

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, 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, trueClips, 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

Support Email: support@caligari.com

trueSpace7

Developer Guide

Contents

1 DESIGN PRINCIPLES	1
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
2 WIDGETS	29
2.1 LOW LEVEL ASSEMBLY	29
Description of widget package nodes	61
3 MATERIALS	75
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
4 SCRIPTING	131
4.1 SCRIPTING IN TRUESPACE	131
4.2 SCRIPT COMMANDS	139
4.2 SCRIPT OBJECTS	155
5 FILE FORMATS	167
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 3rd 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 3rd 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

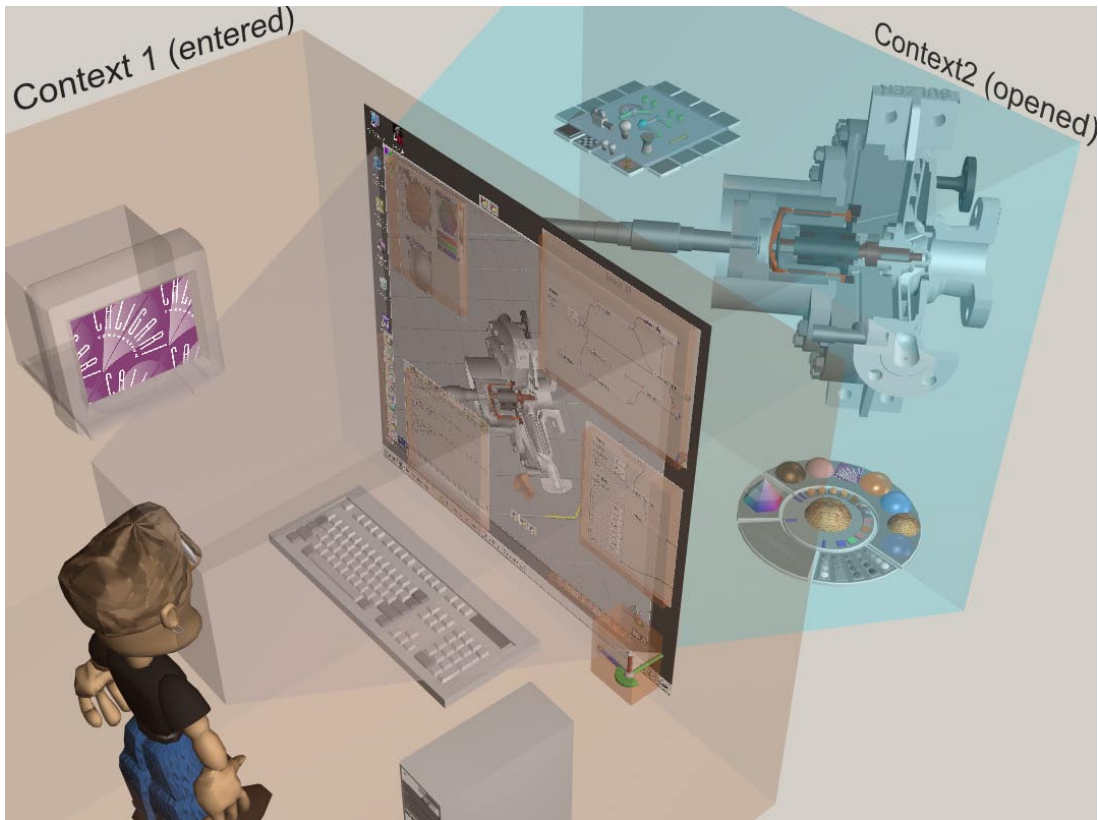
An Activity is a building block for all Rosetta based applications. In Rosetta, activities are sequences of actions with a duration in **time**. Unlike animations (even procedural animations), activities have a **goal**, which the activity participants try to accomplish. If the participants are human 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.

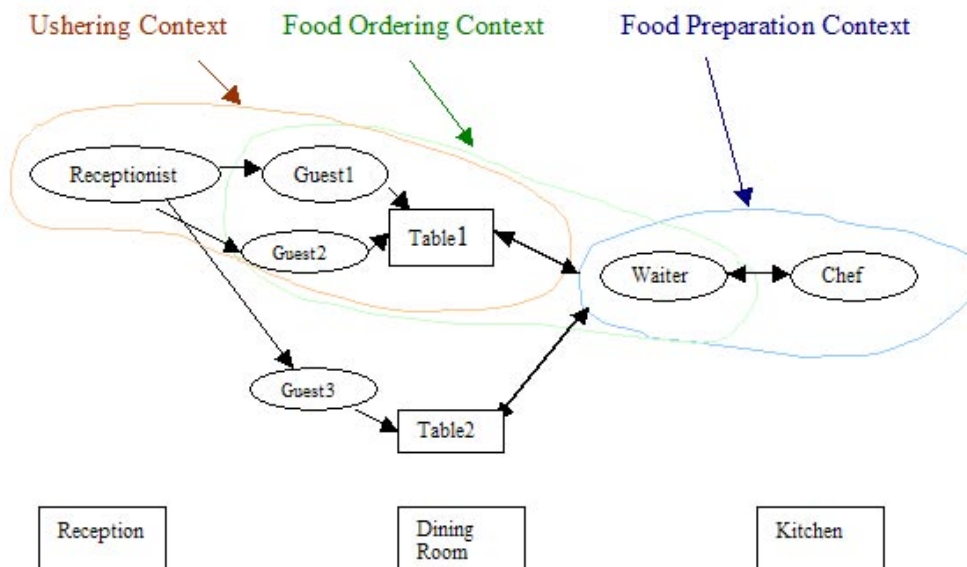
Let us look at an example of concrete, real world contexts and how they participate in a concrete real-life scenario:

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, Surfacer, 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 Surfacer who edits 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

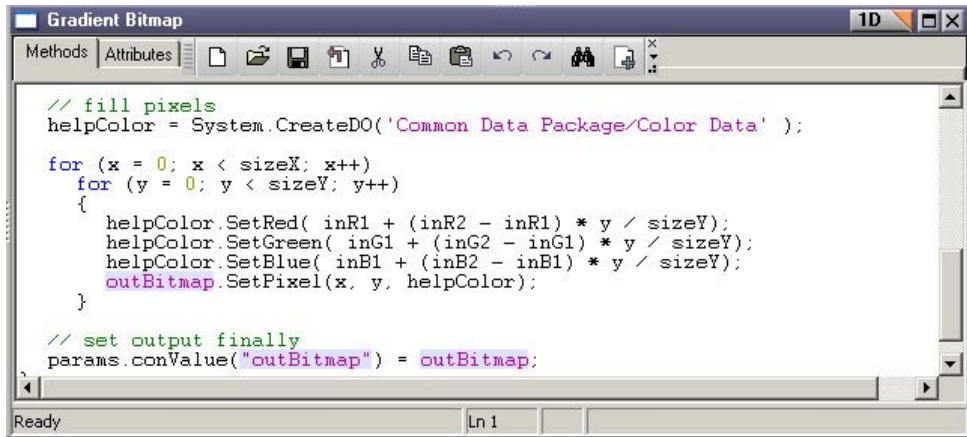
We perceive our world as 3D, but 2D and 1D representations play an important role in many contexts. Artists prefer a 2D user interface (UI) for image and vector based graphics tools, and developers prefer 1D text editors for writing code. Today, most authoring tools hardwire a particular dimension for 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.

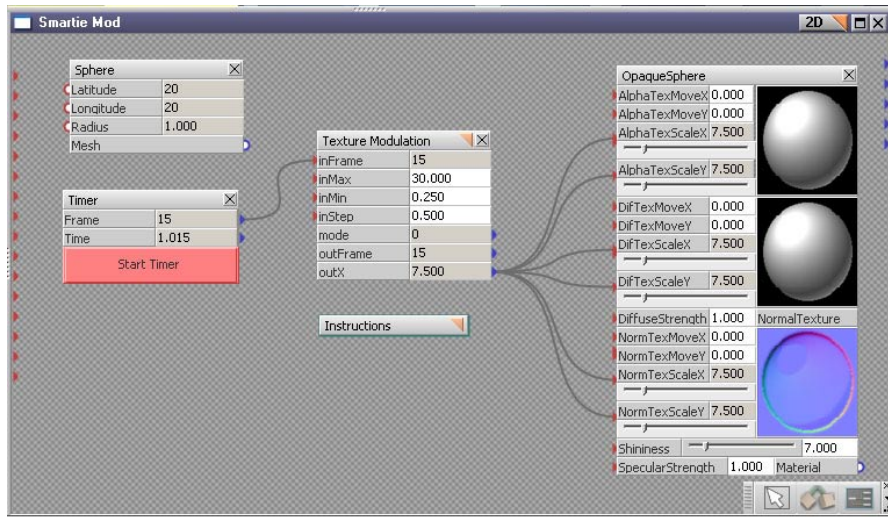


This is how program objects were traditionally designed. In Rosetta, a developer can create a new object in C, C++, JAVA or any other language using a favorite programming environment and then compile it into a binary COM Rosetta object. In addition to this, users can create script objects at run time 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)

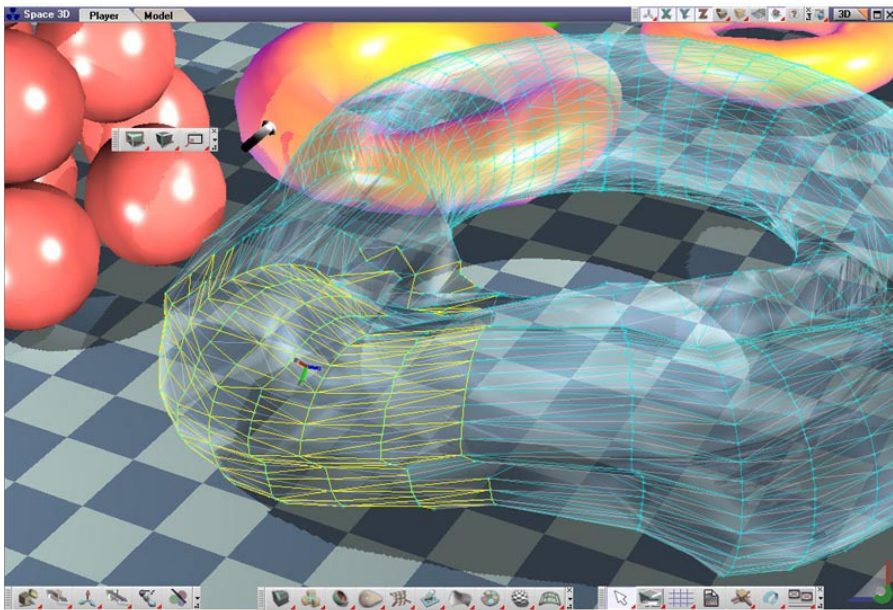
If programming or scripting is not your cup of tea, you can still create first class complex Rosetta objects, including procedural and interactive objects, in the 2D Link Editor by simply dropping existing objects from libraries into a 2D LE window, linking them with visual links, and then encapsulating this 2D structures 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 interactive 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.

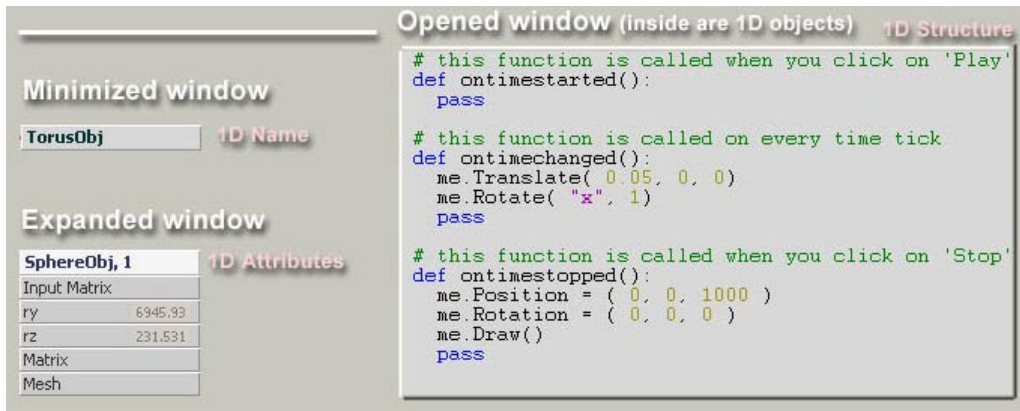
You, the user, also do not exist in a vacuum. In fact you must always be located **inside** of some Rosetta object. That object, together with its current state, is called a **Context**. Context is not static though. You can always decide to exit the current context and enter another one, such as when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

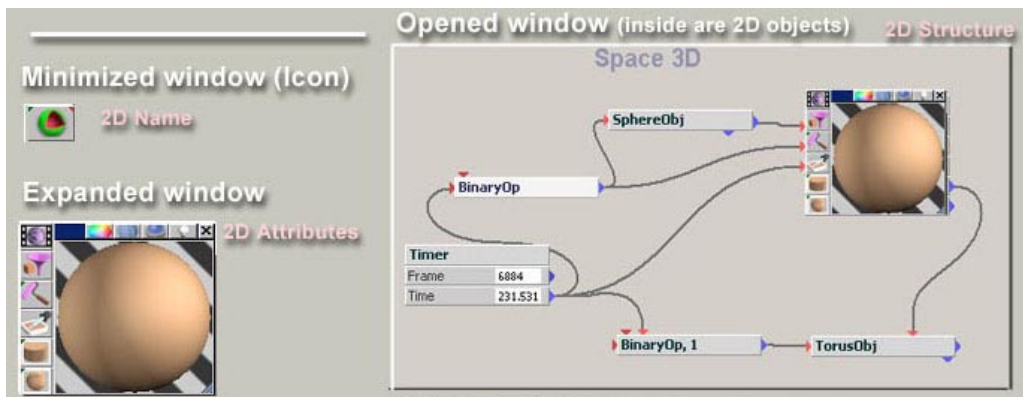
1D Interaction Aspects

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).



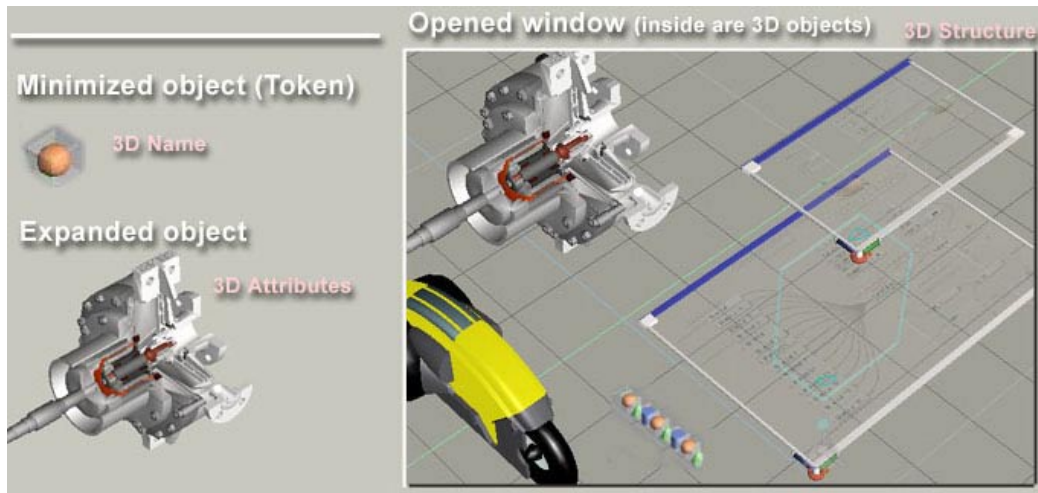
2D Interaction Aspects

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 **symbiosis** 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.

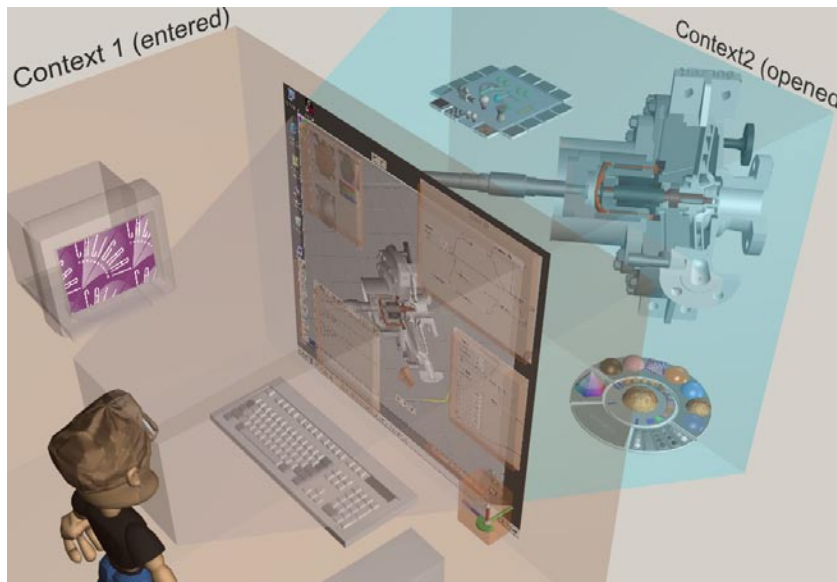
Chapter 2

Widgets

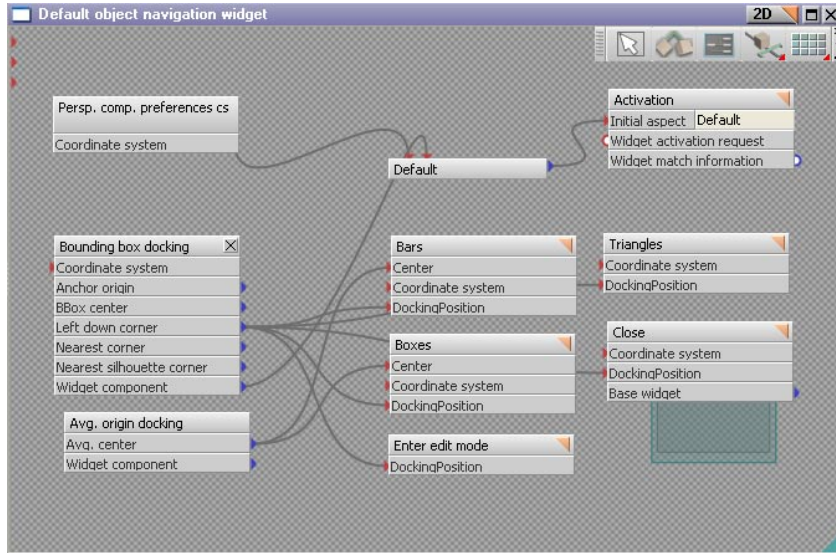
2.1 Low Level Assembly

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.

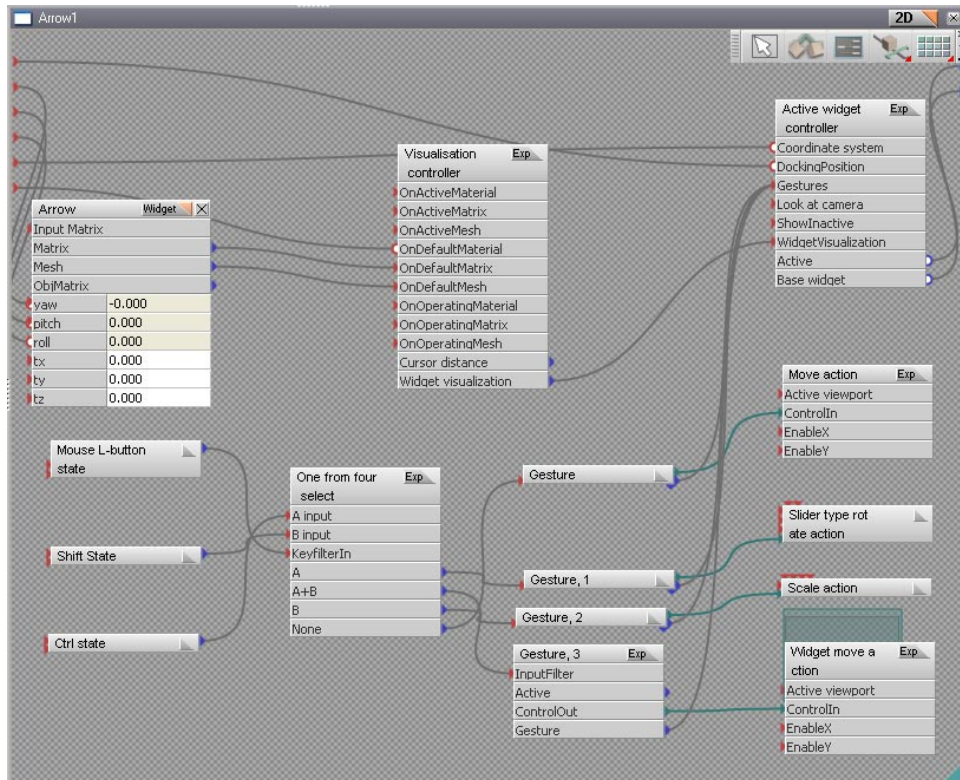


Simple widgets

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*.

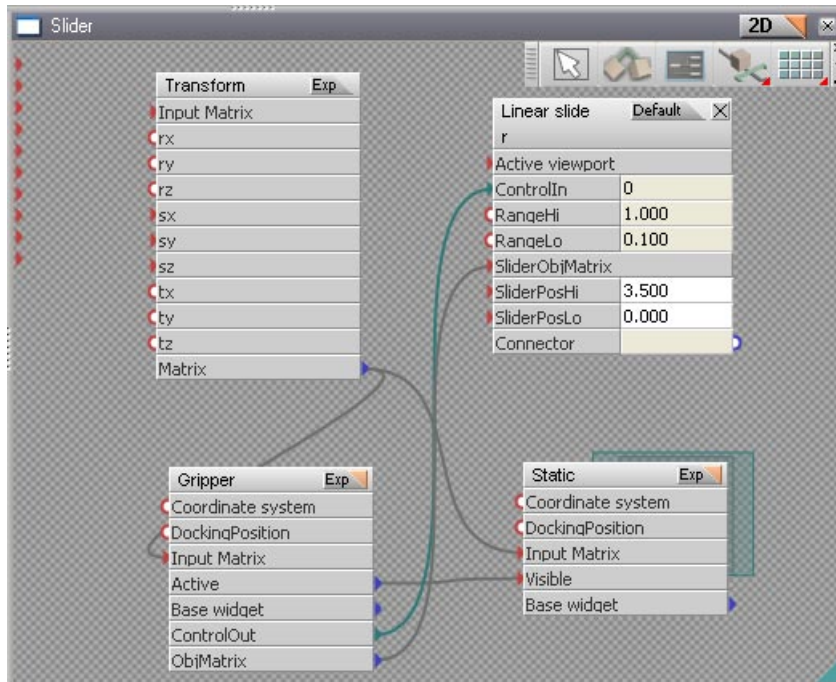


The Arrow node defines shape and consists of shape and Euler transformation nodes. The *Visualization controller* handles the selection of the current shape according widget state. The *Active widget controller* handles base widget management (matrices setup, tool messages dispatching and gestures management).

