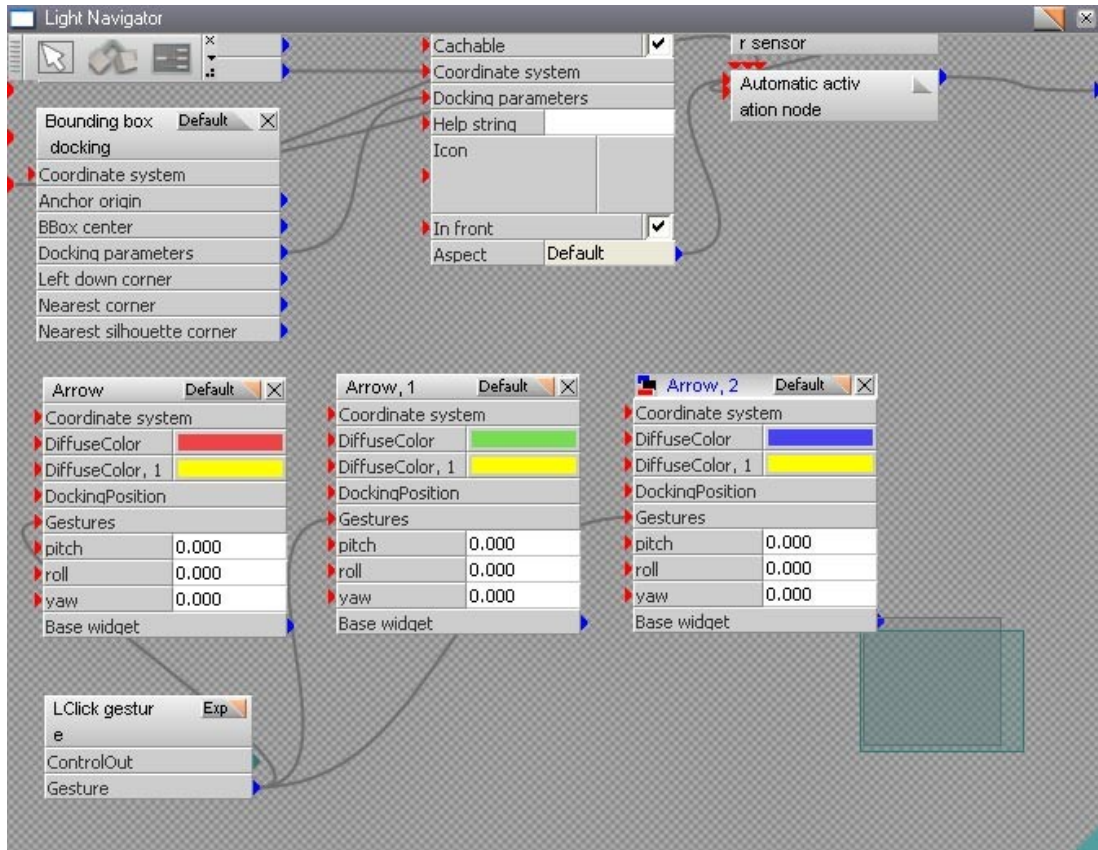
Gestures are specified by Gesture controller and various key/button filters. By connecting KeyfilterOut to InputFilter of Gesture, we limit activation of widget tools (actions) when left mouse button is not pressed. After the participant clicks with left mouse button on base widget's arrow, gesture fires action or activity connected to ControlOut. We want to specify actions outside, so we export this connector out. Also we need to export Gesture, so we can connect it to arrow's gesture input.



We have added Arrow and LClick gesture. We now need to add action. But before we do this, we create other two axes, so we will be able to move a light in all three directions. We can also change colors for default and operating states.

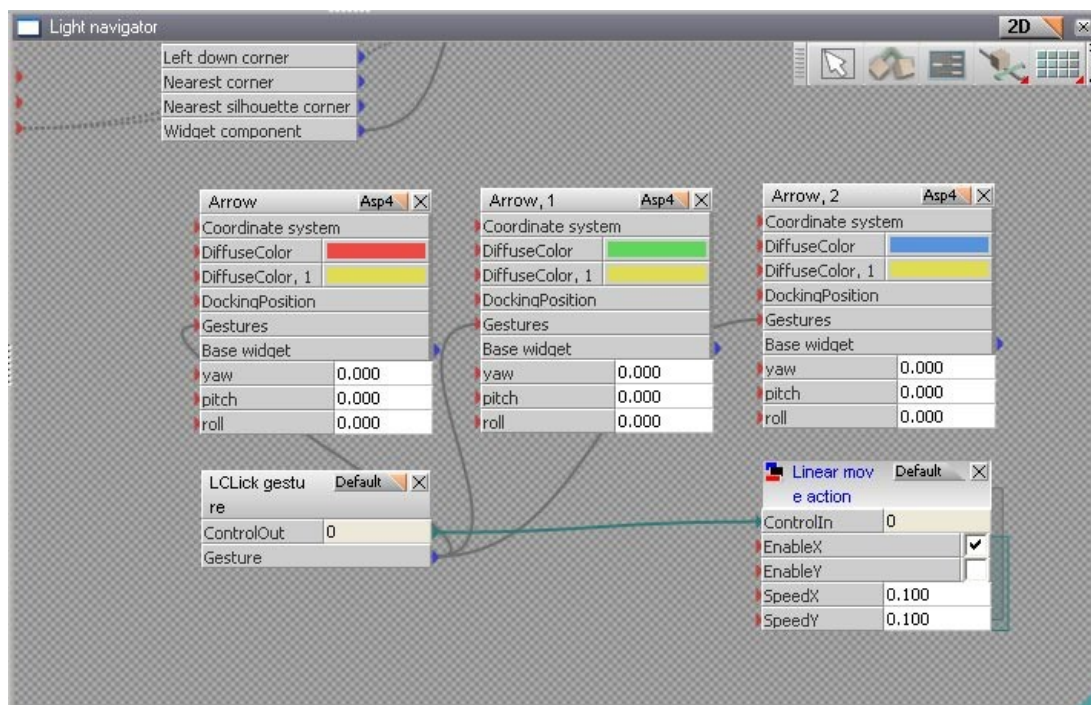


Now, we connect Gesture from LClick gesture to Arrow’s gestures inputs. We need not copy LClick gesture three times, because geometry information is transferred through connectors.



Finally, let's add move action. There are two types of move action. First, there is the **Slider type move action**, which uses mouse step and widget orientation to determine step in 3D. Second is **Move action**, which uses widget's shape XY or XZ plane and computes movements on this plane in 3D such that controlled object is glued to cursor.