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.

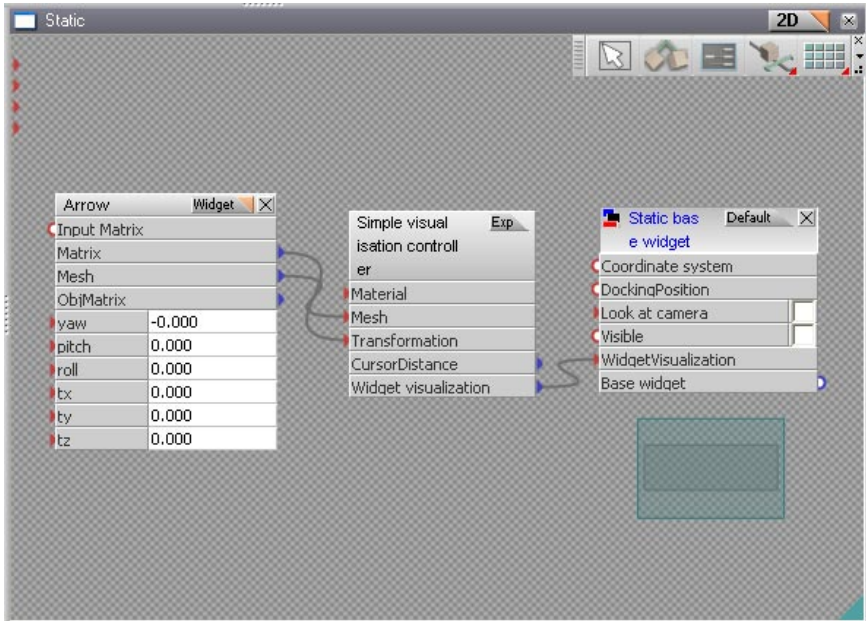
Widgets with decorations

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

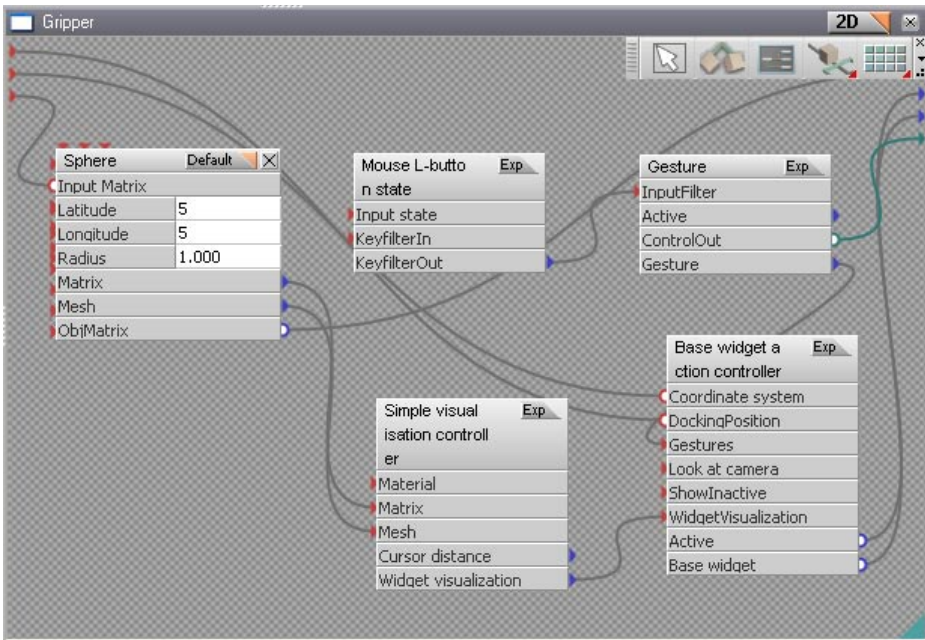


The Slider control contains a Static part (the one without linked actions and a gripper) that can be dragged and moved. The Active part is connected to *Linear slider* node, which is widget tool.

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

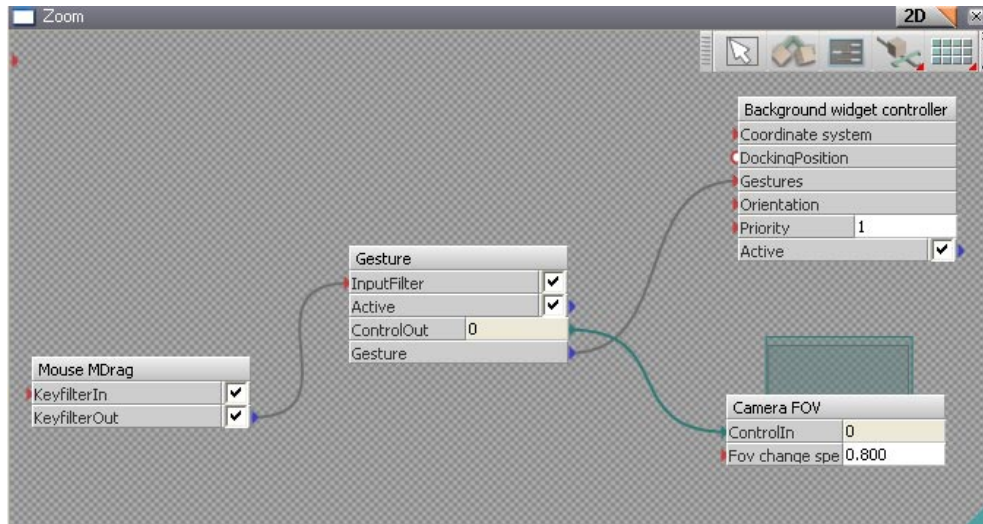


The Active part resembles simple widgets. Note, however, that they export *Active* connector, so the static part during activation knows if base widget active control is active or not.



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.

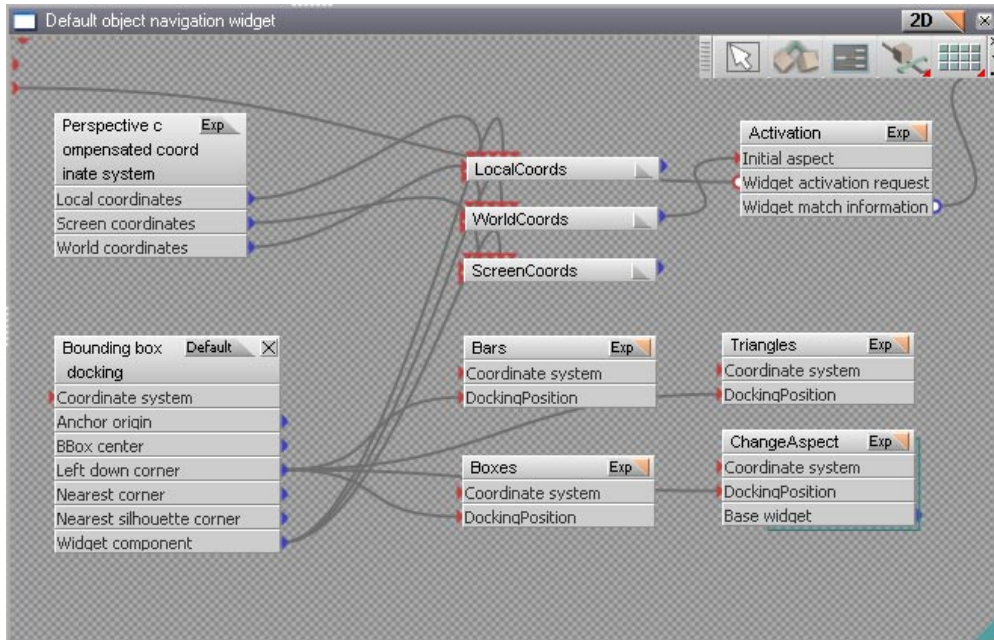
Designing widgets

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.

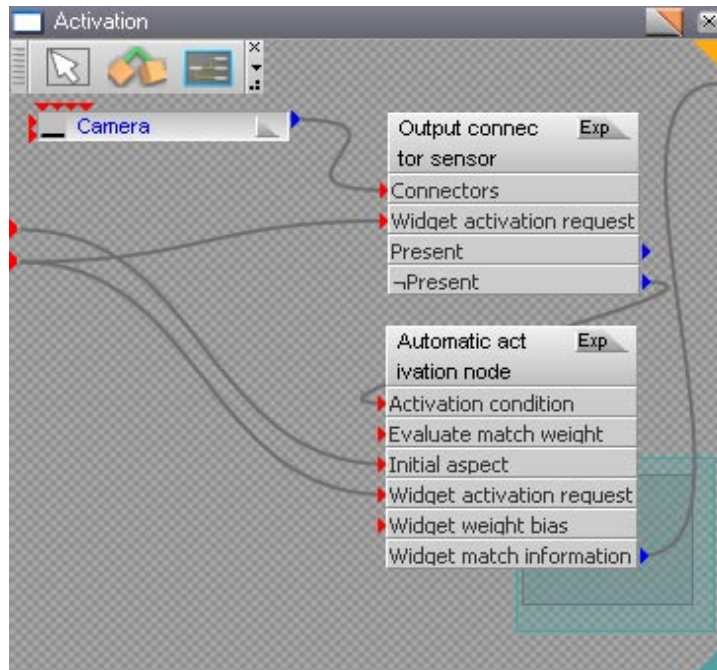
The Widgets library currently provides only limited number of different actions, whose descriptions are available at the end of this chapter. However, you are free to create your own widget tools using scripts or by creating new Rosetta packages.

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** check box. 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.

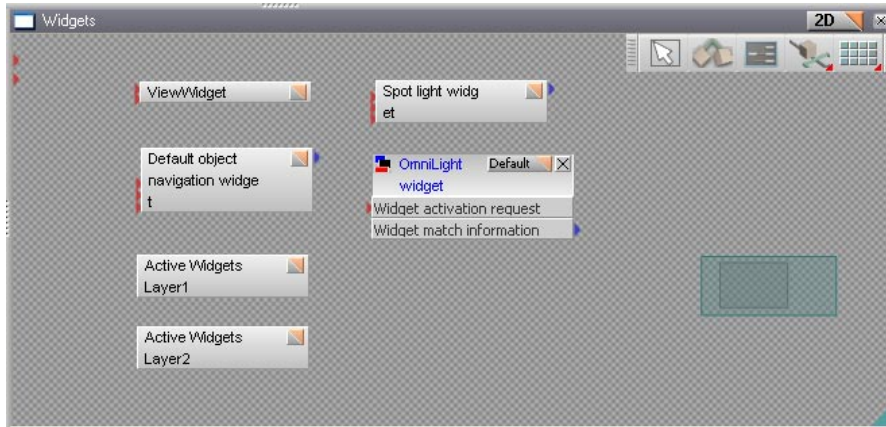
Designing new widgets

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.

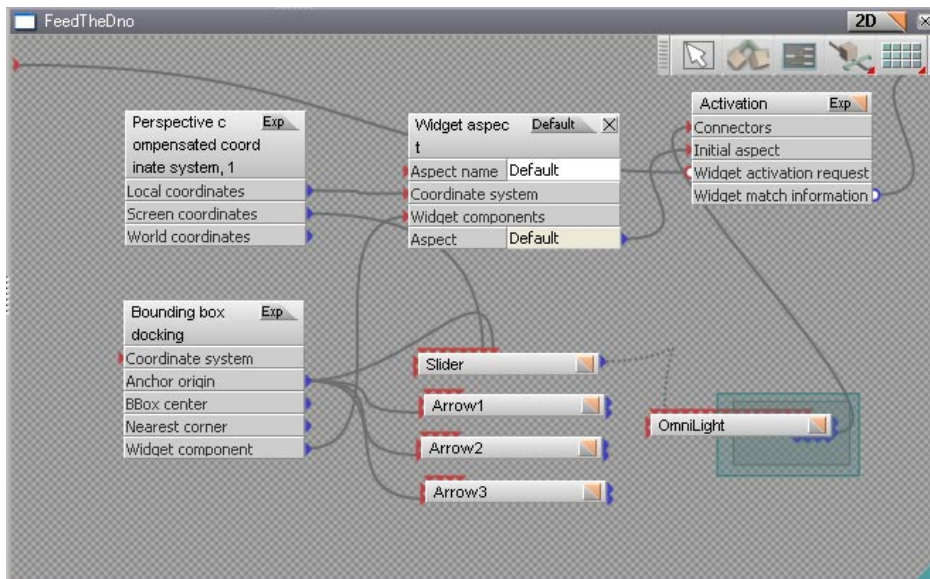
Feed the dino widget example

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

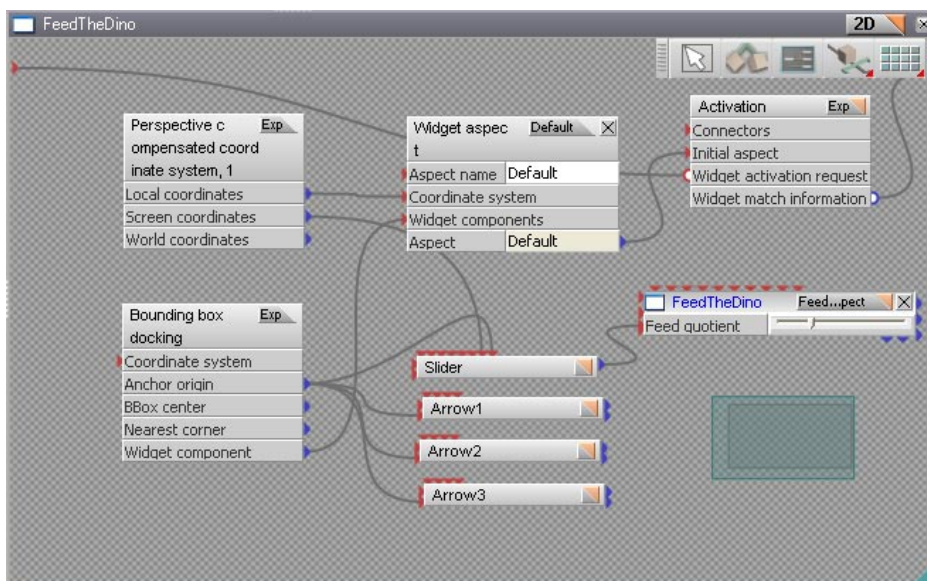
First, locate `OmniLight` widget in widgets library and drop it to `Widgets` subsystem encapsulator. The `Widgets` encapsulator can be found in Desktop.



Rename the OmniLight widget to FeedTheDino, and open it.



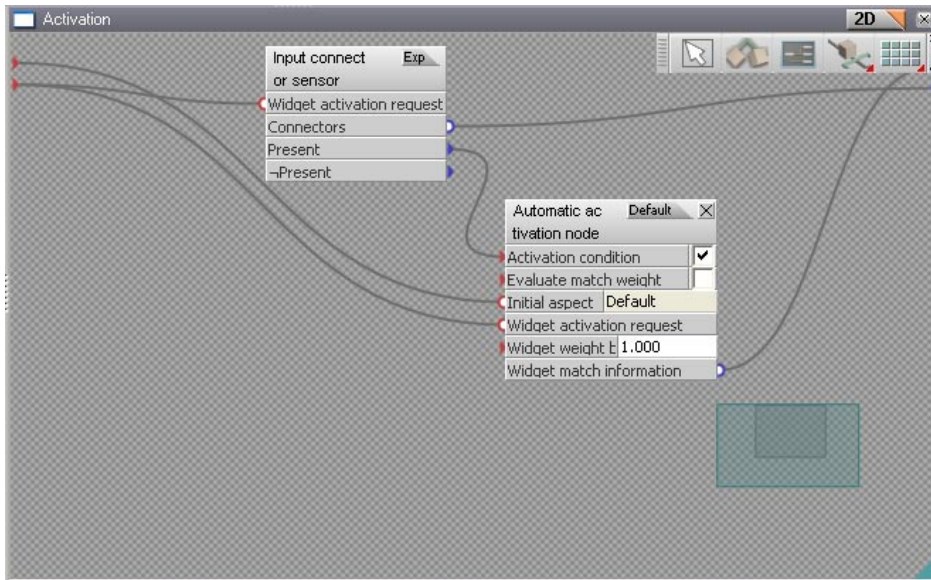
Locate the *OmniLight* node and close it. From scripts objects library, take *FeedTheDino* and drop it to in *FeedTheDino* widget. Link the *Slider Connector* output to *FeedTheDino* quotient.



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

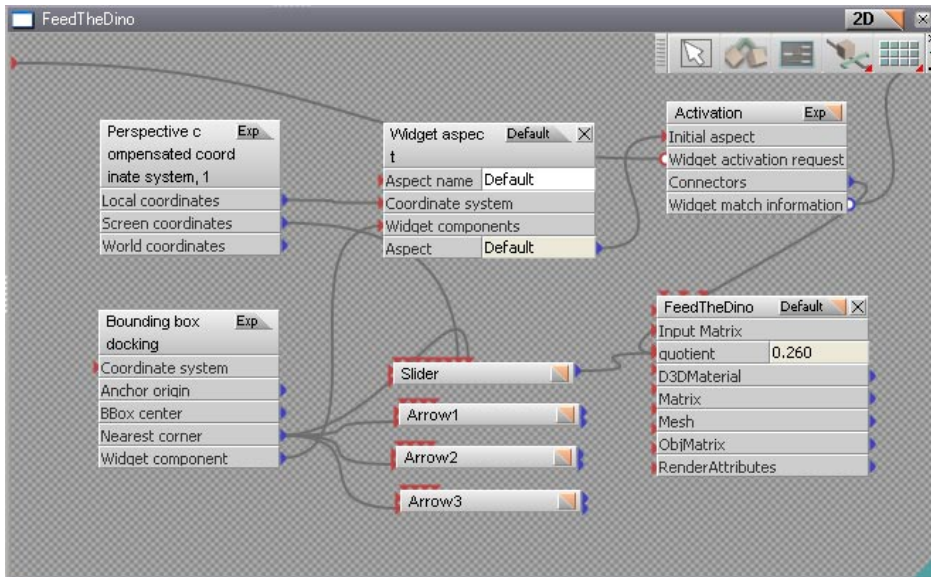
Now, we have to update the Activation node so that the widget subsystem can properly use our new FeedTheDino widget. We will identify the FeedTheDino script object by its quotient input, because if other nodes contain the same Mesh Shrink Filter as dino contains inside, our new widget will operate properly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



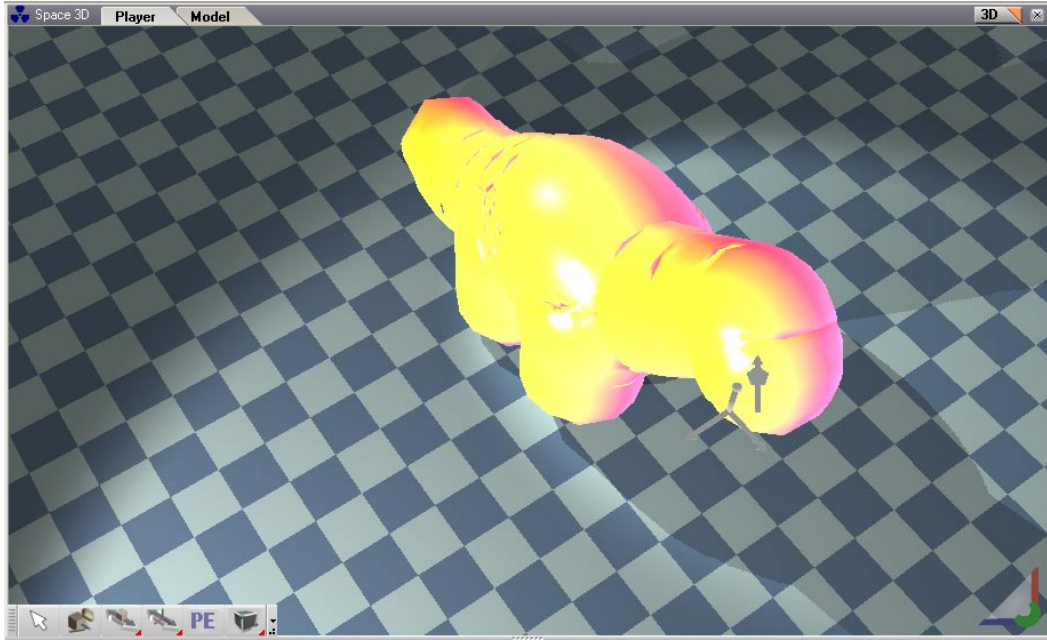
You can also change docking position from Anchor origin to Nearest corner by reconnecting all links from *Bounding box docking* Anchor origin connector to Nearest corner connector. You can specify slider range by RangeLo and RangeHi inputs of Slider node. The good values for FeedTheDino are -1 and 1.

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.

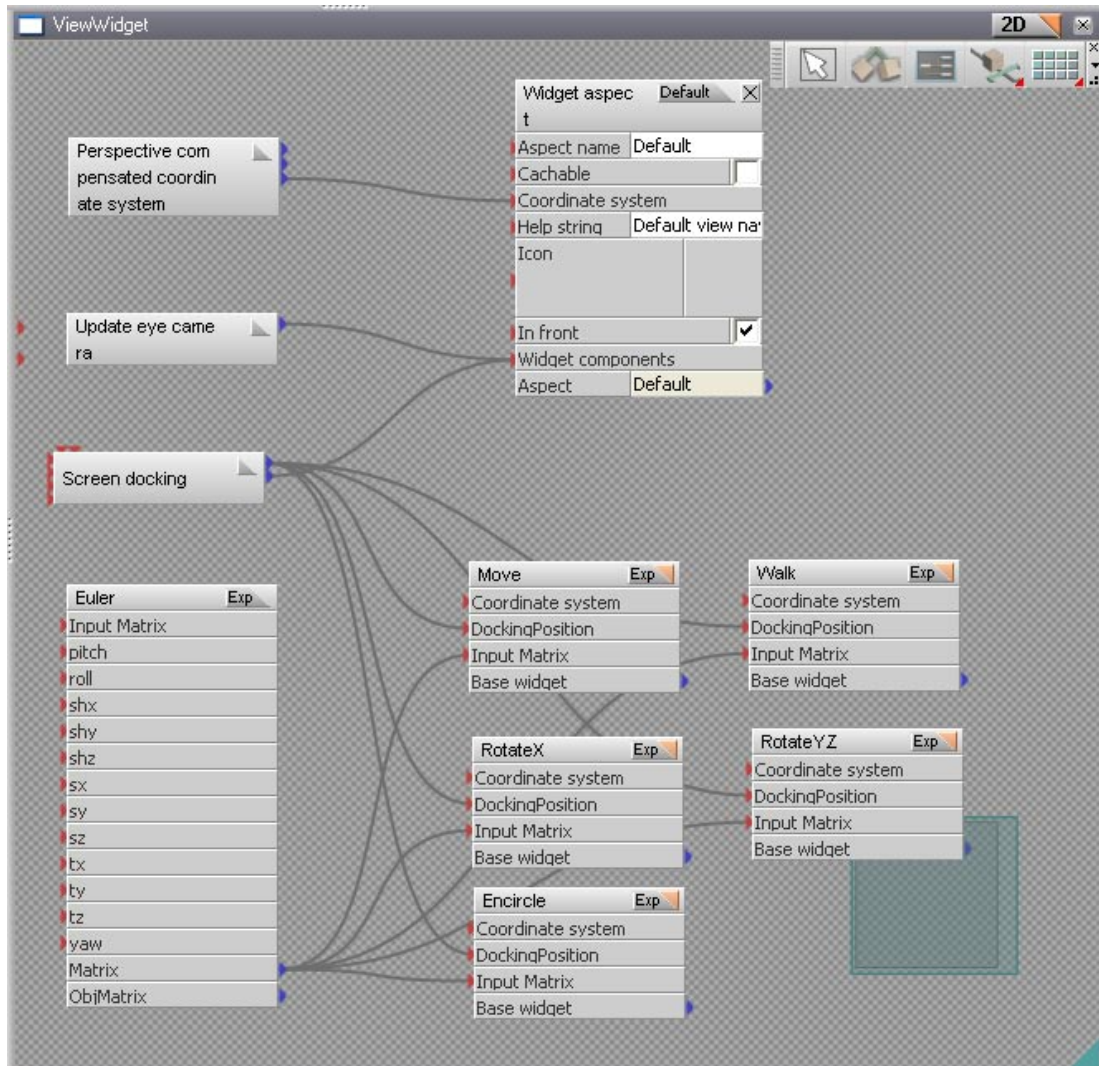
Changing action activation logic (gestures) example

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.

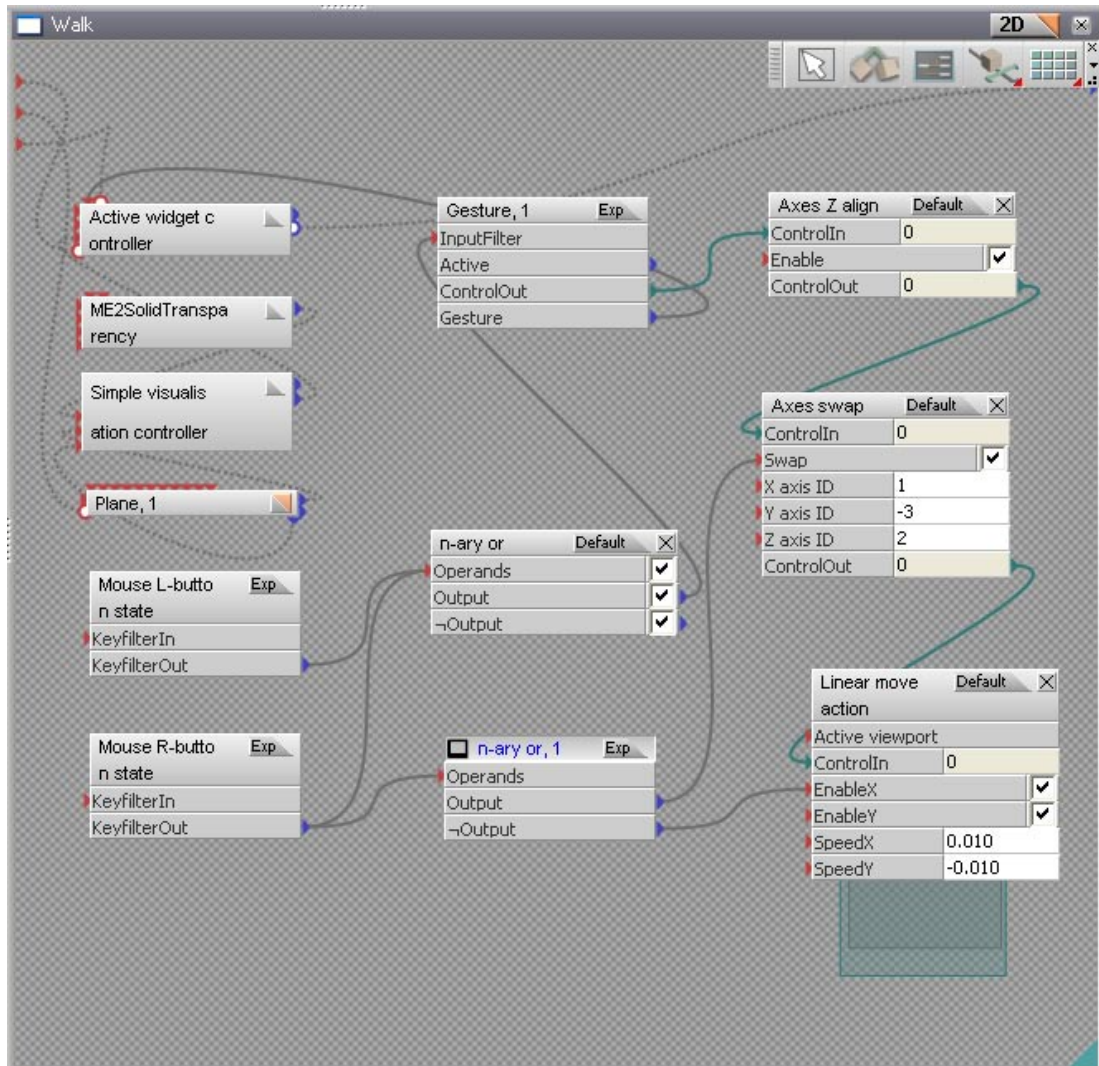
Walk action starts moving the camera forward at the specified speed, while the mouse can be used to look around. This style of movement is well known from computer games.

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.



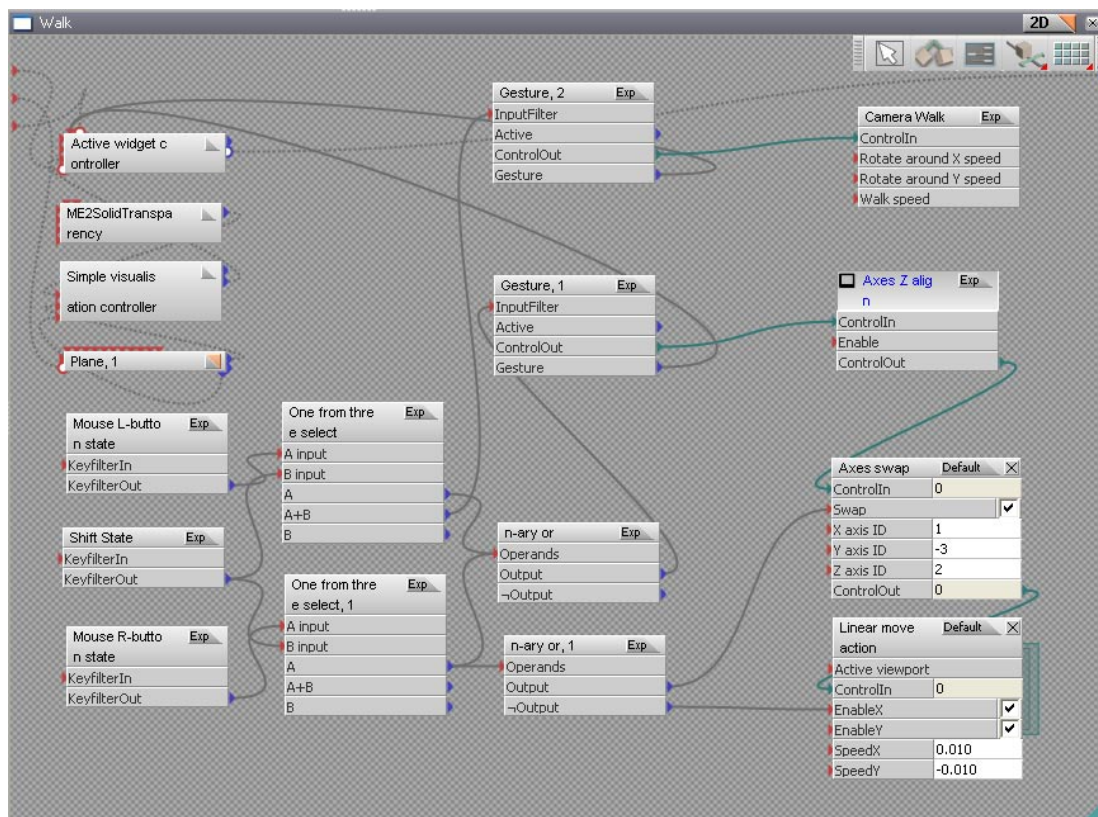
We will modify the Walk base widget, so open it.



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:



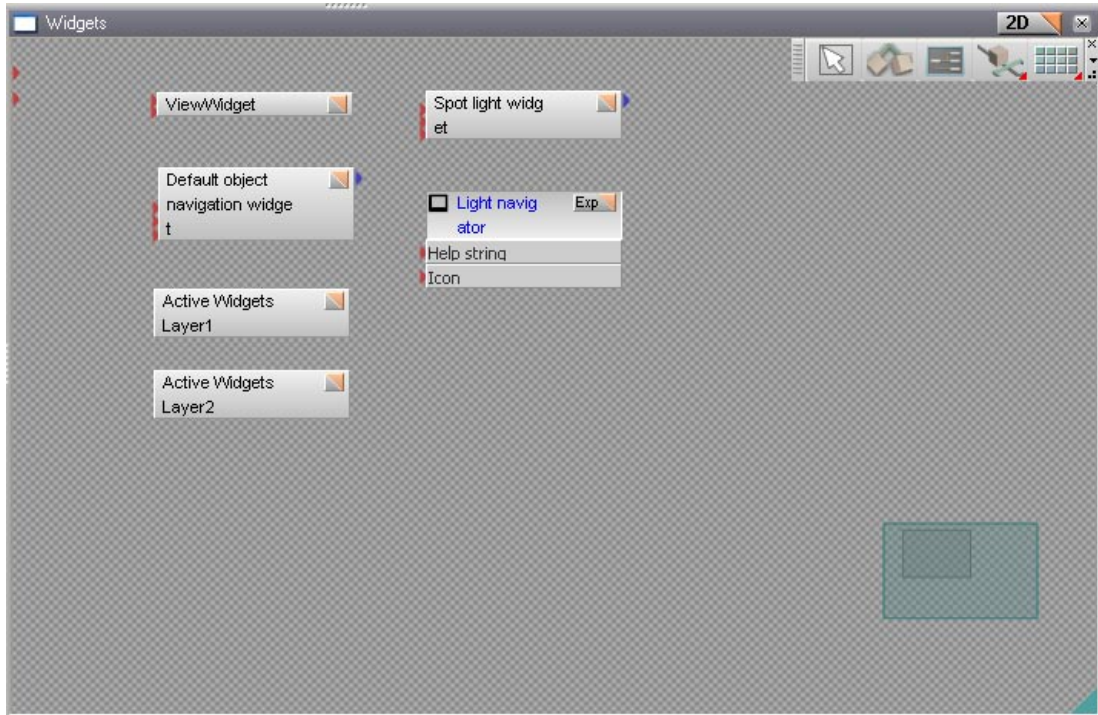
Now, when you press mouse L-button, *One from three select* sets 1 to A output, and only linear move action is activated. When you press R-button, *One from three select, 1* sets 1 to A and everything goes as before. When you hold SHIFT with LClick, *One from three select* sets 1 to A+B output connector 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.

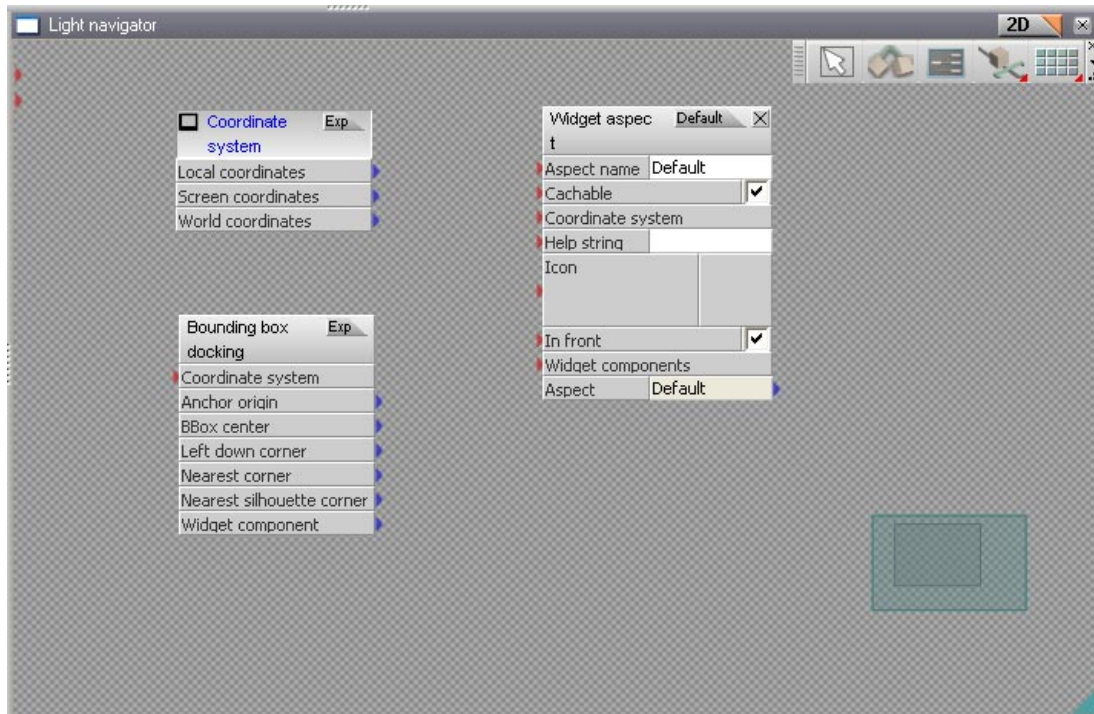
Simple Light Navigation widget example

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.



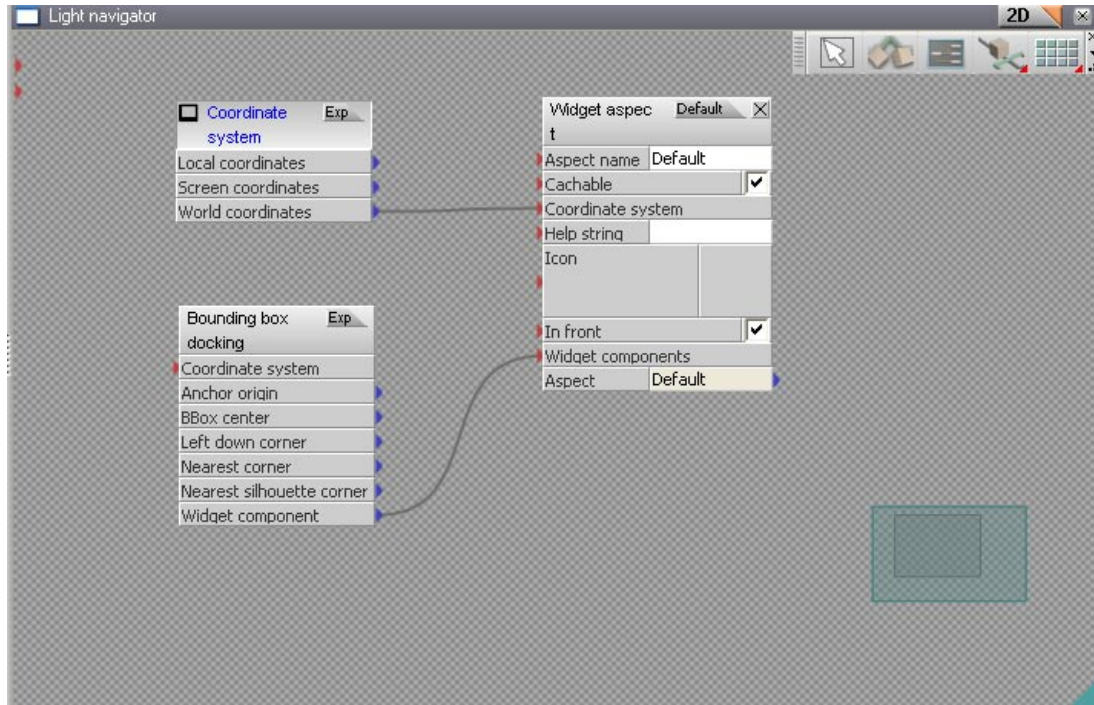
The first thing we need to do for our widget is to add all nodes that are required for widget management and rendering. These nodes are: Widget aspect that specifies widget aspect name and a starting position for widget activation, activation nodes, docking node (**Bounding box docking**) so we will 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**.



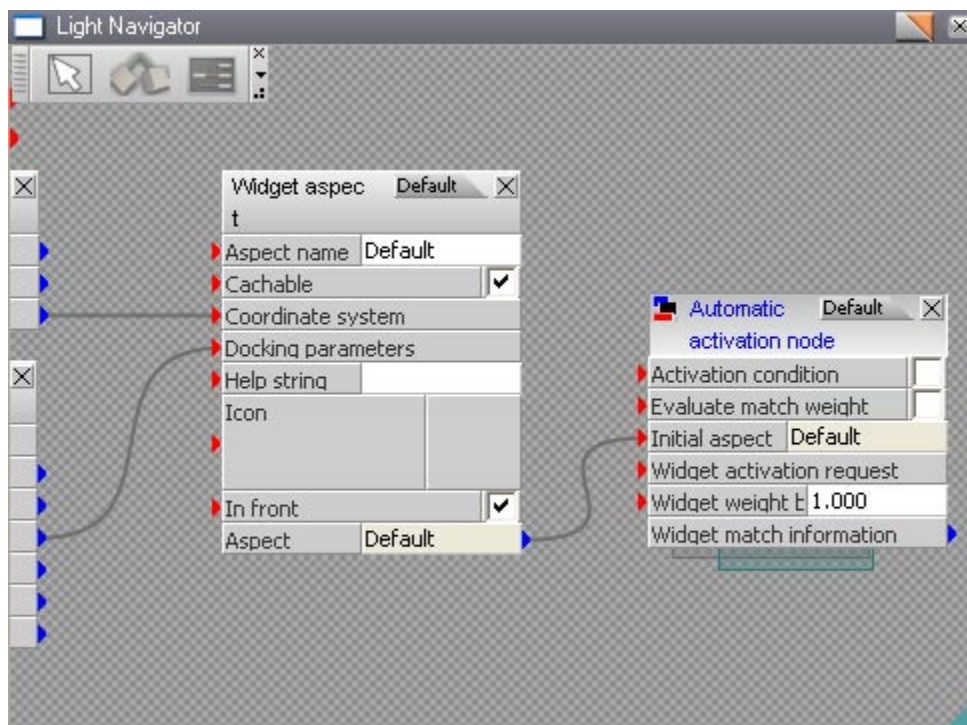
As you can see on widget aspect node, you can specify caching and enable/disable overlapping with scene geometry (with *In front* checkbox). Widget caching takes some memory, but significantly speeds up widget activation and deactivation. Global widget caching strategy can be modified in Widgets encapsulator's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

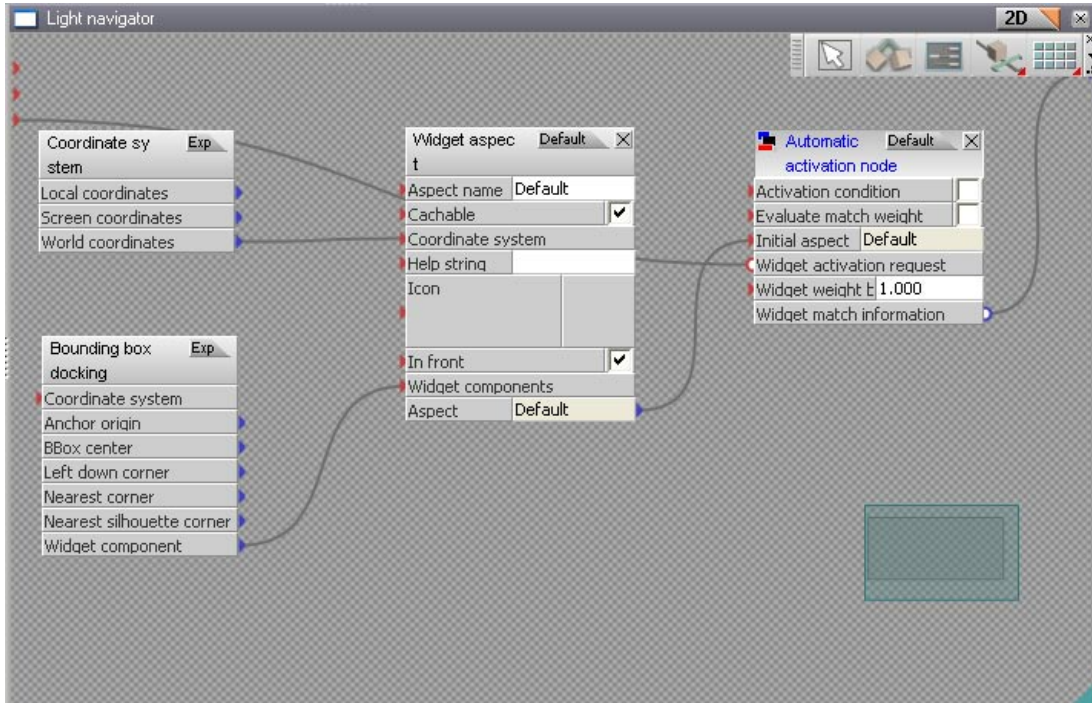
By connecting World coordinates from Widget coordinate system, we say that we want to use world coordinates as default, and by connecting Docking parameters we specify that the default aspect will contain all nodes docked to Bounding box docking.