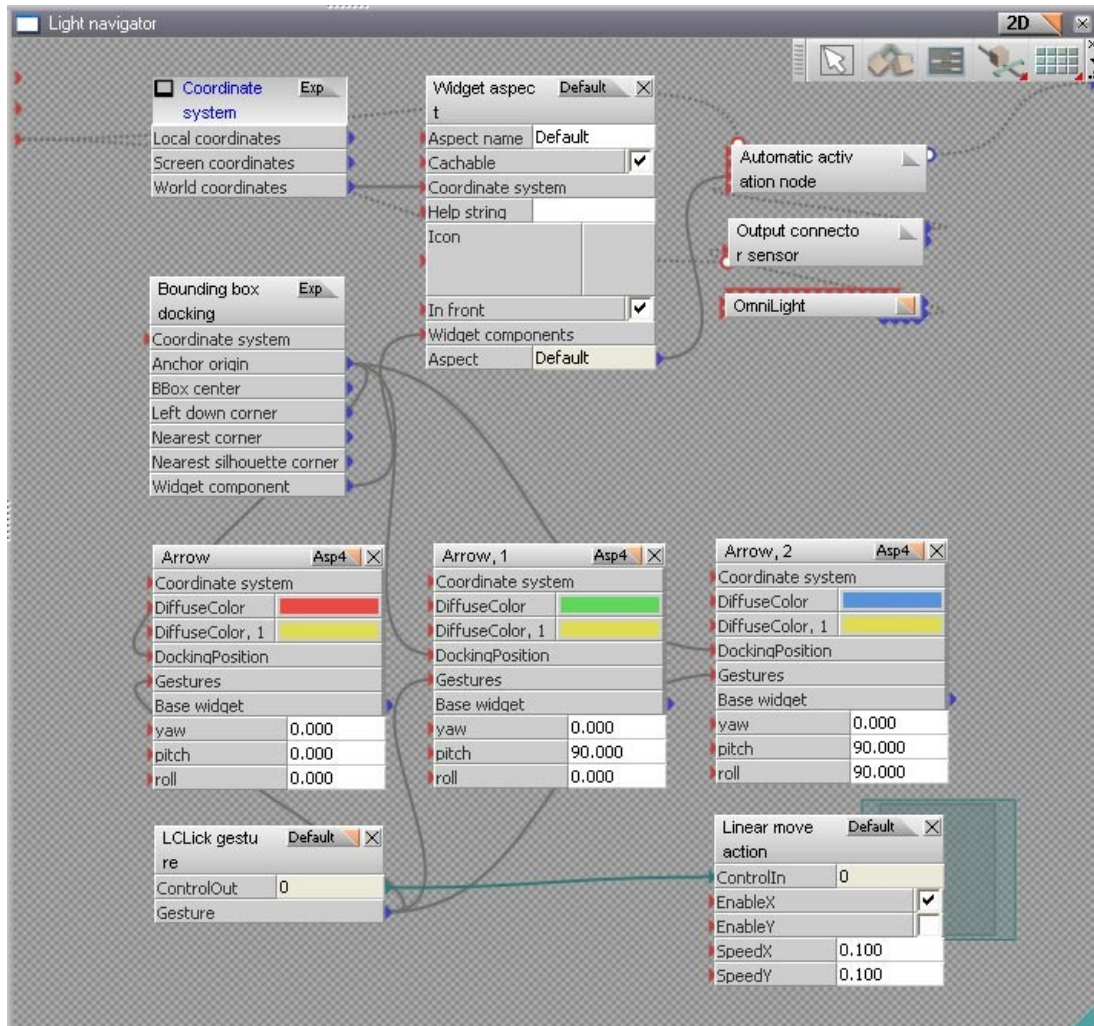
You can check both move actions and select the one you prefer. In this example, we use **Move action**. We connect ControlOut of LClick gesture to ControlIn of Move action.



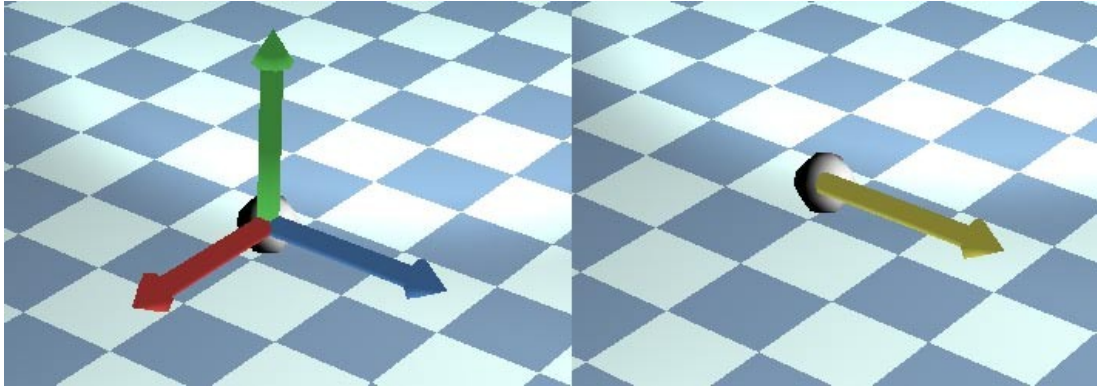
As you can see, Move action node has EnableX and EnableY checkboxes. These lock movements to the selected axes. Move action uses the shape's local matrix (Euler transform stored in Arrow/Arrow) and moves objects in Arrow's X direction. By enabling X and Y directions, move will work in 2D. (Try it!).

To finish our simple light navigation widget, we need to connect arrows to docking positions and change arrows orientation (by setting yaw, pitch and roll) to create orthogonal axes. Because light has sphere as its shape, we can simply dock our axes to Anchor origin.