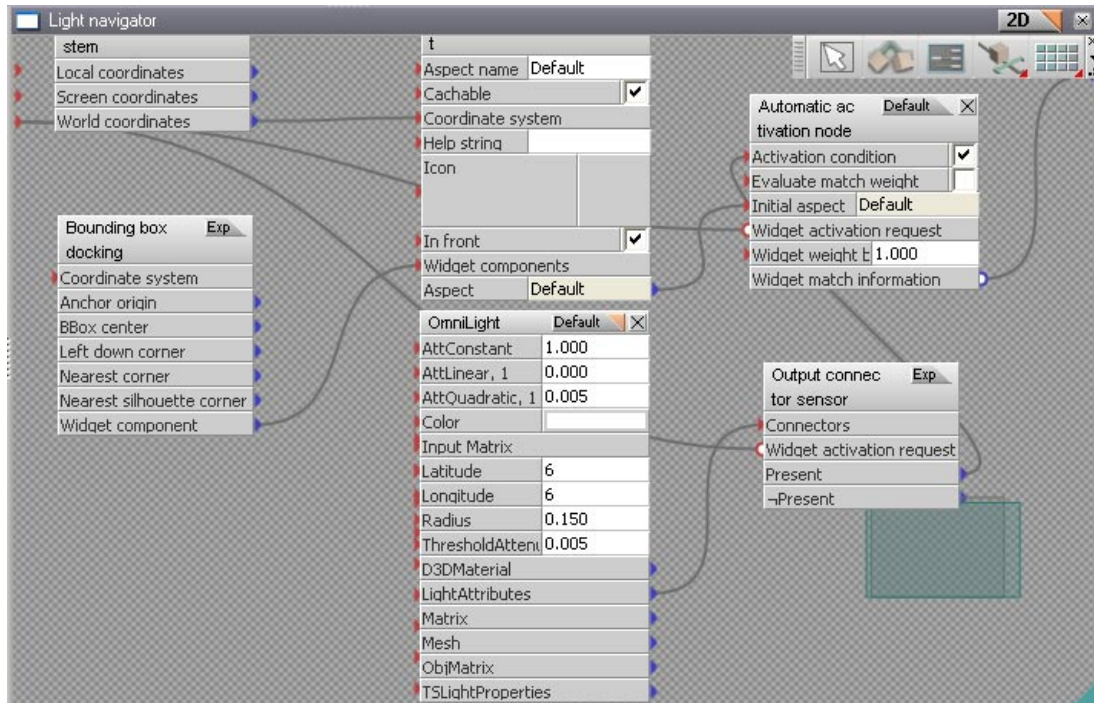
Now, we need to tell the widget subsystem that we want automatic activation of our widget only for lights. This can be performed by adding Automatic activation node to our design.



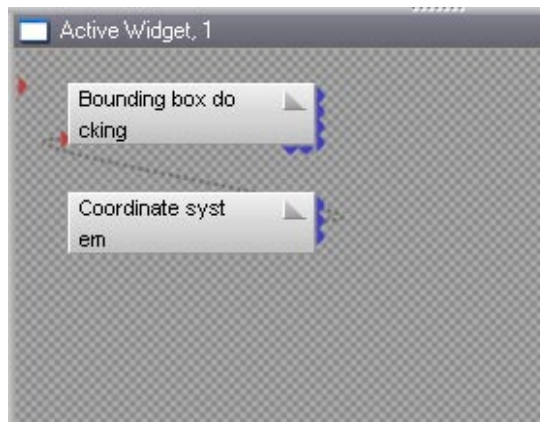
After Rosetta changes selection, the widget subsystem is notified and searches Widgets subsystem nodes for Widget match information. If it finds this connector, the widget can be automatically activated. Widgets subsystem sets Widget activation request and reads back Widget match information. Automatic 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 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.

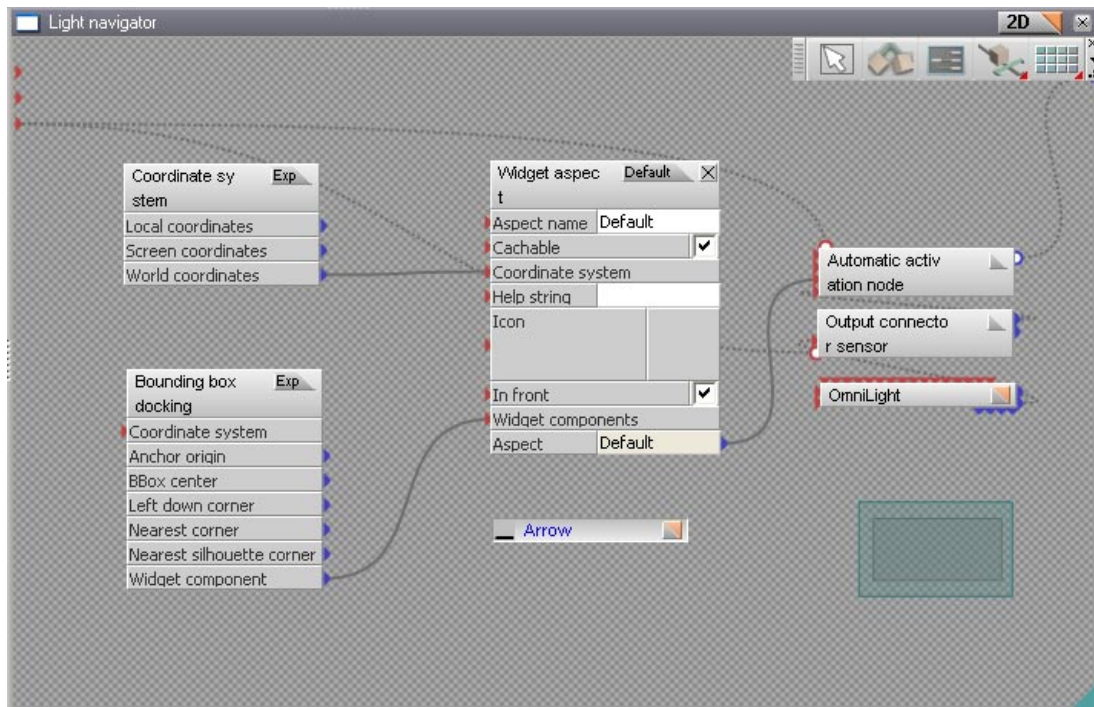


From this point, every time you select a light, Light navigator should be activated. This can be verified by selecting light and checking [Desktop]/Widgets/ Active Widgets Layer2/Active Widget,1. As our widget contains only two active nodes (coordinate system and docking node), active widget will contain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

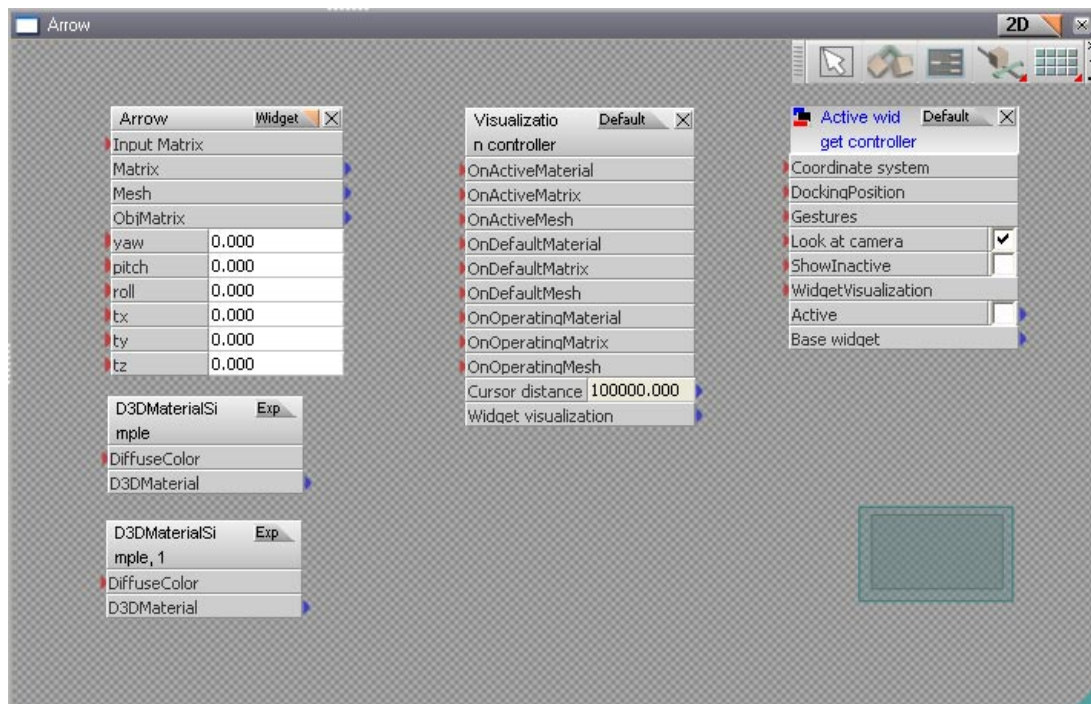


Now, we start to add functionality to our widget. First, let's create a move arrow. After we do this, we will copy it three times to get three movement axes.

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

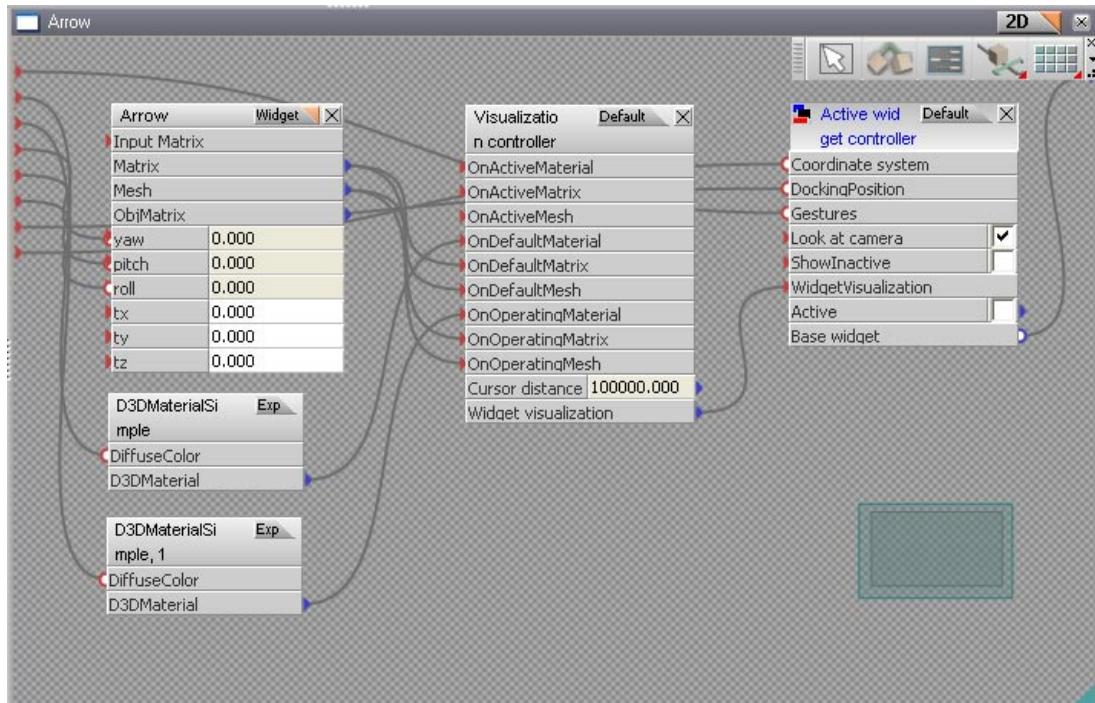


As you have read before, every base widget consists of shape, gesture and action. We add Mesh, Visualization controller, D3D material and Active widget controller nodes into the Arrow encapsulator. As a mesh, you can use a shape you have modeled in tS, but for this example, we will use the Arrow from 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.

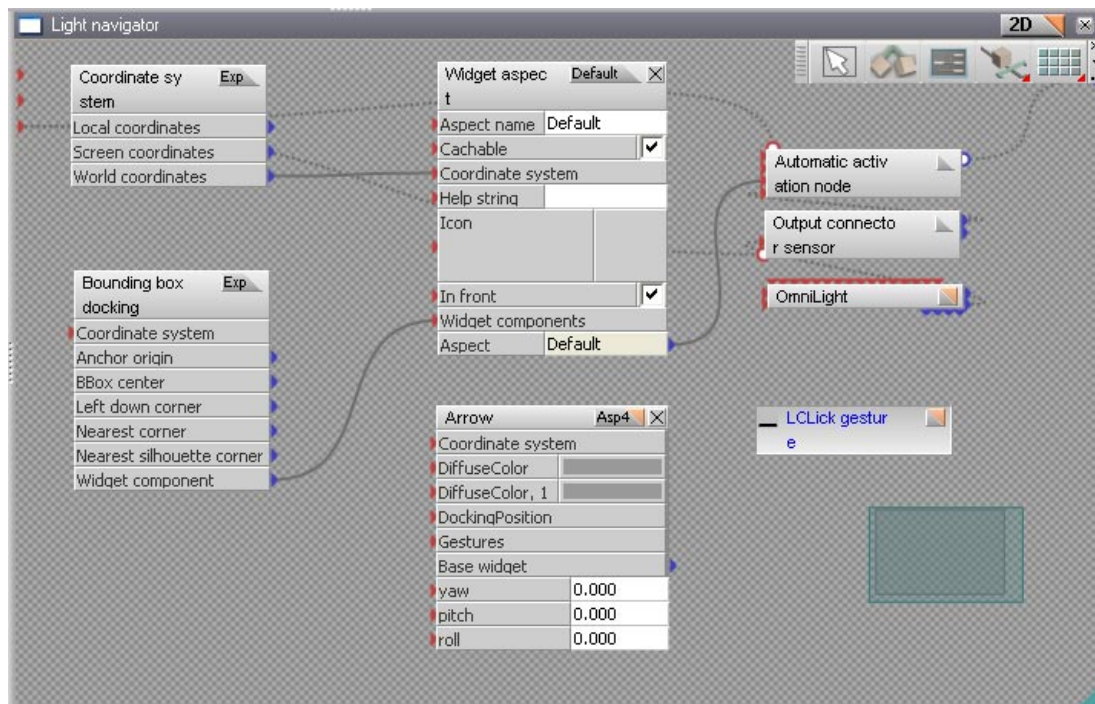


First, we need to export Coordinate system, Docking position, Base widget and Gestures from Base widget action controller. Then, since we want to specify axes from outside of the node, we can export yaw, pitch and roll from Arrow. Also export DiffuseColor from both D3D material nodes.

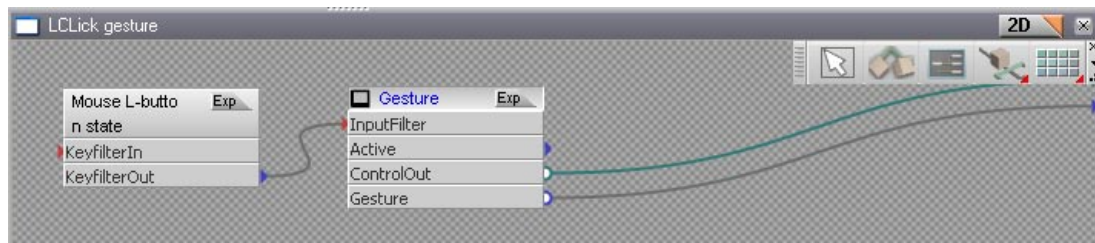
Now we need to connect connectors to create the widget arrow. So, connect Widget visualization from Visualization controller to WidgetVisualization input of Active widget controller. Connect Arrow's mesh to Visualization controller's OnDefaultMesh and OnOperatingMesh. Connect Matrix to OnDefaultTransform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



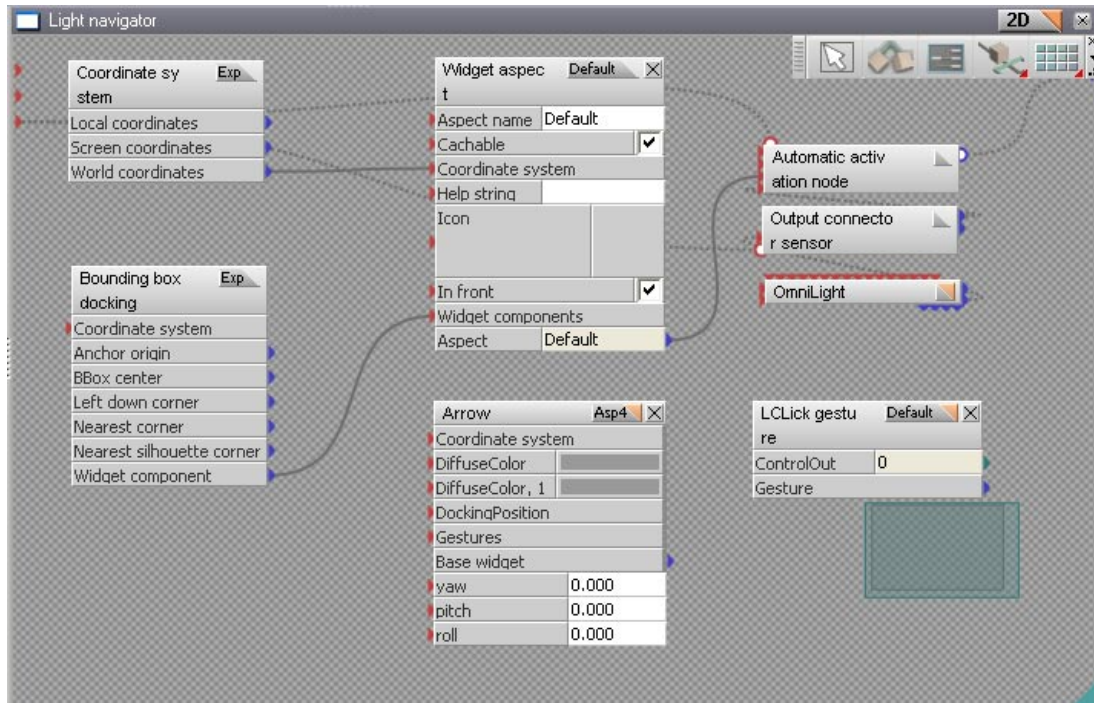
We have just created active arrow. Somehow Rosetta ignores exported yaw, pitch, roll connectors. We need to enter panel editor mode and add these connectors manually. We now need to create gesture and add action to it. Place Object from System/Kernel library to Light navigator widget and rename it to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



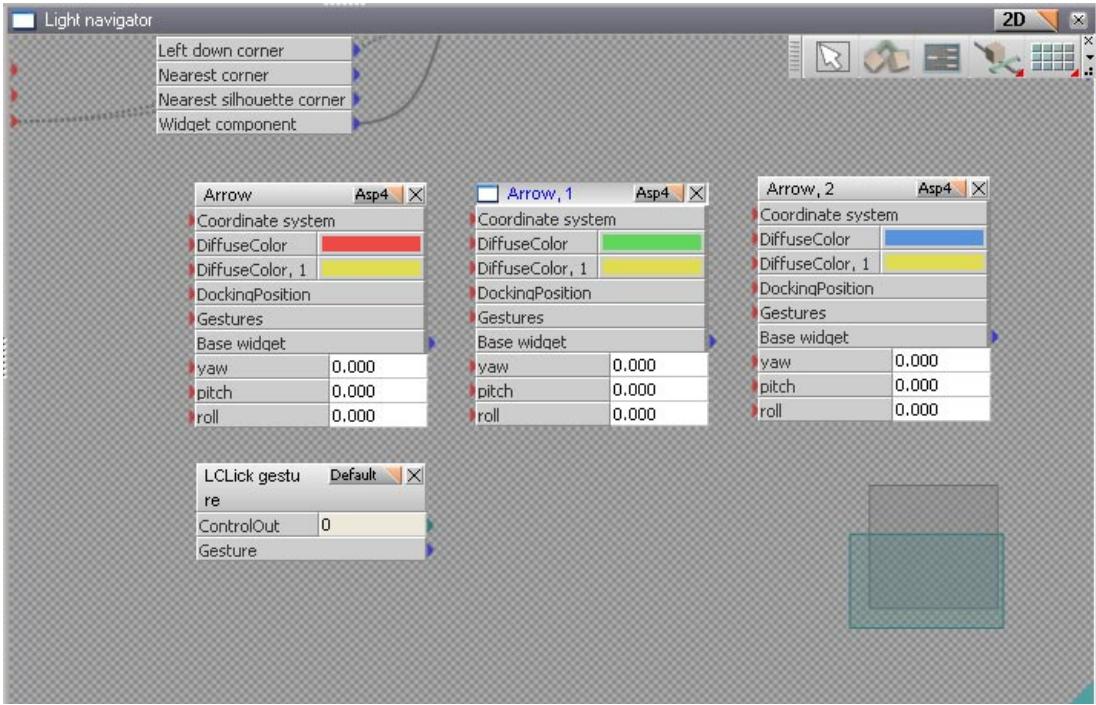
Open it, and place Gesture and Mouse L-button state nodes. Switch nodes to default aspect.



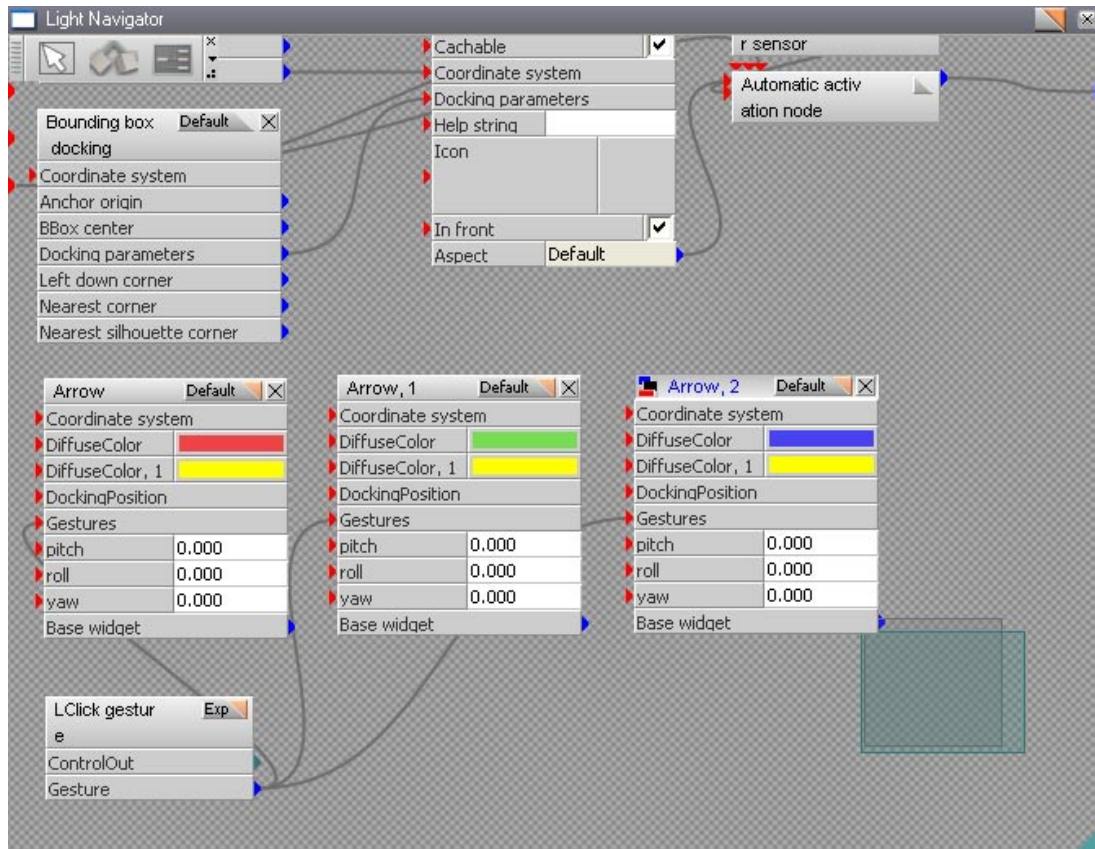
Gestures are specified by Gesture controller and various key/button filters. By connecting KeyfilterOut to InputFilter of Gesture, we limit activation of widget tools (actions) when left mouse button is not pressed. After the participant clicks with left mouse button on base widget's arrow, gesture fires 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.