Here is LE screenshot of a complete Light Navigator widget:



And here is Light navigator widget in action:

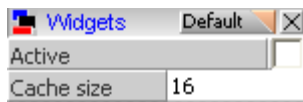


You can store this widget in a library the same way as you store shapes or materials. When someone gives you a library with widgets, you use them by dropping a widget into widgets subsystem node.

Similarly to adding arrows, you can now add triangles to perform movements in 2D.

## Description of widget package nodes

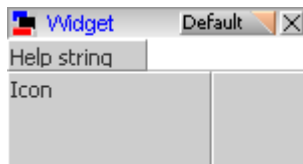
### Widgets



This is widget subsystem node. If Active checkbox is checked, the widget subsystem starts to wait for selection changes and activates/deactivates widgets from inside of encapsulator.

- **Active** enables or disables widgets subsystem
- **Cache size** specifies maximum number of widgets to hold in cache. If there is no space for new widgets, the last one used is replaced with newer one.

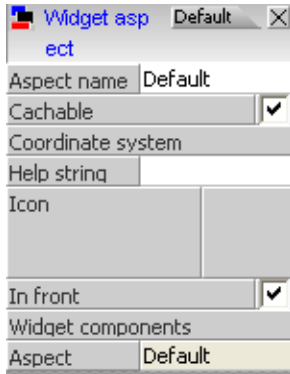
### Widget



The widget node is the encapsulator for widget prototypes. Help string and icon are used for Swap widget menu for tooltips and for better widget identification.

- **Icon and Help string** could be used by another tools to visualize toolbar aspects. Currently, this is not used and can be left blank.

## Widget Aspect



The widget Aspect is used to specify the widget aspect within the widget. The **Coordinate system** input connector specifies the default coordinate system to be used with nodes if no coordinate system is specified by linking connectors to coordinate system nodes.

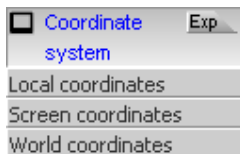
Help string will be used for displaying tool tips, and Icon will be used in change aspect menu. In front checkbox specifies whether widget should be rendered in front of all geometry, or should be mixed into the scene. Cacheable checkbox specifies whether widget aspect is cacheable or not.

Specifying the default coordinate system in this way greatly reduces the number of required links and clears the widget design. During the activation, all unconnected coordinate system connectors will be linked to the Coordinate system link source node and connector.

- **Aspect name** is name of aspect of widget, under which it is accessible.
- **Cacheable** when true, allows caching of widget aspect.
- **Coordinate system** specifies default coordinate system to use for unconnected coordinate system links.
- **Docking parameters** is a symbolic input connector that specifies which base widgets are active with the specified aspect.
- **Icon and Help string** could be used by another tools to visualize toolbar aspects. Currently, this is not used and can be left blank.
- **In front** specifies whether the widget should be rendered in front of geometry or should be mixed with it.
- **Aspect** is output of aspect name.

## Coordinate systems

Coordinate system nodes are used to specify coordinates in which widgets operate.



The **Coordinate systems** are used to evaluate the coordinate system in which widget operates. There are three coordinate systems supported by this node: *Local*, *Screen*, *World*, accessed through appropriate connectors.

### Perspective compensated coordinate system

This is similar to Widget**coordinate system**, but in addition it scales coordinate system matrix so that the widget will have the same on-screen size (not counting perspective distortion).

### Preferences coordinate system

This coordinate system changes according to settings of coordinate system under LE/Preferences/Modeling preferences – it supports World, Local and Screen coordinates.

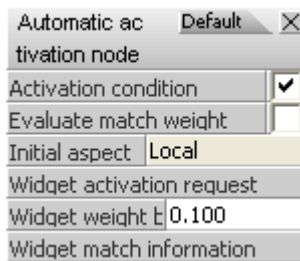
### Persp. comp. preferences cs

Is similar to above, but in addition it scales coordinate system so, that it widget will have the same on-screen size (when not counting perspective distortion).

## Activation nodes

Activation nodes are used in automatic widget activation process. For more information, please refer to Widget Activation subchapter.

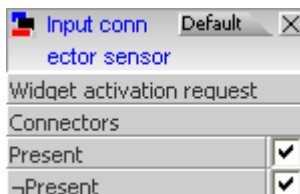
### Automatic activation node



For automatically activated nodes, the **Automatic activation node** is used to scan the widget and object for which the widget is being searched and evaluate the match weight of the widget and the actions it contains.