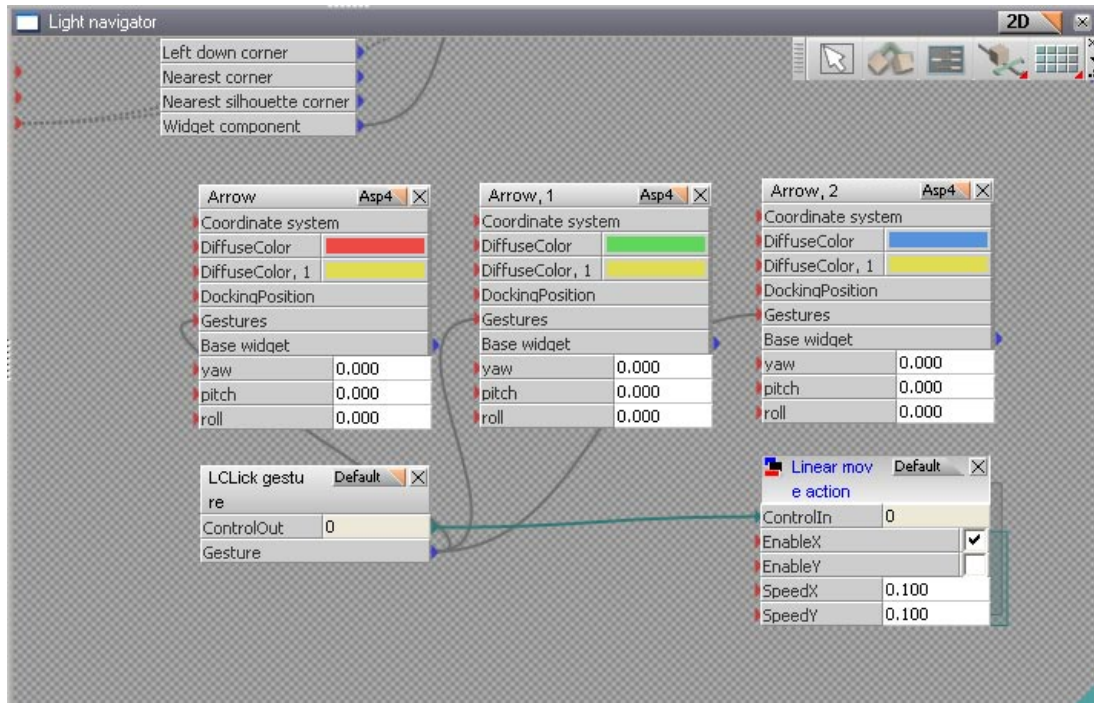


Now, we connect Gesture from LClick gesture to Arrow’s gestures inputs. We need not copy LClick gesture three times, because geometry information is transferred through connectors.



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.

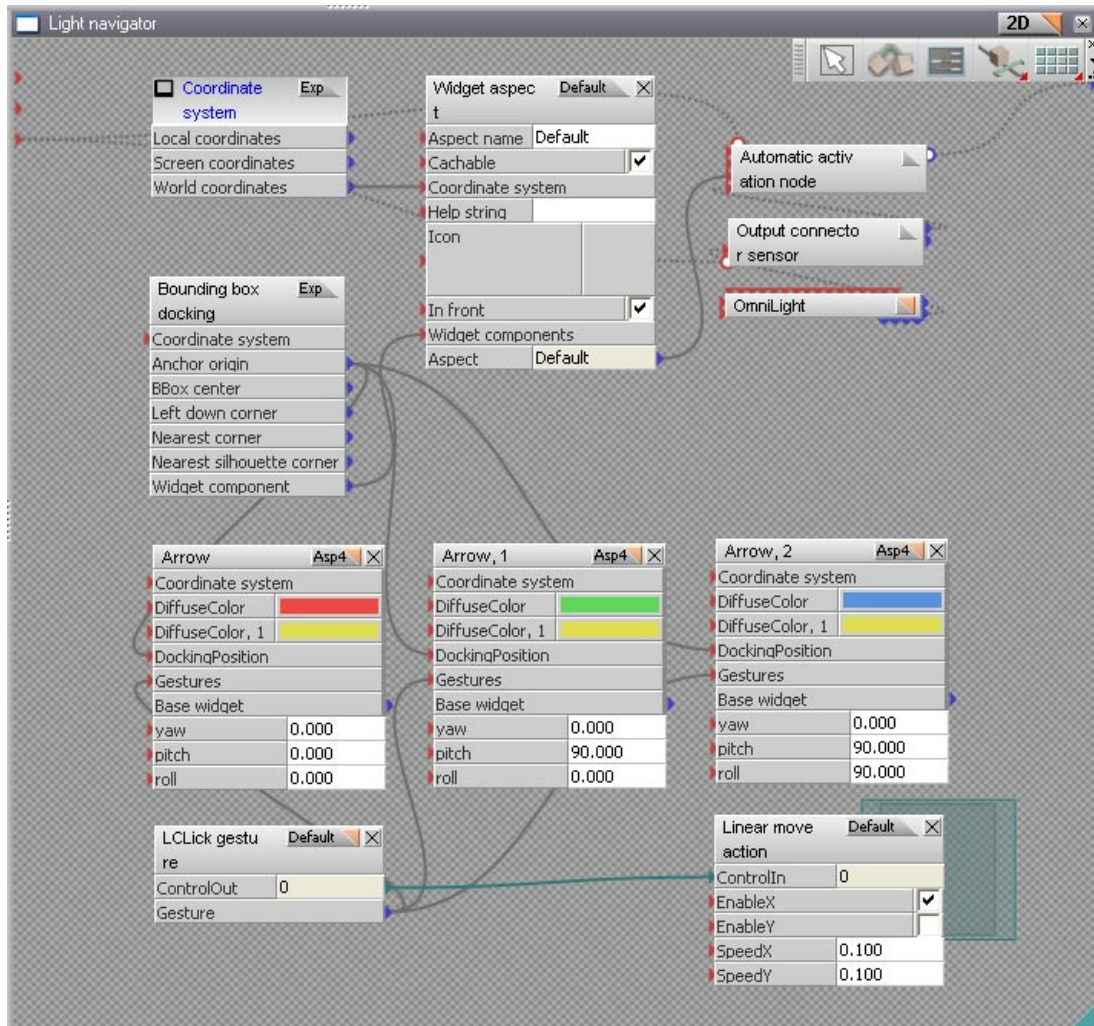
You can check both move actions and select the one you prefer. In this example, we use **Move action**. We connect ControlOut of LClick gesture to ControlIn of Move action.



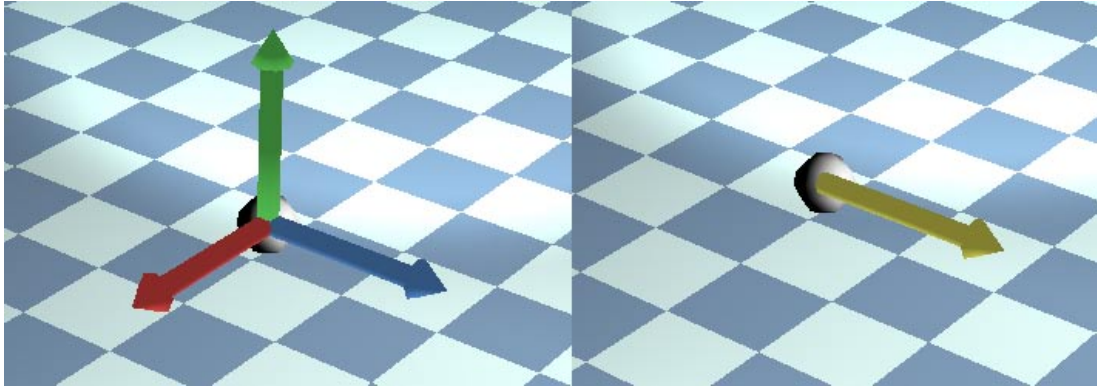
As you can see, Move action node has EnableX and EnableY checkboxes. These lock movements to the selected axes. Move action uses the shape's local matrix (Euler transform stored in Arrow/Arrow) and moves objects in Arrow's X direction. By enabling X and Y directions, move will work in 2D. (Try it!).

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:

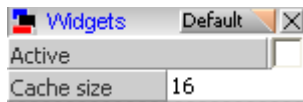


You can store this widget in a library the same way as you store shapes or materials. When someone gives you a library with widgets, you use them by dropping a widget into widgets subsystem node.

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

Description of widget package nodes

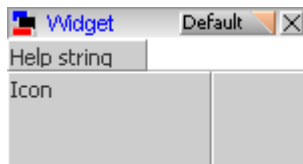
Widgets



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.

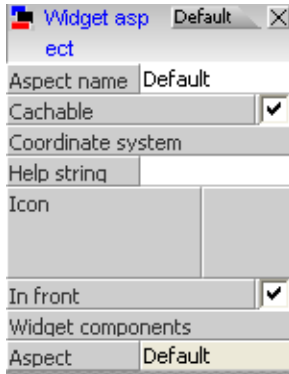
Widget



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.

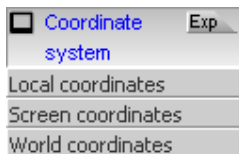
Help string will be used for displaying tool tips, and Icon will be used in change aspect menu. In front checkbox specifies whether widget should be rendered in front of all geometry, or should be mixed into the scene. Cacheable checkbox specifies whether widget aspect is cacheable or not.

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/Prefereces/Modeling preferences – it supports World, Local and Screen coordinates.

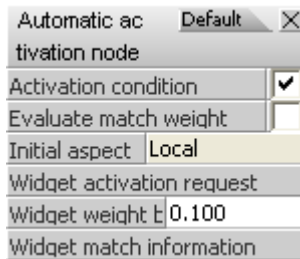
Persp. comp. preferences cs

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.

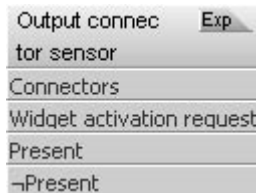
Input connector sensor



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

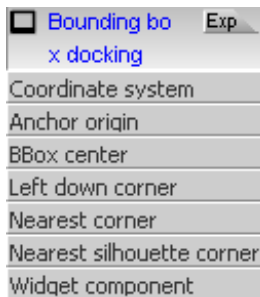


Output connector sensor is similar to **Input connector sensor**, but instead of input connectors, it scan for output connectors.

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).

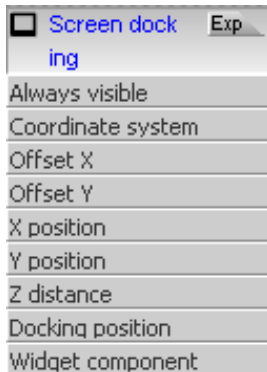
Offset docking



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



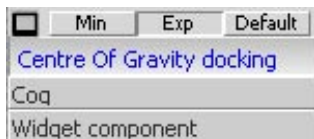
Node evaluates position of anchor node.

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

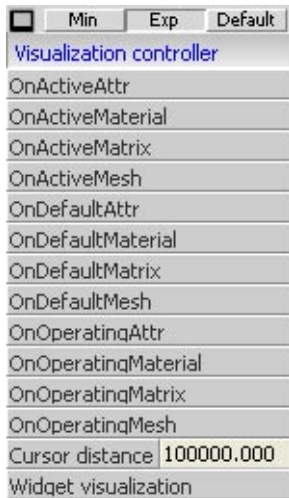


Node returns Center of gravity position

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



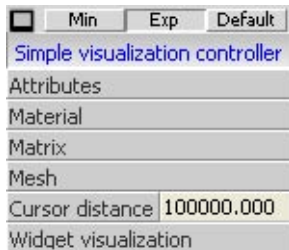
The Visualization controller selects widget mesh according to its three states:

- 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



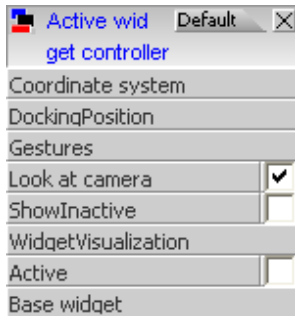
Functionality of this node is similar to Visualization controller, but it provides only one mesh, matrix, and material input. It was designed for optimization purposes.

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.

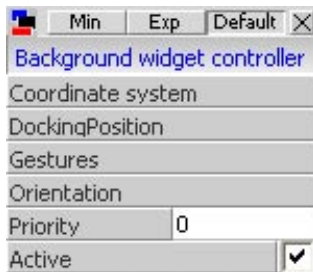
Active widget controller



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.

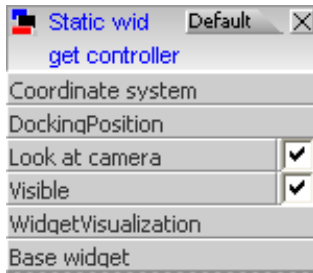
Background widget controller



Background widget controller handles management of background base widgets.

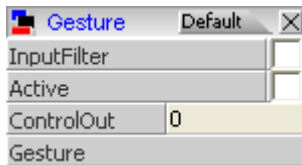
- **Priority** specifies tool background tool priority within tool management.
- **Orientation** defines coordinate system for action
- Other connectors are the same as for Active widget controller

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.

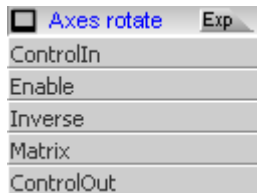
Close widget

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.

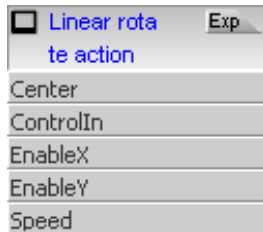
- **ControlIn** is the activation input.
- **Enable** enables matrix multiplication.
- **Inverse** selects between multiplication of inverse transformation or direct transformation.
- **Matrix** is the input for transformation matrix.
- **ControlOut** is the trigger for the following tool.

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.

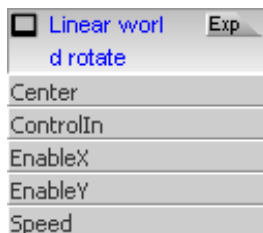
Linear rotate action



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.

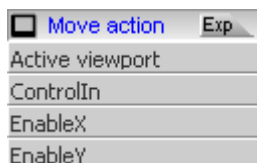
- **Center** defines the center of rotation.
- **ControlIn** is the activation input.
- **EnableX**, **EnableY** enable or disable rotation around X and Y axes.
- **Speed** specifies the ratio between mouse step and angle step. 0.5 means 0.5 degrees to one mouse step.

Linear world rotate



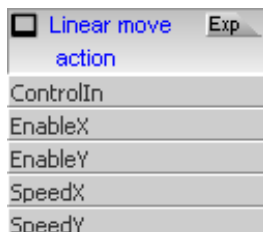
Linear world rotate performs a similar action as the node above, but instead of rotating in widget space it always performs rotation in world coordinate space.

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



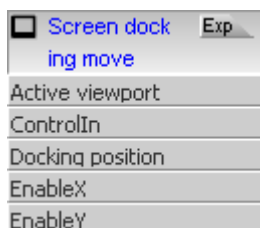
Linear move action performs a similar operation as the Move action, but the difference is in movement step calculation. Slider type move action does not count perspective into movement step evaluation. Movement speed can be altered by the Speed input.

Widget move



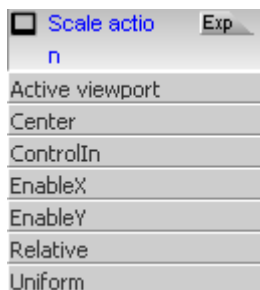
Widget move moves widget from its original position. Movement is performed in the widget shape's XY space, where movement along X and Y axes can be enabled or disabled with the EnableX and EnableY inputs.

Screen docking move



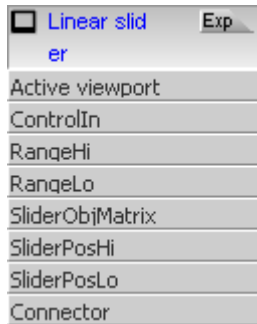
Screen docking move changes relative position of the widget on the screen **docking**, therefore this node can be only used with screen docking node. Action is linked to docking node by the Docking position input. Movement can be enabled or disabled for X and Y axes.

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.

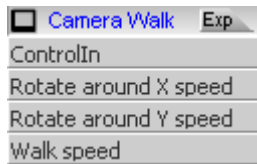
Linear slider



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



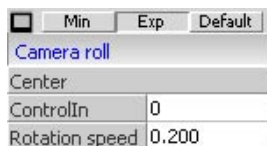
Camera navigate uses the mouse X coordinate to rotate controlled objects around the global Y axis, and the Y coordinate to perform movement in local Z axis. Rotation and Movement speed can be defined by the speed inputs.

Camera FOV



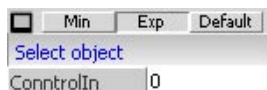
Camera FOV modifies field of view for projective cameras and zoom factor for orthogonal views.

Camera Roll



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

Select object



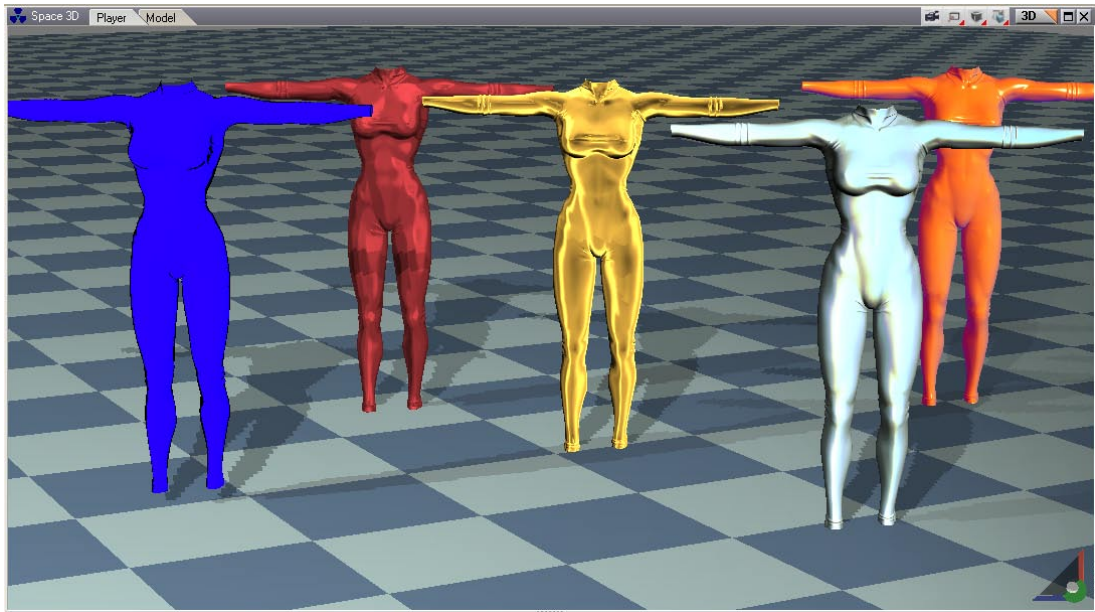
Select object Command raycasts scene and selects nearest object under mouse cursor

Chapter 3

Materials

3.1 Advanced Materials in trueSpace

The basic material libraries included with trueSpace offer a broad range of options for surfacing your 3D objects, but there will be times when you want to go even further. You can do this with the low-level material construction facilities provided by trueSpace. Using the Material Editor and shader scripts you can create your own advanced materials, with a wide range of possible results.



A selection of advanced DirectX materials shown in the Player window

Building Advanced Materials

There are two primary methods for constructing advanced materials in trueSpace. The first is by combining shader and material component objects in the Material Editor, a special aspect of the Link Editor, and the second is using HLSL or LUA to create scripted shaders.

You can use the Material Editor to create a material from the components provided in the various libraries. Often you can simply modify an existing material to achieve the effect you want. Other times you will want to start from scratch, connecting components in the Material Editor to build up your desired effect. For more information on using the trueSpace Material Editor for creating advanced materials see the next section.

You can also use scripting to create scripted shaders for your objects. Although creating shader scripts is an involved process it is also extremely powerful, giving you complete and explicit control over every aspect of your shader's appearance. If you want to write DirectX shaders then you should read "Writing DirectX Shader Scripts". To learn more about writing shader scripts for Super LightWorks using LUA see the section entitled "Writing Super LightWorks Material Scripts".

trueSpace Material Types

trueSpace offers several different material types and deciding which to use for a given task will depend on your goals. Available materials types are:

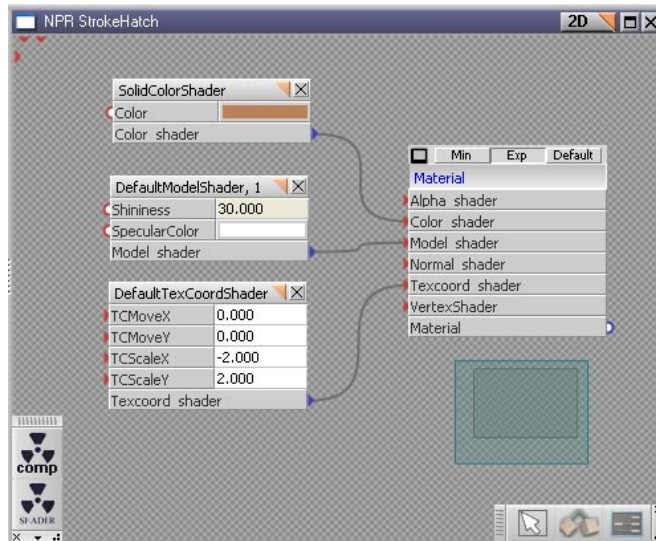
- **DirectX:** For a material that provides real-time effects in the Player view, such as per-pixel or per-vertex operations, you will want to use a DirectX material on your model.
- **LightWorks and Super LightWorks:** If you are using the LightWorks renderer you will want to take full advantage of the features available using a LightWorks or Super LightWorks material.

These different material types and their features are discussed in detail in the following sections. See "Using the Material Editor" below for more information on building these materials with the trueSpace Material Editor. See "Writing Shader Scripts" for more details on creating shader scripts for DirectX and Super LightWorks materials.

DirectX Materials

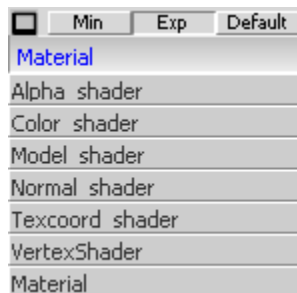
DirectX Materials are real-time materials, meaning that certain components of these materials may be processed on your graphics card on an ongoing basis to create special rendering effects. While a DirectX material can be as simple as a solid color or bitmapped texture it can also include more complex, calculated tasks such as per-pixel coloration or per-vertex distortion.

A DirectX material may have several components. The following image shows the components inside the NPR StrokeHatch material, found in the Materials library under Shaders.



Inside of the NPR StrokeHatch material

There are a number of objects here, the most important of which is the Material object. This object essentially serves as a material ‘compiler’, taking the outputs of all shader components and combining them to create the end result. You can see that the Material attribute of this object is exported, giving trueSpace access to the final material for rendering in the player window.



The Material object acts as a material ‘compiler’

Shader Components

The Material object includes input connectors for six separate shader components. These are:

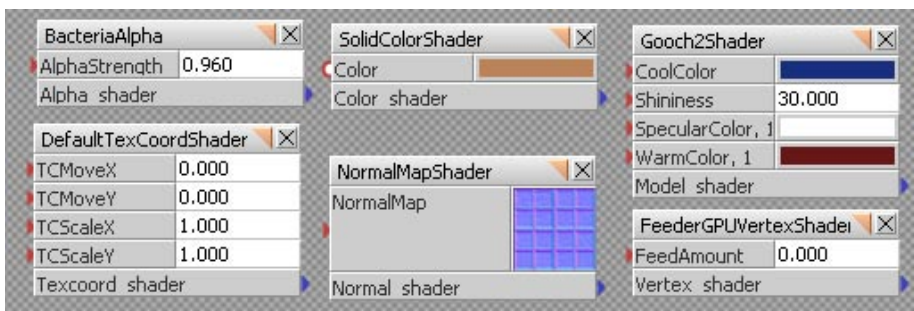
- **Alpha Shader:** Information on transparency in the material. This can be defined by using a masking bitmap or through another means.
- **Color Shader:** Coloration data for the material. Like alpha, this information can be provided from a bitmap, by specifying a solid color, or via custom shader components.
- **Model Shader:** The overall ‘shading model’ to be used for the material. The Model Shader

specifies how light, shadow, specularity, reflectivity, and other components will interact to create the final, rendered look of the material on the object.

- **Normal Shader:** Specifies how the surface normals of the model will be used to affect the appearance of the rendered image. For instance, a bitmap could be used to alter surface normals to make the surface appear rough or to feature an embossed design.
- **Texcoord Shader:** Determines how the model's texture coordinates will be modified in rendering the final material. They can be stretched, offset, or completely altered by custom routines.
- **Vertex Shader:** DirectX materials can include a special shader that procedurally modifies the vertices of the model itself each frame. You can use this to create special effects such as displacement mapping or expansion and contraction.

There are many available pre-built shader components located in the libraries that appear when you enter a material in the Material Editor - or you can build your own shader components from the various shader objects, also found in these libraries. Advanced material designers can explore creating DirectX shader scripts to extend the possibilities even further.

A collection of shader components is shown in the following image.



Stock shader components for alpha, color, model, texture coordinates, normals, and vertex

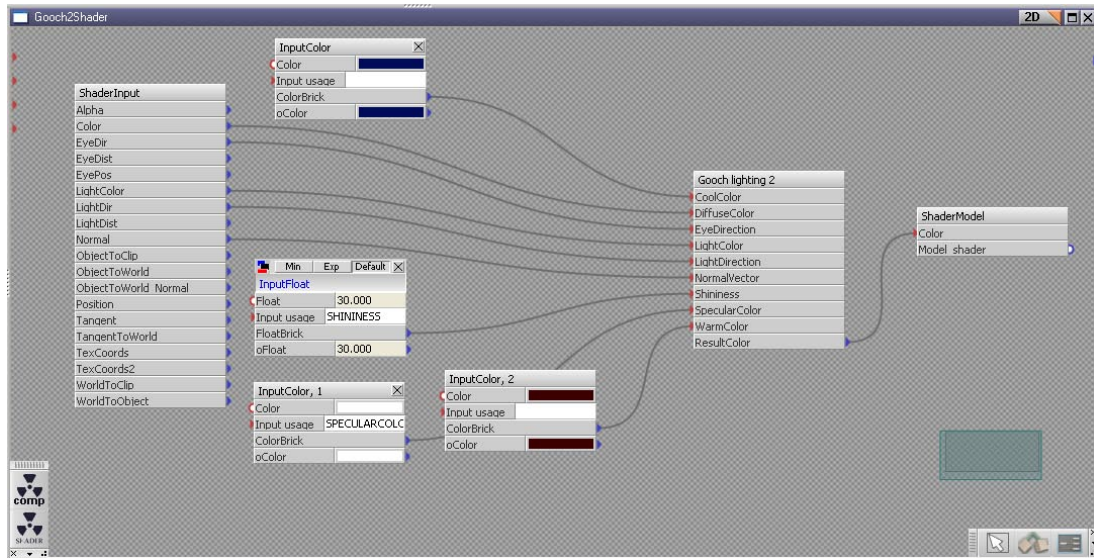
Note that the output for all of the shader components above matches one of the shader inputs on the Material object. So, for instance, to connect the SolidColorShader object to your material, simply join the Color Shader output on that object to the Color Shader input on your Material object. Other shader components can be connected in the same manner.

Custom Shader Components

Often you will want to create an effect that requires a custom shader component. Building a custom shader component in the Material Editor is similar to creating an activity or other object in the Link Editor. First, think about your effect and how you will build it, collect the necessary components from the Material Editor library, then connect them and test the result by connecting the final output to the appropriate shader component input on the Material object.

One such example is the Gooch2Shader, found in the Materials library under Shaders. Drag this material onto an object in the Player view and then enter the object, then enter the Gooch2 object in the Link Edi-

tor. This will take you to the Material Editor view. Finally, enter the Gooch2Shader shader component and you should see the collection of objects shown below.



Inside a custom Model Shader we see the components that create the effect

While this looks like a complex assembly bear in mind that the only function of this component is to take into account lighting and surface information to tell trueSpace how to render the material. If you look to the far right of the image above you will see a ShaderModel object, with an exported Model Shader output attribute. This is, like Material, a sort of compiler - taking your finished computed color and converting it to the proper format for use with the Material object.

Gooch Lighting 2 is an object that takes in a number of details about the model, lighting, color, and other surface qualities, and computes a final color for the pixel being rendered based on a Gooch lighting algorithm. Because it is a compiled object we can't see what is going on inside it but information on the Gooch lighting model is available in any good computer graphics reference if you are interested. You can see that the object's input connectors are attached to several input color objects and a number input object.

Shader Component Inputs

Many of Gooch Lighting 2's inputs are connected to an object called ShaderInput, shown by itself below. This object is extremely important because its job is to provide information about the camera, lighting, the model and its surface, and more to your shader components on a per-pixel basic. That is, as your scene is being rendered, this object will constantly update itself - changing its Normal output, for instance, to match the normal vector of the model's surface at the point currently being rendered.

ShaderInput
Alpha
Color
EyeDir
EyeDist
EyePos
LightColor
LightDir
LightDist
Normal
ObjectToClip
ObjectToWorld
ObjectToWorld Normal
Position
Tangent
TangentToWorld
TexCoords
TexCoords2
WorldToClip
WorldToObject

The ShaderInput object provides information to your shader components

As you can see there is a lot of information available via the ShaderInput object - a total of nineteen separate outputs. Some of these, such as Normal and Color, you will use quite frequently to power your custom shaders. Others, such as WorldToClip may be rarely used. A brief description of these outputs follows:

- **Alpha** - The transparency of the current point based on maps and other factors.
- **Color** - The color of the current point based on texture and other color contributors.
- **EyeDir** - Provides a world space vector to the camera from the current point.
- **EyeDist** - The current point's distance from the camera.
- **EyePos** - The location of the camera in world space.
- **LightColor** - The color of the light affecting the current point.
- **LightDir** - The world space vector to the light from the current point.
- **LightDist** - The distance from the light to the current point.
- **Normal** - Provides the normalized world space normal of the current point.
- **ObjectToClip** - Transform from local object space to screen space. Perspective projection - if you want the real screen coordinates then divide by the W component.
- **ObjectToWorld** - Transform from object to world space.
- **ObjectToWorld Normal** - Transform from object to world space - use with normals and tangents.
- **Position** - Provides the position of the current point in world space.
- **Tangent** - The normalized world space tangent of the current point.
- **TangentToWorld** - Transform from tangent to world space.

- **TexCoords** - The current point's texture coordinates in UV space.
- **TexCoords2** - Coordinates from the second UV set.
- **WorldToClip** - Transform from world to screen space. Perspective projection - if you want the real screen coordinates then divide by the W component.
- **WorldToObject** - Transform from world to object space.

Material Compatibility

In order for your materials to render correctly on a wide range of hardware the trueSpace Direct3D pipeline provides a compatibility rendering mode. In order for materials to render on any generation of hardware you must specify “usage” information for some of the input objects (i.e. InputBitmap, InputFloat, ...) used in your shader components.

The Direct3D pipeline is then able to use such input objects for rendering the simple Phong material with similar appearance, even if the original complex shader might not be available on your generation of graphics hardware. You can write the usage string into the “Usage” attribute of each shader input object.

Usage string	Description	Applicable input objects
DIFFUSEMAP	Diffuse texture	InputBitmap, InputBitmap1D
DIFFUSECOLOR	Constant diffuse color	InputColor*
DIFFUSEMAPSTRENGTH	Diffuse map strength	InputFloat*
VERTEXCOLORSTRENGTH	Vertex color strength	InputFloat*
NORMALMAP	Normal map	InputBitmap, InputBitmap1D
SPECULARCOLOR	Specular color	InputColor*
SHININESS	Shininess	InputFloat*
SPECULARSTRENGTH	Strength of specular reflections	InputFloat*
DIFFUSESTRENGTH	Strength of diffuse reflections	InputFloat*
C_TCSCALEX	Diffuse texture scale in X direction	InputFloat*
C_TCSCALEY	Diffuse texture scale in Y direction	InputFloat*
C_TCMOVEX	Diffuse texture move in X direction	InputFloat*
C_TCMOVEY	Diffuse texture move in Y direction	InputFloat*
N_TCSCALEX	Normal map scale in X direction	InputFloat*
N_TCSCALEY	Normal map scale in Y direction	InputFloat*
N_TCMOVEX	Normal map move in X direction	InputFloat*
N_TCMOVEY	Normal map move in Y direction	InputFloat*

*Optimal component, but any of these types can be used: InputFloat, InputPoint, InputColor, InputMatrix and the optimal conversion is done internally.

LightWorks Materials

{information on LightWorks materials from Pavol}

3.2 Using the Material Editor

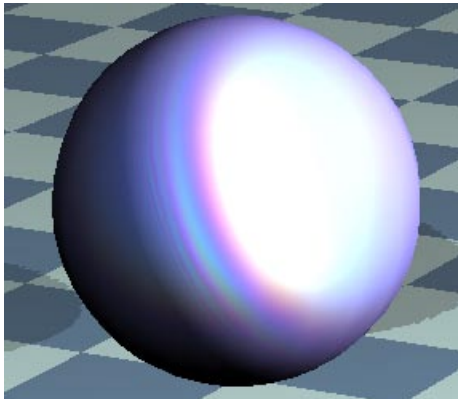
When you create advanced materials in trueSpace you will work primarily within the Material Editor, a special view of the Link Editor with a selection of libraries devoted specially to material creation. A material object is visually similar to any other object you might encounter in the Link Editor view with the exception that it exports a Material attribute - which trueSpace uses to know how to render objects that are painted with that material.

DirectX Materials in the Material Editor

The trueSpace Player view fully renders DirectX compatible materials. This section looks at these materials, their components, and how to create them in more detail. Also, be sure to check out “Writing DirectX Shader Scripts” for information on using HLSL to define DirectX shader components.

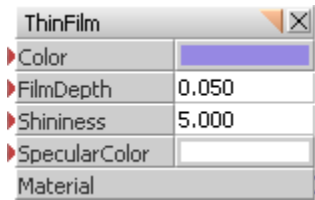
Examining a DirectX Material

To look at the internal details of a material in the Material Editor, locate the Sphere object in the Objects library under the Base panel. Drag it into the Link Editor and then find the ThinFilm material, located in the Materials library under Shaders, and drag that onto the sphere in the Player view. Your sphere should now be colored with a bluish, shiny material, as shown in the image below.



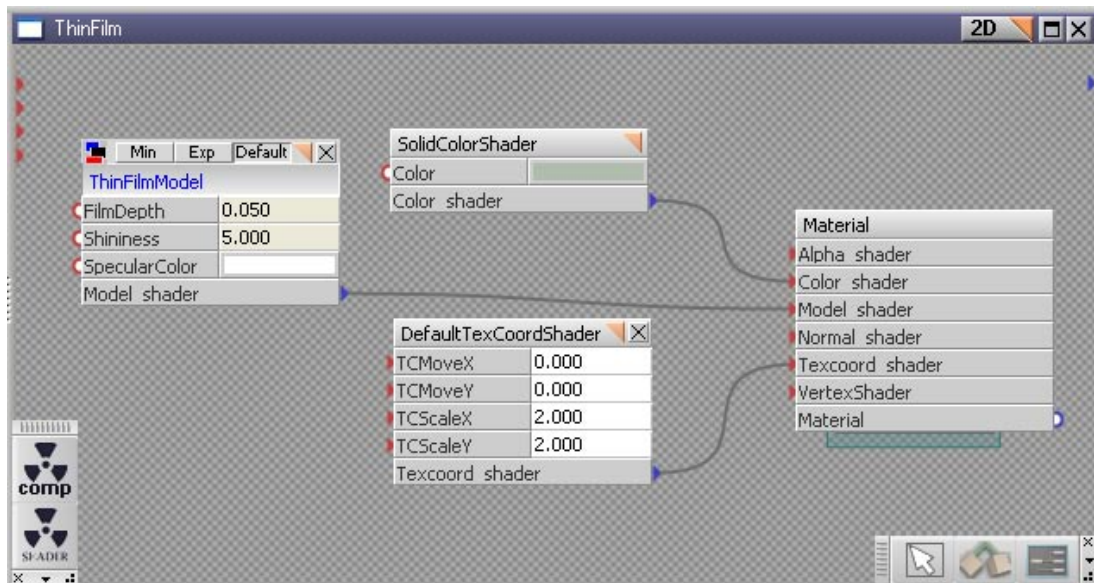
The ThinFilm material applied to a sphere in the Player view

Enter the Sphere object by clicking the orange enter icon. Once inside you should see a material object called ThinFilm, shown in the image below. This is the material we just applied. To enter the material, starting the Material Editor, simply click the orange enter icon on the material object's front panel.



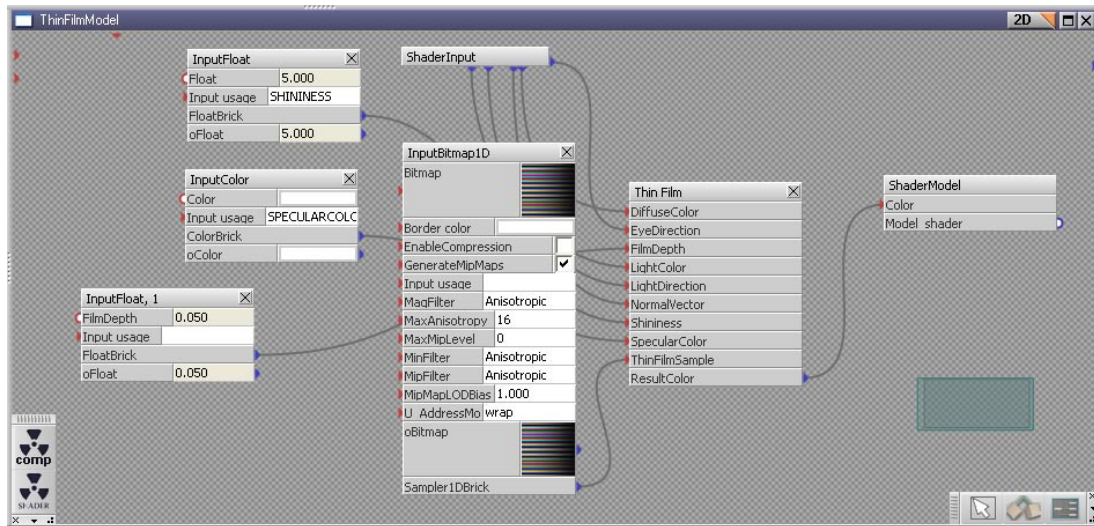
The ThinFilm material object in the Link Editor

Once inside the Material Editor you will see the parts that work together to create the material's effects, as shown in the image below. This material contains a Material object (see DirectX section above for more information) and several shader components that feed into it: a SolidColorShader, a DefaultTexCoordShader, and finally a ThinFilmModel model shader.



Inside ThinFilm we find the shader components that create the effect

As in the Link Editor, you can enter any object that displays an orange triangular icon in the upper right-hand corner of its panel. For instance, click on the ThinFilmModel shader component to see its contents. You should see something like the image below.



Another level of detail resides inside the ThinFilmModel shader component

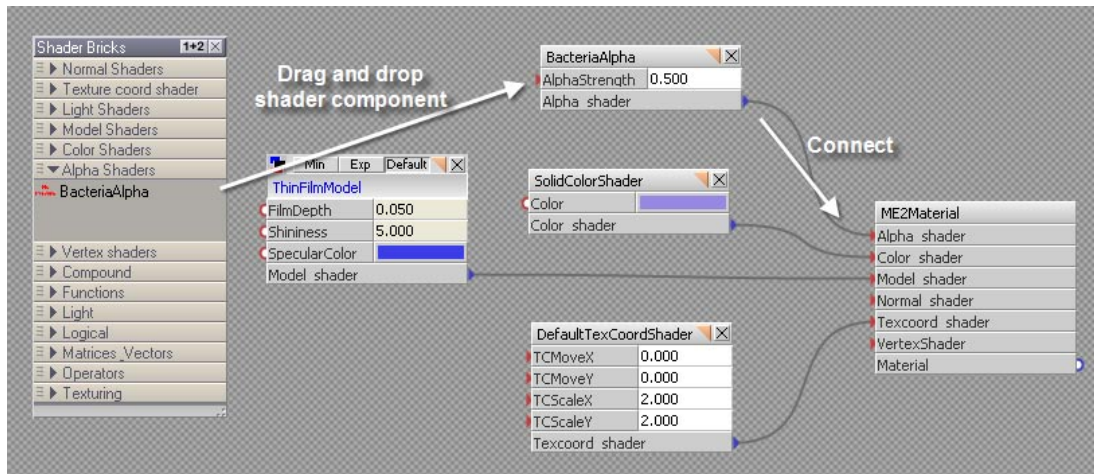
The main parts of this shader component are the Thin Film object, which takes lighting and surface information to determine the final output, the ShaderModel object, which exports that result for use with the Material object, and the ShaderInput object, which provides the shader component with data about lighting and other conditions.

The Thin Film object takes in two floating point numbers, provided by the InputFloat objects, to control Shininess and FilmDepth. It also takes a color, provided by InputColor, to use as the specular color of the material. Finally it takes the input of an InputBitmap1D, which is a one-dimensional bitmap with alternating light and dark bands. This bitmap is sampled to provide additional shading information for the material - resulting in the circular ‘oily puddle’ effect you can see on the sphere.

The ThinFilm object also gathers data from the ShaderInput object to make its calculations. Inputs taken from this object are Color, EyeDir (because the effect is view-dependent), LightColor and LightDir (to determine the contribution from lights in the scene), and Normal (because the effect depends on the surface normal - i.e. the shape of the surface - of the model itself).

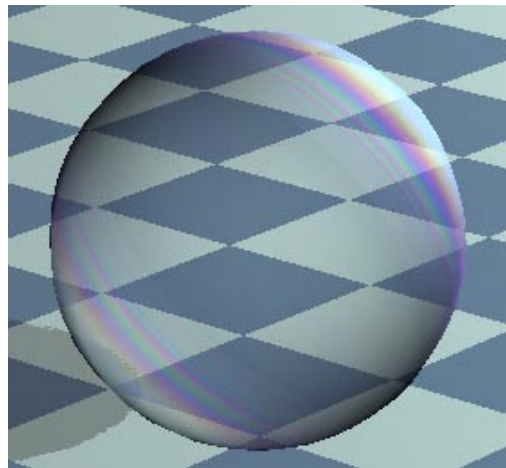
Modifying a DirectX Material

You can, of course, modify any of the connections or objects you find in a material using the Material Editor. For instance, to add an alpha component, exit the ThinFilmModel shader component one level so that you can see the Material object. Now, drag in a BacteriaAlpha shader component (see below) from the Shader library and connect its Alpha Shader output to the Alpha Shader input connector on the Material object. Try typing in different values for AlphaStrength (0.5 in this example).



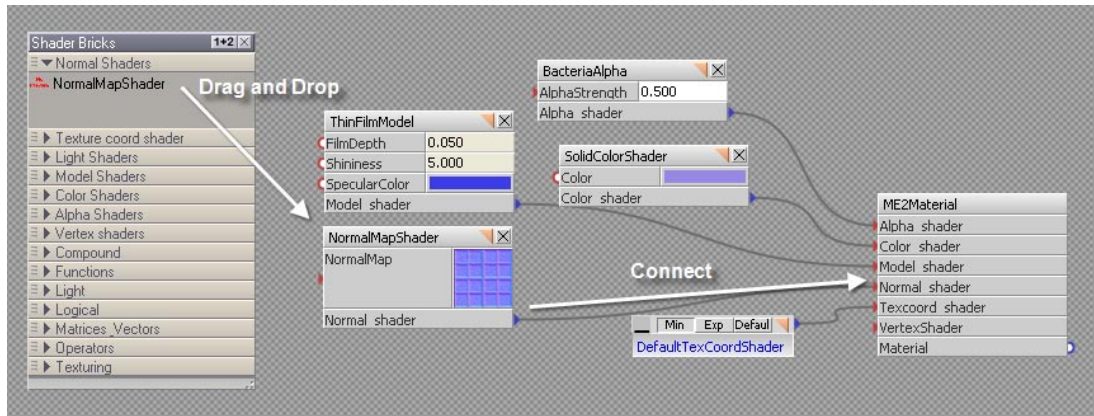
Adding an alpha shader component to the ThinFilm material

When you have made these changes take a look at the Player view. Your sphere should look something like the one in the image below. By adding an alpha shader component we have altered the look of the material by introducing transparency.