- **Activation condition** connector is an input node that specifies activation condition.
- **Evaluate match weight** specifies whether to evaluate widget match probability on action level or not. It is possible to disable this evaluation for performance purposes if the activation condition is strong enough.
- **Initial aspect** contains the aspect for widget cloning.
- **Widget activation request** has to be visible to widget subsystem. Through this connector widget activation sends a list of objects and other resources required for activation condition determination.
- **Widget weight bias** specifies a value to add to match weight after match weight evaluation. If Evaluate match weight is unchecked, this value is returned as widget match weight.
- **Widget match information** is an output connector that serves for returning the activation weight, selected aspect, and anchor node to widget activation command.

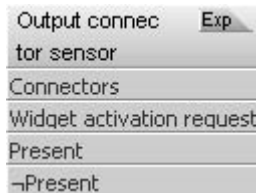
### Input connector sensor



Input connector sensor node is used for scanning connectors of nodes for the presence of a particular connector. List of nodes is provided by the widget subsystem through Widget activation request, and connectors are specified by linking them to Connectors connector.

- **Widget activation request** provides a list of nodes for which the connector presence should be verified.
- **Connectors** is a special associative connector. It can be connected to any connector. The Link Editor contains rules that this connector cannot be used to connect two incompatible connectors. Therefore, this connector only accepts dropped links. You can link more connectors to this input.
- **Present** output specifies whether any of the provided nodes contained connectors specified by Connectors connector.
- **Not Present** is a negation of **Present** output.

### Output connector sensor

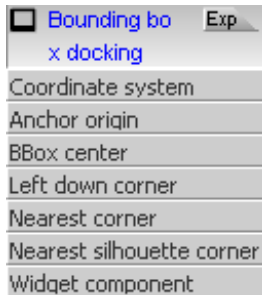


Output connector sensor is similar to **Input connector sensor**, but instead of input connectors, it scan for output connectors.

### Docking nodes

Docking nodes are used to specify position, where widget should be fixed.

### Bounding box docking



Bounding box docking uses the anchor's mesh bounding box **evaluated** in Coordinate system to compute various docking positions.

If Docking parameters are connected to active aspect, all nodes that are connected to bounding box docking positions will be activated during activation process.

- **Coordinate system** specifies coordinate system in which the bounding box should be **evaluated**.
- **Anchor origin** is the origin of the anchor's local space.
- **BBox center** is center of the evaluated bounding box.
- **Docking parameters** are used for activation.
- **Left down corner** is the docking position of the corner nearest to left down corner.
- **Nearest corner** is the nearest visible bounding box corner (to the camera).
- **Nearest silhouette corner** is the nearest visible bounding box silhouette corner (to the camera).

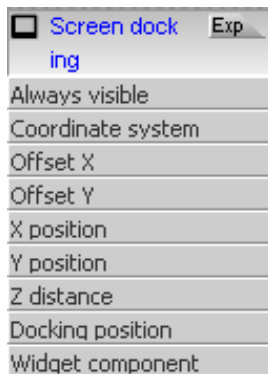
## Offset docking



Offset docking takes the coordinate system and evaluates docking position as a center of coordinate system translated by offset. Offset X in x-**direction**, Offset Y in y-direction, Offset Z in z-direction where x,y,z axes are defined by Mode. Mode 0 uses coordinate system axes, Mode 1 uses world coordinates, and Mode 2 evaluates offset in camera space.

- **Coordinate system** is the input coordinate system, used to take center point.
- **Mode** specifies axes for offsetting. It can be one of following:
  - **0** – **coordinate** system axes
  - **1** – **world** coordinates
  - **2** – **camera** space
- **Docking parameters** are used for activation
- **Docking position** contains the evaluated docking position.

## Screen docking



Screen docking node evaluates the specified position in screen space. Screen space position is defined by relative and absolute coordinates. The distance is specified by Z distance input. Relative position is defined by X and Y position inputs. Screen resolution is -1,-1 to 1,1. Absolute position is defined by Offset X and Y and is in pixels.

## Selection center docking



Selection center takes current selection and evaluates its center as an **average** of selected nodes' origins.

## World docking center



**World** docking returns simply world origin (0,0,0).

### Anchor origin docking



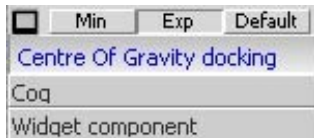
Node evaluates position of anchor node.

### Bounding rectangle docking



Node evaluates four docking positions, each for one screen rectangle corner. It expects, that Coordinate system input is connected to Screen coordinate system.

### Center of gravity docking

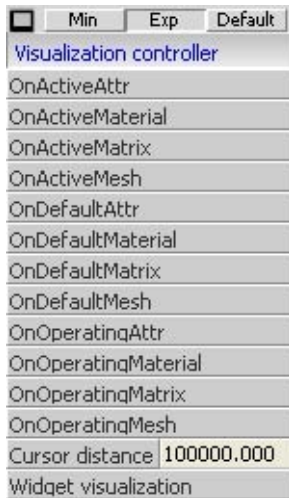


Node returns Center of gravity position

## Visualization

**Visualization** nodes define how base widget shape should look like. They provide mechanism to change widget material, shape, or transformation matrix depending on current widget state.

### Visualization controller



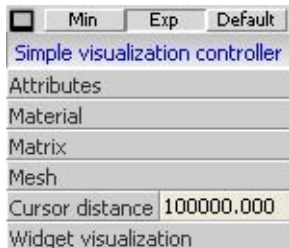
The Visualization controller selects widget mesh according to its three states:

- Active – mesh caused tool activation
- Operating – widget is in active state and performing some actions
- Default – all other states.

The controller also provides cursor distance output, which ranges from 0 to 1 (0 is when mouse floats above mesh, 1 is when mouse cursor is more than 100 pixels away from mesh). Note, than if you do not specify OnOperating or OnActive mesh, the base widget will hide.

- **On\*Material** is material input for state \*. Because widgets are independent from D3D view, they have to allow usage of different material sets. To accomplish this, material input accepts all data types.
- **On\*Attr** is mesh render attributes input for state \*.
- **On\*Mesh** is mesh shape input for state \*.
- **On\*Transform** is mesh Euler transformation input for state \*.

### Simple visualization controller



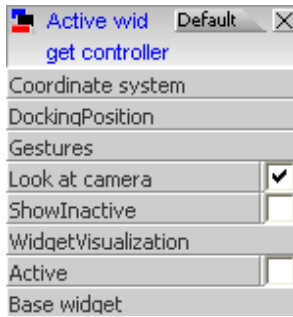
Functionality of this node is similar to Visualization controller, but it provides only one mesh, matrix, and material input. It was designed for optimization purposes.

Use this node if you do not want your base widget to hide or change shape during manipulation with it.

### Widget action controllers

Widget action controllers manage all widget actions. They control activation, deactivation, undo, redo, and various actions connected with widget runtime management.

#### Active widget controller



Active widget controller manages all actions related to base widgets, which includes: widget action activation and deactivation, undo - redo operation grouping, transformation of coordinate systems, base widget state handling and transformation matrices evaluation. Active widget controller communicates directly with D3D view window through Base widget connector.

- **Coordinate system** specifies transformation matrix to apply on mesh shape. Also specifies the coordinate system in which widget actions operate.
- **DockingPosition** defines position of widget. Widget is displaced to this position.
- **Gestures** connector is used to connect multiple Gesture nodes to visualization controller node. Together with Gesture nodes, this node manages commands and widget tools execution.
- **Look at camera** specifies whether the coordinate system should be rotated such that camera is in positive octant. This is useful for more complex widgets, because the widget will be always facing toward camera.
- **ShowInactive** determines whether the widget will be rendered even if all its actions are not activated.
- **Active** output is true if this base widget is rendered
- **Base widget** is the communication interface to the view window.

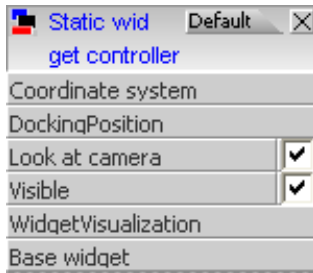
### Background widget controller



Background widget controller handles management of background base widgets.