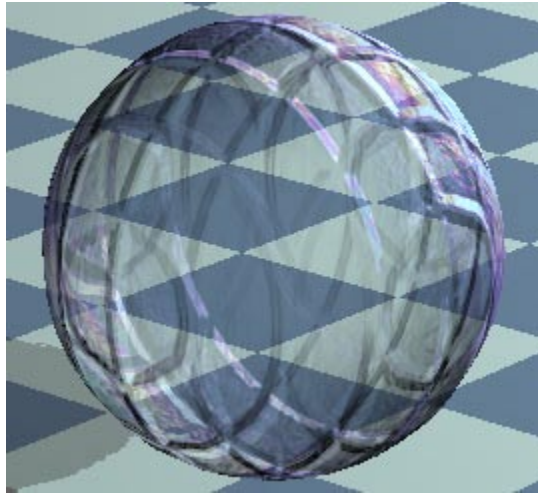
The Player view shows the sphere with transparency added

You can use the same procedure to add a 'bumpiness' effect to the material. Try dragging in a NormalMapShader component from the Shader library and connect its Normal Shader output to the Normal Shader input connector on the Material object (see below). You can change the bitmap used for the NormalMap if you desire, or just keep the default grid bitmap.



Adding a normal shader component to the ThinFilm material

When you have made these changes take another look at the Player view. Your sphere should look like the one in the image below, unless you changed the default normal map. By adding a normal shader component we have introduced bumpiness to the material, giving the sphere an "embossed" look.



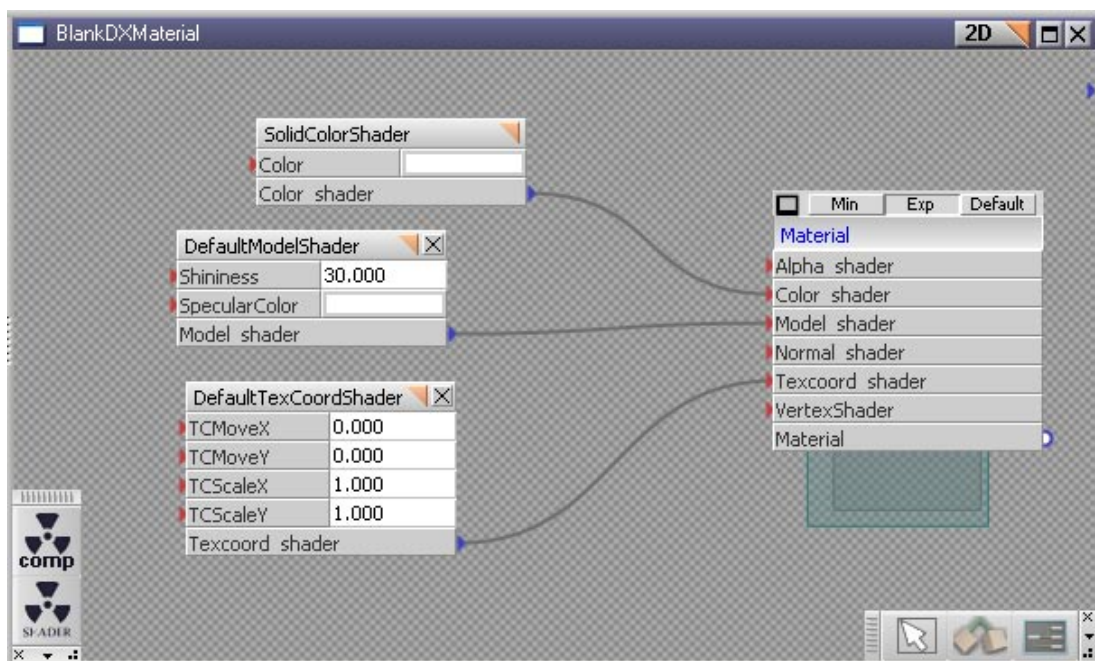
The Player view shows the sphere with a normal map added

Often, when you are designing a material you can simply use an existing material from the library and modify it to create a desired effect. You can delete or modify existing shader components as desired or create new ones using the various components found in the library. As an exercise, before attempting to construct a material from scratch, browse through the Material library and examine how the materials are built and how their various attributes are used to determine the final look of a material.

DirectX Material Creation

When you want to go beyond using and modifying existing materials you will want to try creating your own materials from the components found in the Shaders library. To get started creating your own material, first drag a Sphere object from the Base panel of the Objects library into the Link Editor. Then locate the BlankDXMaterial in the Material library and drag it onto the sphere in the 3D view.

This material template can be used as a base on which to build your desired material. If you enter the material you will see that it contains a SolidColorShader with a white color selected, a DefaultModelShader, and a DefaultTexCoordShaders. You can, of course, replace these with your own shader components and add an alpha shader, normal shader, and a vertex shader if desired. Simply connect these to the inputs of Material to add them to your material.



The contents of BlankDXMaterial in the Material Editor

While exploring the creation of your own materials you might want to examine the following list of commonly used objects (found in the context-sensitive Shader library).

- **ShaderInput:** The shader input object is used to bring data for each pixel from the rendering engine to the shader. Use it to obtain information such as surface normal for calculations that will drive the appearance of your material.
- **Component Input Objects:** Several input objects are available to provide data to your shader components. These include InputBitmap and InputBitmap1D, InputFloat, InputColor, InputMatrix, and InputPoint.

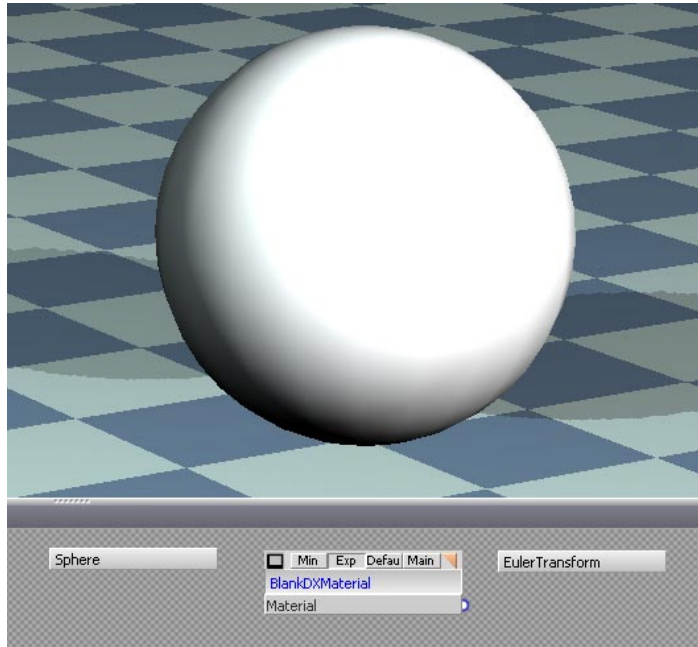
- **Shader Component Compilers:** For each shader component type, there is a special object that compiles input data into output data of the appropriate type for input to the Material object. These objects are ShaderAlpha, ShaderColor, ShaderNormal, ShaderModel, ShaderVertex, and ShaderTexCoord.
- **General Functions and Operators:** There are a number of objects for performing calculations within shaders. These include matrix and vector operators like Dot-product and Cross-product, logical operators such as Not and Any, and mathematical functions such as Arcsine and Cosine.
- **Other Shader Objects:** The Shader library also includes a number of additional components that can be used to construct materials. For instance the Tex2D object is used to extract a color from a bitmap using a set of 2-dimensional texture coordinates.

You should also bear in mind the purpose of each of the various shader components. These are covered briefly in the following list.

- **Alpha Shader:** This is the shader component where you can specify any information about transparency in your material. You can do this simply using an alpha map or you can use a more complex algorithm of your own devising, such as making the material more transparent as the surface normal faces the view direction.
- **Color Shader:** This shader component is responsible for the basic surface coloration of your material. As with alpha shaders, you can simply apply a bitmap as a texture or use a solid color. Alternatively, you could for instance, try to implement a more complex blending routine that takes into account the surface normal of the underlying geometry (as in the terrain tutorial below).
- **Model Shader:** This is where most of the shading work for your material is handled and defines the interaction of the surface with lights and other factors. You can use one of the model shaders from the library or implement your own model. A computer graphics reference text that covers lighting models might come in handy if you plan to explore creating custom model shaders.
- **Normal Shader:** This component is used to describe the ‘smoothness’ or ‘bumpiness’ of a surface. You can apply a normal map, a special kind of bitmap that specifies normal direction at a given point on the surface or implement your own normal modification routine.
- **TexCoord Shader:** Changes made here affect how textures and other maps are applied to your model. Using just the DefaultTexCoordShader object you can change texture offset and scaling. You can also apply your own mapping models if you want to achieve special effects such as animated textures.
- **Vertex Shader:** This is a special shader component that alters the apparent position of your model’s vertices based on conditions you specify. For instance, you can use a vertex shader to ‘slide’ the vertices in and out along their normal vectors - creating an expanding and contracting surface like a soap bubble. Many unique effects can be achieved with vertex shaders.

Tutorial: Creating a Simple DirectX Material

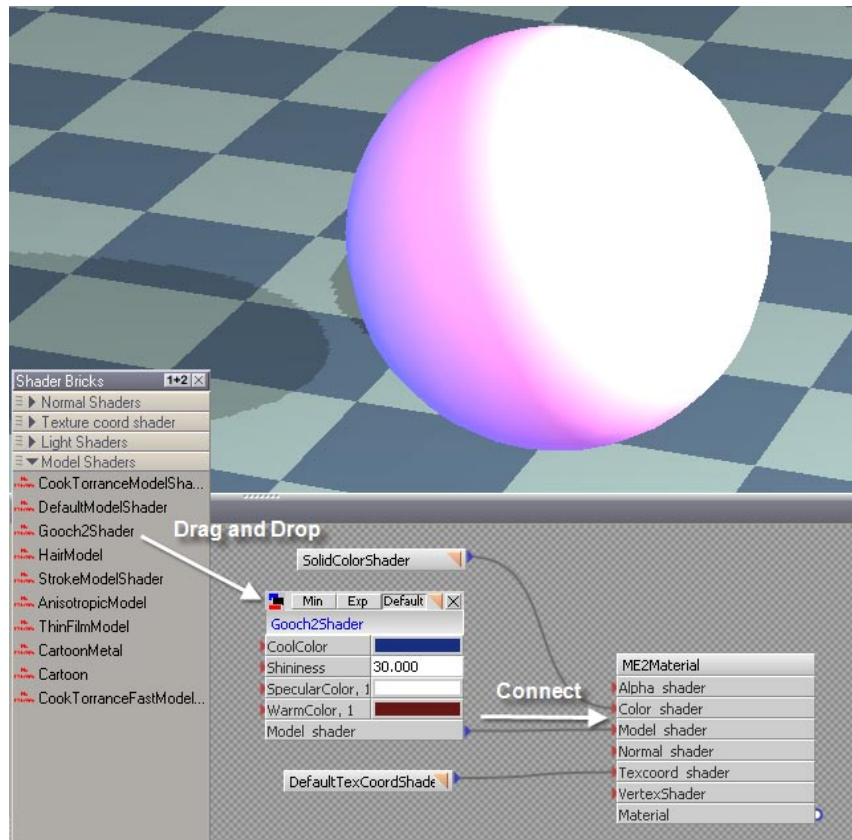
In this example we will create a simple DirectX material using the Material Editor using only stock shader components from the Shader library. To begin, drag a Sphere (Objects|Base library) into the Player view, and then drag the BlankDXMaterial onto the sphere in that view. Now, enter the Sphere object in the Link Editor. You should see something like the image below.



The BlankDXMaterial applied to a Sphere in the Player view

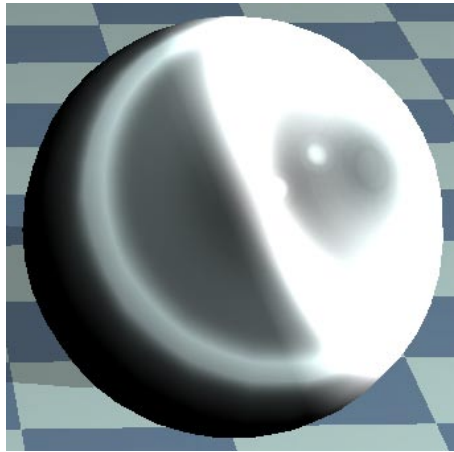
Now, enter the material. You will see the components mentioned previously; SolidColorShader, DefaultModelShader, and DefaultTexCoordShader. First, let's try changing the model shader to see what effects we can achieve quickly. A good way to show what a model shader really does is to take it away and watch the results. Break the link between the DefaultModelShader and the Material object. The sphere should now appear completely white in the Player view with no shading at all.

The model shader, defines how light interacts with the object and material to create the final appearance of your 3D object when rendered. The DefaultModelShader uses a Phong lighting model {is this correct?} which results in a look that can range from matte to glossy plastic. Let's try a different model shader. Open the Shader library and find the Model Shaders panel. Drag a Gooch2Shader into the Material Editor and then connect its Model Shader output to the Material object. You sphere should now be a bit more colorful, as in the image below.



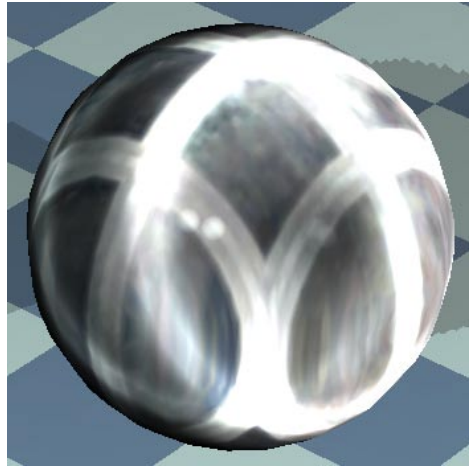
Replace the model shader with the Gooch2Shader

You can experiment with changing the various colors by double-clicking them. Try changing the Cool-Color to a greenish hue to see how it changes the look of the sphere in the player window. Now delete the Gooch2Shader and drag in the AnisotropicModel object from the library and connect it to the Material object. Your sphere should change appearance again, as in the image below.



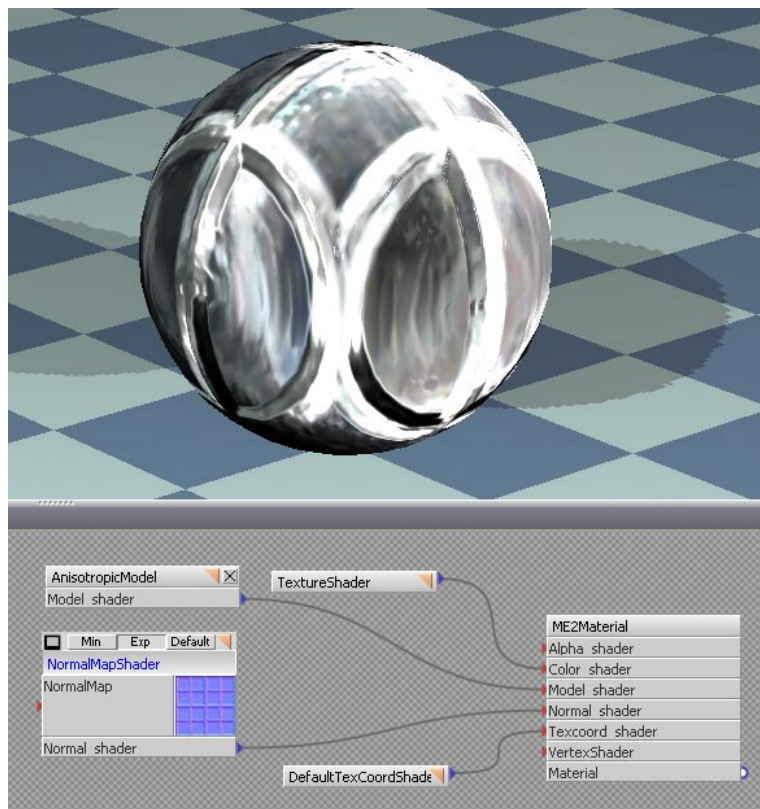
The Sphere with the AnisotropicModel shader component added

Now for a little color, how about deleting the SolidColorShader and dragging in the TextureShader object from the library (Shaders|Color Shaders). Connect its Color Shader output to the Color Shader input on the Material object. Feel free to change the bitmap to something more interesting than the default grid. Here is an image of our sphere, with the default texture in TextureShader applied.



The Sphere with the TextureShader component added

Now, let's add one final component; a normal shader component. Find the NormalMapShader in the Shader library and drag it into the Material Editor. Connect its Normal Shader output to the Normal Shader input of the Material object. Now the Sphere has a bumpy grid to match its grid texture. Your material and its result on the Sphere are shown in the following image.



The finished material and its result in the Player view

Truthfully, our finished material doesn't look like much but this lesson has hopefully shown you the basics of building a material from "stock" shader components. Try some other library items to see how they affect your final material and also be sure to enter library components that have an orange enter icon to examine how they are built. You can try making small changes here and there to see how they change the output. Do not worry if you break something, just delete the object and get a new one from the library.

Tutorial: Examining a More Advanced DirectX Material

In this tutorial we will go inside a more complex material to see how it functions, in this case a material that blends between two bitmaps based on the direction of the surface normal. First, locate the Terrain System object (Objects|Script Objects) and drag it into the Link Editor. You should see a set of rolling hills in the player window (you may need to zoom out a bit).



This material blends between two bitmaps based on surface normal

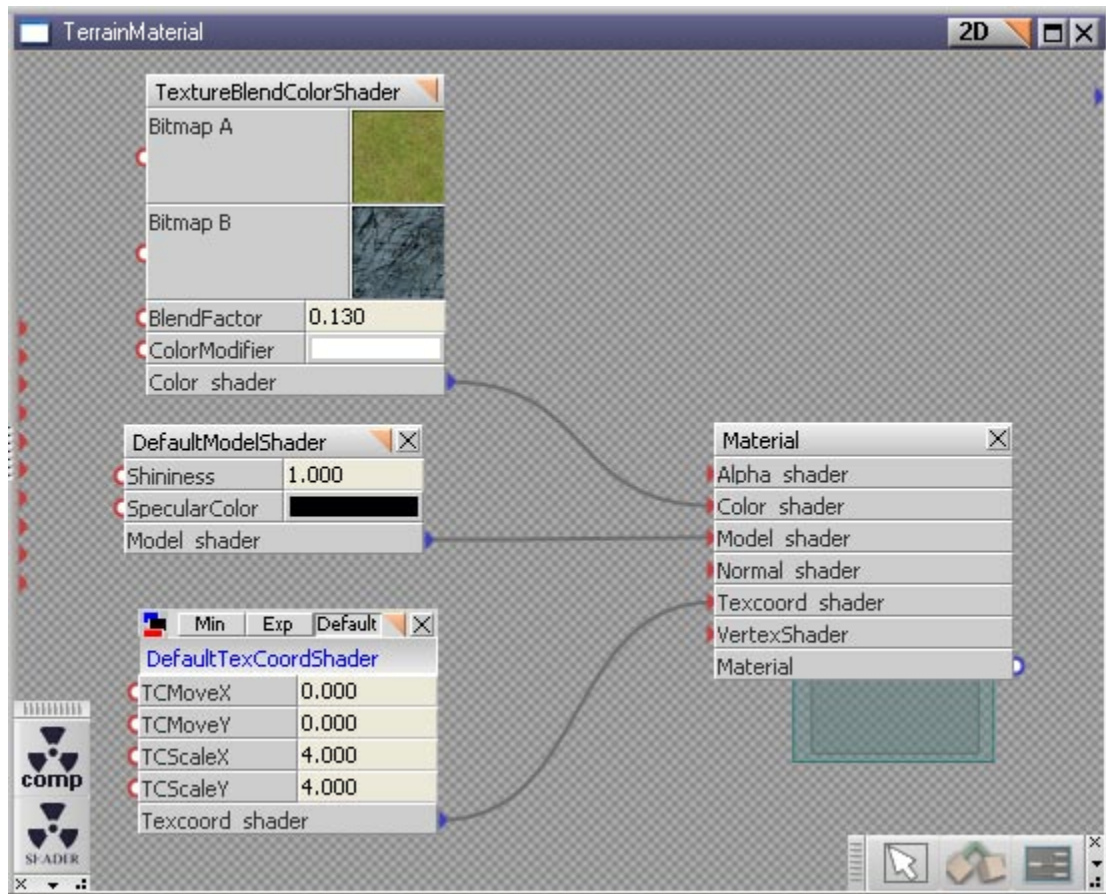
Try changing the Z Scale slider on the front panel of the Terrain System object. You should notice that, as the terrain is scaled in Z, the texture on it changes. As the terrain becomes flatter you see that more parts of it are covered with a grassy texture. As it becomes rougher the vertical areas begin to be covered with a rocky texture. This is the result of a texture blending material within the object.

Now, enter the object in the Link Editor and look around. There are a lot of objects here but we are primarily interested in the TerrainMaterial object shown in the image below. If you would like to learn more about how the terrain geometry itself is created then see the terrain tutorial in **DEVELOPER GUIDE CHAPTER 4: SCRIPTING**. For now, though, just go ahead and enter the TerrainMaterial object.



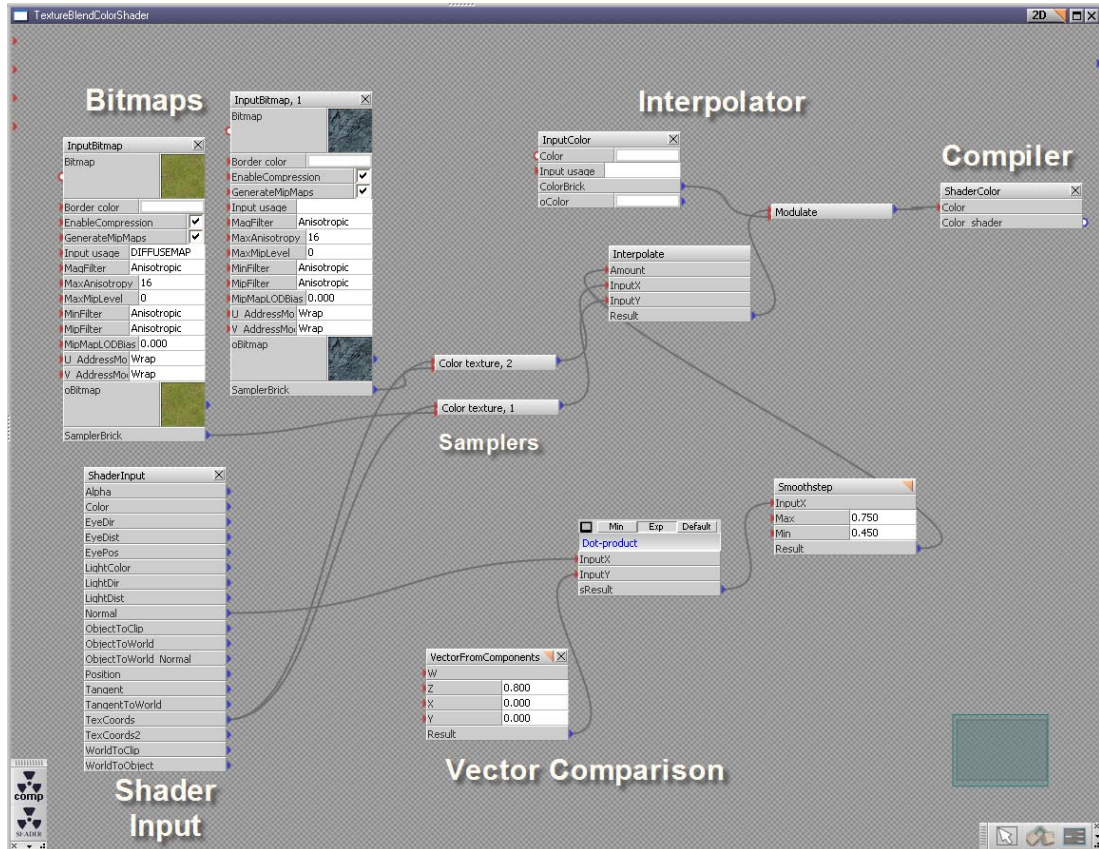
The TerrainMaterial object in the Link Editor

Inside (see image below) you will find four objects: Material, DefaultTexCoordShader, DefaultModelShader, and TextureBlendColorShader. We have already explored the first three objects in depth in the previous sections so our main area of focus is the TextureBlendColorShader where the real work of the material is carried out.