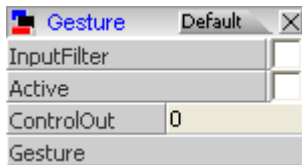
- **Priority** specifies tool background tool priority within tool management.
- **Orientation** defines coordinate system for action
- Other connectors are the same as for Active widget controller

### Static base widget



The Static widget controller is intended to simplify definition of widget parts **without** actions. This node therefore does not contain Gestures input, and visibility is determined directly from its Visible input connector.

## Gesture



Gesture node connects actions and activation conditions with base widget action controller. If the mouse cursor is above the widget shape, the base widget action controller sends this message to gesture node. Node checks whether activation condition is true (defined through InputFilter). If yes, gesture initializes connected tools and activates them. Gesture can activate two types of actions: Activities (activated on button release), and Widget tools (activated in tool style).

- **InputFilter** specifies whether key and mouse button combinations are valid for tool activation.
- **Active** is true if any of widget tools can operate on controlled nodes.
- **ControlOut** is output activity or widget tool trigger.
- **Gesture** is node output.

## Widget actions

Widget actions actually do the work. They are either activities designed specially for widgets (change aspect, for example), or widget tools (like Slider move). They work on a list of nodes – controlled objects.

### Change aspect

Change aspect command displays change aspect menu. Currently, coordinate system selection is implemented through aspects.

### Close widget

Close widget command closes current widget. If it is activated, calling widget will be deactivated.

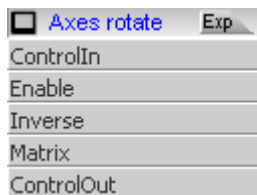
### Axes swap



Widgets operate on the same coordinate system that is used to transform widget shape to world space. Sometimes it is required to alter this default behavior. For example, trueSpace's default behavior for movement is that LClick moves in XY, and RClick moves in Z. This operation is however accessed through one surface. This node can be used to reorder and swap coordinate system axes.

- **ControlIn** is activation input
- **Swap** is Boolean input. The coordinate system is altered only if this input is true.
- **X axis ID...Z axis ID** are IDs of new axes. 1 specifies X, 2 specifies Y, 3 specifies Z, -1 specifies -X, etc.
- **ControlOut** is a trigger for the following tool.

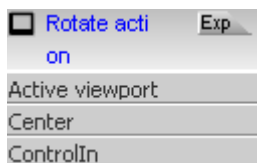
### Axes rotate



Axes rotation provides similar, but more general functionality as Axes Swap. With this node, you are able to transform the coordinate system with a matrix. The matrix can be inverted before multiplication. Multiplication is performed only if Enable is true.

- **ControlIn** is the activation input.
- **Enable** enables matrix multiplication.
- **Inverse** selects between multiplication of inverse transformation or direct transformation.
- **Matrix** is the input for transformation matrix.
- **ControlOut** is the trigger for the following tool.

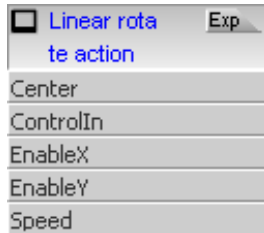
### Rotate action



Rotate action performs rotation of the controlled objects around Center. Rotation is performed around widget Z axis. Angle of rotation is evaluated in widget XY space.

- **Active viewport** is the widget management input.
- **Center** specifies center of rotation. It can be connected to any docking position.
- **ControlIn** is the activation input.

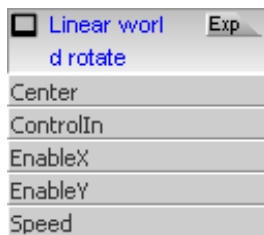
## Linear rotate action



Linear rotate action performs rotation of the controlled objects around Center. Rotation is performed around the X and Y axes, angle is computed from mouse step and projection of widget X axis to screen. Rotation around X or Y axis can be disabled or enabled by EnableX and EnableY inputs. Rotation speed can be controlled by Speed input.

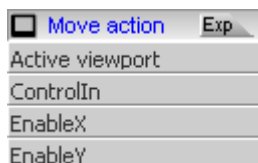
- **Center** defines the center of rotation.
- **ControlIn** is the activation input.
- **EnableX, EnableY** enable or disable rotation around X and Y axes.
- **Speed** specifies the ratio between mouse step and angle step. 0.5 means 0.5 degrees to one mouse step.

## Linear world rotate



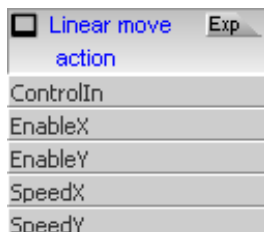
Linear world rotate performs a similar action as the node above, but instead of rotating in widget space it always performs rotation in world coordinate space.

## Move action



Move action performs movement of controlled objects. Movement step is computed from the widget local coordinate system so that moved object tracks the cursor. If movement along both coordinate axes is enabled, widget XY plane is used. If only one axis is enabled, the second one is selected to make movements most precise.

## Linear move action



Linear move action performs a similar operation as the Move action, but the difference is in movement step calculation. Slider type move action does not count perspective into movement step evaluation. Movement speed can be altered by the Speed input.

Widget move



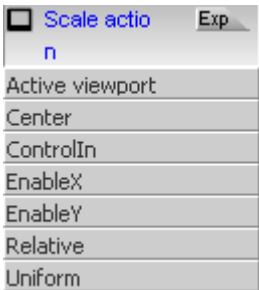
Widget move moves widget from its original position. Movement is performed in the widget shape’s XY space, where movement along X and Y axes can be enabled or disabled with the EnableX and EnableY inputs.

Screen docking move



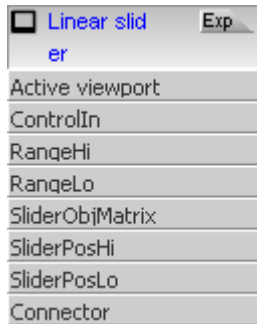
Screen docking move changes relative position of the widget on the screen **docking**, therefore this node can be only used with screen docking node. Action is linked to docking node by the Docking position input. Movement can be enabled or disabled for X and Y axes.

Scale action



Scale action performs scaling of controlled objects around Center in the XY widget shape’s coordinate space. Scaling can be enabled or **disabled** for each axis. Scaling coefficients can be computed relative to center of scaling, or they can be evaluated from direction of widget coordinate system. Uniform scaling scales along all three axes with equal coefficient.

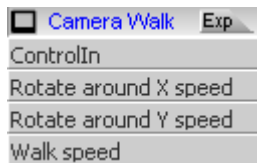
Linear slider



Linear slider provides control of all floating point inputs that request limited real linear range of input values. Slider range is **defined** by RangeHi and RangeLo inputs. Controlled connector is defined by example with connecting Connector to node with requested connector.

- **RangeHi** is the high limit **for** slider values.
- **RangeLo** specifies the low **limit** for slider values.
- **SliderObjMatrix** should be connected to the moveable part of the slider.
- **SliderPosHi** specifies **high** limit of range of movement for moveable slider part.
- **SliderPosLo** specifies **low** limit.
- **Connector** is **used** to set connector you wish to change identification.

### Camera walk



Camera walk starts moving the controlled object in its Z direction with constant speed defined by Walk speed. It uses mouse X and Y coordinates to perform rotation around local X axis (looking Up and Down) and global Y axis (looking left and right).

### Camera look around



Camera look around uses mouse X and Y coordinates to perform rotation around local X axis (looking Up and Down) and global Y axis (looking left and right). Rotation speed can be defined by appropriate speed inputs.

### Camera navigate



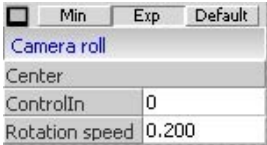
Camera navigate uses the mouse X coordinate to rotate controlled objects around the global Y axis, and the Y coordinate to perform movement in local Z axis. Rotation and Movement speed can be defined by the speed inputs.

### Camera FOV



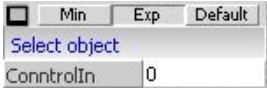
Camera FOV modifies field of view for projective cameras and zoom factor for orthogonal views.

**Camera Roll**



Camera Roll rotates eye camera around axis formed by Center input and camera position.

**Select object**



Select object Command raycasts scene and selects nearest object under mouse cursor