Inside the TerrainMaterial object in the Material Editor

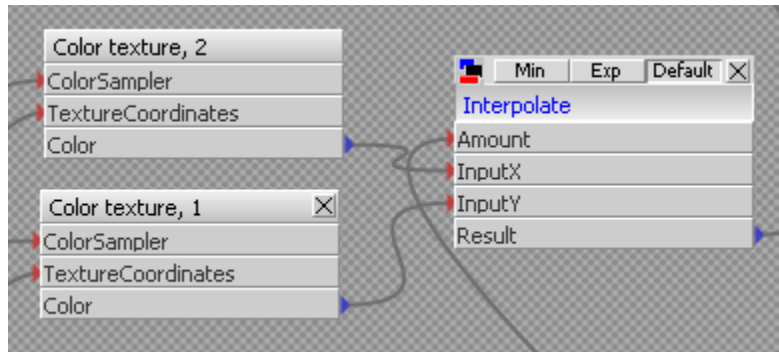
Enter TextureBlendColorShader and take a look at its contents, shown below. You should see a pretty complex network of shader objects, all feeding into a ShaderColor object at the right.



TerrainMaterial input bitmaps, samplers, blender, and color shader compiler

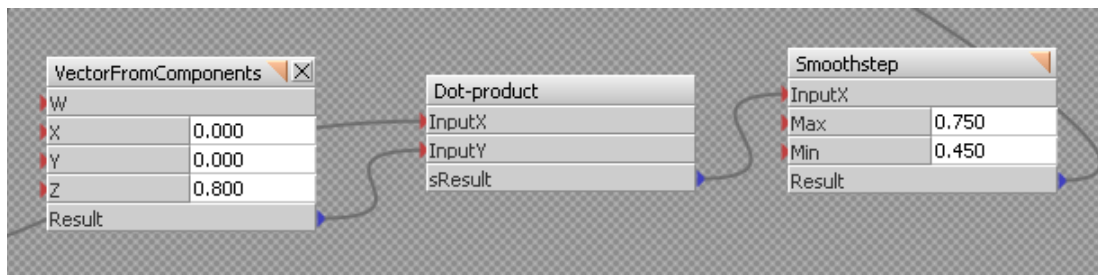
Let's take it from the top and follow the flow of data in this material. First, at the upper left of the construct you should see two InputBitmap objects. These provide the basic color information that will be used to calculate the final color for any given pixel on the material's surface. As each pixel is rendered we will grab a value from these bitmaps using Color Texture objects. These objects take texture coordinates from the ShaderInput object and then look into the bitmap to find the color at that location.

Now the shader needs to blend between those two colors. We will do this based on surface normal - the vector perpendicular to the surface at the pixel currently being rendered - using an Interpolate object. This object blends between two values based on the input value Amount. You can see that the Color output from the two Color Texture objects feeds into the Interpolate object.



The bitmap samplers feed color values into Interpolate, which blends based on Amount

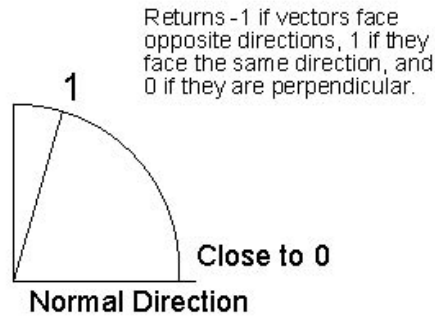
Amount comes from another structure, a calculation based on the surface normal (see image below). This structure performs a dot product operation on the surface normal (obtained from ShaderInput) and a vector (0.0, 0.0, 0.8) created by the InputFloat and VectorFromComponents objects.



This structure calculates blend Amount based on the surface normal

Dot product returns a number from -1 to 1 by comparing two vectors (directions in 3D space). The result will be 1 if the vectors face in the same direction, 0 if they are perpendicular to each other, and -1 if they face opposite directions. In this case we are comparing the surface normal to a normal that points nearly vertically (0, 0, 0.8). Thus, if the surface normal points straight up (i.e. the terrain is 'flat') the value will be close to 1. If the surface normal points horizontally (i.e. the terrain is 'steep') the value will approach 0. Incidentally, we know that the value will be between 0 and 1 because the terrain mesh never faces 'downward', always 'upward'.

Dot Product Results



Now that we have this result we can use it to control Amount on the texture interpolator described earlier. First though, we'll perform one more calculation; this is where the Smoothstep and the other two Input-Float objects come into play. Smoothstep returns a Result of 0 if the value of InputX is less than the value of Min and 1 if it is above Max. For InputX values between Min and Max the object will return a Result that is blended smoothly between the two.

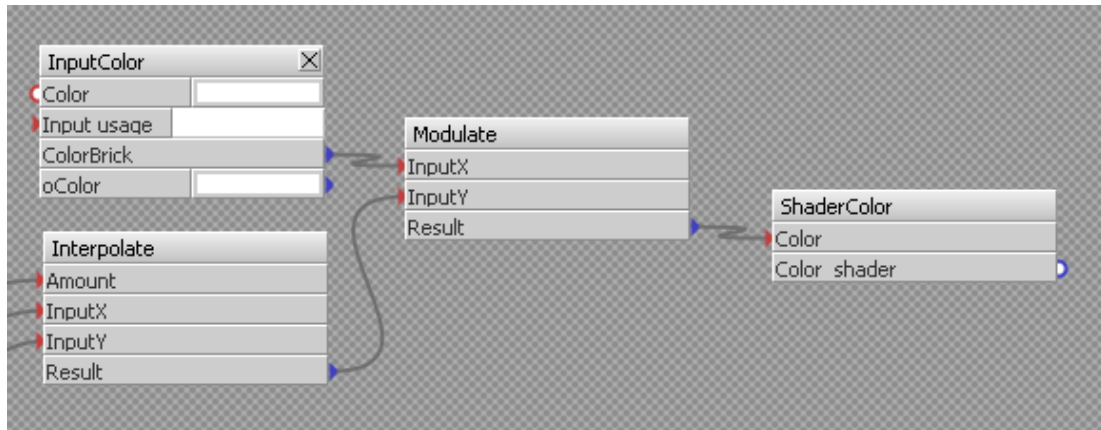
Smoothstep Results

Blends between Min and Max. Values higher than Max are set to 1 and values lower than Min are set to 0.



The effect of this, in English, is to say that if “the terrain is mainly flat then return a value of 1, if it is mostly vertical then return a value of 0, if it is somewhere in between Min and Max then return a blended value. This result controls the blending between grass and rock texture by the Interpolate object, giving us our nicely blended terrain material.

Finally, the resulting blended color is mixed with white via a Modulate object (essentially a multiplication) and output to the ShaderColor object for use with our Material object.



Final texture blending and mixing with white before outputting the Color Shader

To experiment with this shader component you could try changing the values of the various InputFloat objects to see the effects. For instance, if you change the value of the floating point number feeding into Dot-product you should see a corresponding change in the material. Try changing the values that feed into the Smoothstep object as well to see how it controls blending. You can also try using different bitmaps for some interesting effects.

As an exercise when you're ready to move on to even more advanced shader creation, try adding a third bitmap and blending it into the mix. You could try adding a darker green foliage texture that only covers those parts of the terrain that are almost completely flat, to simulate areas where water gathers and plants grow more thickly.

3.3 Writing Shader Scripts

For advanced developers, shader scripts offer an even wider range of possibilities than the trueSpace Material Editor. Depending on your needs you can create shader scripts using HLSL for DirectX materials or using the LUA scripting language for Super LightWorks materials. Both types of shader scripts are covered in the sections below.

Writing DirectX Shader Scripts

trueSpace DirectX shader scripts are built using a special Material Editor component called the HLSL Script Brick. By adding this script-based object, defining attributes, and creating an HLSL script to act on those attributes you can create any type of DirectX shader component.

What is HLSL?

Microsoft's HLSL (high-level shader language) is a special scripting language that allows developers to write shader programs that run on compatible graphics cards. HLSL is very similar to the C programming language with some extensions and limitations related specifically to programming graphics hardware.

A full description of HLSL is beyond the scope of this reference but you are encouraged to read the HLSL documentation and reference guide online at the Microsoft Developer Network site:

- The Microsoft DirectX developer site includes tutorials, articles, and an HLSL reference guide.
<http://msdn.microsoft.com/directx/>
- The starting page of the DirectX Graphics documentation guide. The documentation tree also includes an HLSL guide with many shader examples and tutorials.
http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/dxgraphics.asp
- The HLSL reference guide contains description of the language syntax, supported types, and all available functions.
http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/reference/hlsreference/hlsreference.asp

When learning about HLSL it is important to understand the differences between HLSL and standard C. For example, HLSL supports vector operations on structures; it supports access to multiple members of a vector in a single command and allows 'component swizzling' and 'component masking' which allows you to modify the order of vector members inside the command.

HLSL in trueSpace

trueSpace makes available a customized subset of HLSL, detailed below, to shader authors.

Comments

HLSL comments are the same as in C. Single line comments are preceded by two forward slashes (//). Block comments begin with a forward slash followed by an asterisk (/*) and end with an asterisk followed by a forward slash (*/).

```
//This is a single line comment

/*
    This is a block comment
*/
```

Data types

trueSpace provides a number of custom HLSL data types including types for vectors and matrices.

RtFloat

RtFloat is a single-component floating point number.

```
RtFloat x;  
RtFloat b = 6.0f;
```

RtFloat3

RtFloat3 is a three-component floating point vector structure. The member components of this vector can be accessed using .x, .y, .z or .r, .g, .b suffix.

```
RtFloat3 x;  
RtFloat3 b = {1.0f, 2.0f, 3.0f};  
RtFloat3 c = RtFloat3(1.0f, 2.0f, 3.0f);
```

RtFloat4

RtFloat4 is a four-component floating point vector structure. The member components of this vector can be accessed using .x, .y, .z, .w or .r, .g, .b, .a suffix.

```
RtFloat4 x;  
RtFloat4 y = {1.0f, 2.0f, 3.0f, 4.0f};  
RtFloat4 z = RtFloat3(1.0f, 2.0f, 3.0f, 4.0f);
```

RtFloat3x3 and RtFloat4x4

RtFloat3x3 and RtFloat4x4 are 3x3 and 4x4 component matrix structures, respectively. A matrix contains values organized in rows and columns and provides several member access methods (shown for a 4x4 matrix. The 3x3 matrix uses the top left of the 4x4 matrix).

The zero-based row-column positions are:

```
_m00, _m01, _m02, _m03  
_m10, _m11, _m12, _m13  
_m20, _m21, _m22, _m23  
_m30, _m31, _m32, _m33
```

To access the value of row 0, column 0 use:

```
RtFloat x = Matrix._m00;
```

The one-based row-column positions are:

```

_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44

```

To access the value of row 1, column 1, use:

```
RtFloat x = Matrix._11;
```

Using an “array” type access the positions are:

```

[0][0], [0][1], [0][2], [0][3]
[1][0], [1][1], [1][2], [1][3]
[2][0], [2][1], [2][2], [2][3]
[3][0], [3][1], [3][2], [3][3]

```

To access the value at row 0, column 0, use:

```
RtFloat x = Matrix.[0][0];
```

To declare or declare and initialize a matrix, use:

```

RtFloat4x4 Matrix1;
RtFloat3x3 Matrix2 = {      1.0f, 2.0f, 3.0f, // row 1
                          4.0f, 5.0f, 6.0f, // row 2
                          7.0f, 8.0f, 9.0f, // row
                          3};

```

RtSampler1D and RtSampler2D objects

RtSampler1D is used to load color data from a one-dimensional texture, i.e. only an X coordinate. RtSampler2D can load color data from two-dimensional textures. Please note that samplers cannot be declared inside of an HLSL function; they are used only as input parameters.

Type conversion

To convert between data types you should use one of the methods provided below. These are the same methods that automatically convert between data types when connecting attributes of different types in the Material Editor itself. Please note that sampler objects cannot be converted.

When you need to convert one data type to another, use the following syntax:

```
RsConvert_<DestType>_<SourceType>(<destination_variable>, <source_
variable>)
```

The following example converts an RtFloat4 vector to a single RtFloat value. It does this by picking only the .x component of the vector. In this case they .y, .z, and .w components are not converted because RtFloat can only hold a single floating point number.

```
RsConvert_RtFloat_RtFloat4(MyFloat, MyVector);
```

Conversion rules:

- Conversion to RtFloat is done by selecting the .x component of a vector or ._m00 component of a matrix. Conversion from RtFloat to any type components of the target type with that value.
- Conversion from RtFloat3 into RtFloat4 is done by filling the .w component of the target vector with a value of zero. Other components are filled by assignment. When converting from RtFloat4 to RtFloat3 the value of the .w component is lost
- Conversion from vector to matrix is done by replicating the vector into all rows of the matrix.
- Conversion from RtFloat3x3 into RtFloat4x4 is done by adding zero values to the last column and row of the matrix. When converting from RtFloat4x4 to RtFloat3x3 the last column and row values are lost.
- In any other case the conversion is done by filling matching components.

Swizzling and masking, per-component operations

HLSL can perform per-component operations on vector and matrix types. All basic operations (addition, subtraction, etc.) and comparisons are performed on a per-component basis. The following example shows the addition of two RtFloat3 vectors:

```
RtFloat3 dest, src0, src1;
dest = src0 + src1;
```

This is the same as writing:

```
dest.x = src0.x + src1.x;
dest.y = src0.y + src1.y;
dest.z = src0.z + src1.z;
```

Also almost all functions that work on scalar values (i.e. sin, cos, max, min, and many other) also accept either vector or matrix operands and perform the operation on all components of the operands.

Another extension of HLSL allows you to specify only a portion of a vector or matrix to take part in the operation. Masking on the destination variable allows you to limit the operation to affect only specific parts of the operand vector or matrix. The following example shows how to write only into the first two components of the vector while the .z component remains unchanged.

```
RtFloat3 dest, src0, src1;
dest.xy = src0 + src1;
```

This is the same as writing:

```
dest.x = src0.x + src1.x;
dest.y = src0.y + src1.y;
dest.z = dest.z;           //just to show the unchanged component
```

Masking on source registers allows you to limit the operation to using only a part of the vector or matrix.

```
RtFloat3 dest, src0, src1;  
dest = src0 + src1.x;
```

This is the same as writing:

```
dest.x = src0.x + src1.x;  
dest.y = src0.y + src1.x;  
dest.z = src0.z + src1.x;
```

Swizzling allows you to change the order of components in the operation.

```
RtFloat3 dest, src0, src1;  
dest = src0.xyz + src1.zyz;
```

This is the same as writing:

```
dest.x = src0.x + src1.z;  
dest.y = src0.y + src1.y;  
dest.z = src0.z + src1.x;
```

Please refer to the DirectX HLSL reference and programming guide for a full list of allowed swizzle and masking combinations for various shader versions.

Functions

Functions are defined similarly to the C or Java (Jscript language).

```
return_type function_name(access param_type param_name1,  
                           access param_type param_name2,  
                           ...,  
                           access param_type param_name3)  
{  
    //function body  
}
```

The return type can be also be void, which specifies that the function does not return a value using the return keyword.

The access specifier tells if the parameter can be read, written, or both:

- in - input parameter can be read within the function and is filled by the caller of the function

- out - output parameter can be written within the function and is read by the caller of the function. Can be used as a return value
- inout – parameter acts as in and out parameter at the same time.

Example:

```
//Function returns maximum of two values and fills Minimum parameter with minimum
//of two values
RtFloat3 MinMax(in RtFloat3 input1, in RtFloat3 input2, out RtFloat3 Minimum)
{
    Minimum = rtx_Min(input1, input2);
    return rtx_Max(input1, input2);
}
```

Note that rtx_Min and rtx_Max are predefined functions that will be described later.

Macros

trueSpace HLSL supports some preprocessor directives similar to the C language. This includes #define, #undef, #if, #elif, #else, #ifdef, #endif, #ifndef. Refer to the HLSL online documentation for a description of these preprocessor directives and available modifications.

Rosetta defines these custom tokens using #define:

- One (and only one) of following tokens specifying available pixel shader version: RSD3D_PROFILE_PS_2_0, RSD3D_PROFILE_PS_2_A, RSD3D_PROFILE_PS_2_B, RSD3D_PROFILE_PS_3_0.
- One (and only one) of following tokens specifying available vertex shader version: RSD3D_PROFILE_VS_2_0, RSD3D_PROFILE_VS_2_A, RSD3D_PROFILE_VS_3_0.
- #define ME_PS_PROFILE available_profile where available_profile might be “ps_2_0”, “ps_2_a”, “ps_2_b” or “ps_3_0”.
- #define ME_VS_PROFILE available_profile where available_profile might be “vs_2_0”, “vs_2_a” or “vs_3_0”.

These predefined macros allow you to modify your shader code according to the available shader version. Macros and preprocessor directives are a very powerful means of controlling the generated shader code. You should never change or undefine any predefined trueSpace macro or use directives that might affect the shader outside of the scope of your function.

Predefined functions

trueSpace HLSL provides a number of predefined functions that you can use in your own HLSL scripts. Most of these functions are also available in the form of shader components in the Material Editor.

The following table contains a complete list of predefined functions along with the syntax for using them. All functions are mappings of the original predefined HLSL functions and the name of the original function is shown in bold in the Description column. Use these original names to find more detailed help in the DirectX HLSL reference guide. All functions have the syntax: `returnvalue = function(parameters)`. Some functions also return an additional value as the last parameter.

Name	Syntax	Description
rtx_Abs	rtx_Abs(a)	Absolute value (per component). [abs]
rxt_Acos	rxt_Acos(x)	Returns the arccosine of each component of x. Each component should be in the range [-1, 1]. [acos]
rxt_All	rxt_All(x)	Test if all components of x are nonzero. [all]
rxt_Any	rxt_Any(x)	Test if any component of x is nonzero. [any]
rxt_Asin	rxt_Asin(x)	Returns the arcsine of each component of x. Each component should be in the range [-pi/2, pi/2]. [asin]
rxt_Atan	rxt_Atan(x)	Returns the arctangent of x. The return values are in the range [-pi/2, pi/2]. [atan]
rxt_Atan2	rxt_Atan2(y, x)	Returns the arctangent of y/x. The signs of y and x are used to determine the quadrant of the return values in the range [-pi, pi]. atan2 is well-defined for every point other than the origin, even if x equals 0 and y does not equal 0. [atan2]
rxt_Ceil	rxt_Ceil(x)	Returns the smallest integer which is greater than or equal to x. [ceil]
rxt_Clamp	rxt_Clamp(x, min, max)	Clamps x to the range [min, max]. [clamp]
rxt_Clip	rxt_Clip(x)	Discards the current pixel, if any component of x is less than zero. This can be used to simulate clip planes, if each component of x represents the distance from a plane. [clip]
rxt_Cos	rxt_Cos(x)	Returns the cosine of x. [cos]
rxt_Cosh	rxt_Cosh(x)	Returns the hyperbolic cosine of x. [cosh]
rxt_Cross	rxt_Cross(a, b)	Returns the cross product of two 3-D vectors a and b. [cross]
rxt_Ddx	rxt_Ddx(x)	Returns the partial derivative of x with respect to the screen-space x-coordinate. [ddx]
rxt_Ddy	rxt_Ddy(x)	Returns the partial derivative of x with respect to the screen-space y-coordinate. [ddy]
rxt_Degrees	rxt_Degrees(x)	Converts x from radians to degrees. [degrees]
rxt_Determinant	rxt_Determinant(m)	Returns the determinant of the square matrix m. [determinant]
rxt_Distance	rxt_Distance(a, b)	Returns the distance between two points, a and b. [distance]
rxt_Dot	rxt_Dot(a, b)	Returns the • product of two vectors, a and b. [dot]
rxt_Exp	rxt_Exp(x)	Returns the base-e exponent. [exp]
rxt_Exp2	rxt_Exp2(a)	Base 2 Exp (per component). [exp2]
rxt_Faceforward	rxt_Faceforward(n, i, ng)	Returns -n * sign(•(i, ng)). [faceforward]
rxt_Floor	rxt_Floor(x)	Returns the greatest integer which is less than or equal to x. [floor]

<code>rxt_Fmod</code>	<code>rxt_Fmod(a, b)</code>	Returns the floating point remainder f of a / b such that $a = i * b + f$, where i is an integer, f has the same sign as x , and the absolute value of f is less than the absolute value of b . [fmod]
<code>rxt_Frac</code>	<code>rxt_Frac(x)</code>	Returns the fractional part f of x , such that f is a value greater than or equal to 0, and less than 1. [frac]
<code>rxt_Frexp</code>	<code>rxt_Frexp(x, out exp)</code>	Returns the mantissa and exponent of x . <code>frexp</code> returns the mantissa, and the exponent is stored in the output parameter <code>exp</code> . If x is 0, the function returns 0 for both the mantissa and the exponent. [frexp]
<code>rxt_Fwidth</code>	<code>rxt_Fwidth(x)</code>	Returns $\text{abs}(\text{ddx}(x)) + \text{abs}(\text{ddy}(x))$. [fwidth]
<code>rxt_IsFinite</code>	<code>rxt_IsFinite(x)</code>	Returns true if x is finite, false otherwise. [isfinite]
<code>rxt_IsInfinite</code>	<code>rxt_IsInfinite(x)</code>	Returns true if x is $+\text{INF}$ or $-\text{INF}$, false otherwise. [isinf]
<code>rxt_IsNaN</code>	<code>rxt_IsNaN(x)</code>	Returns true if x is NaN or QNaN, false otherwise. [isnan]
<code>rxt_Ldexp</code>	<code>rxt_Ldexp(x, exp)</code>	Returns $x * 2^{\text{exp}}$. [ldexp]
<code>rxt_Length</code>	<code>rxt_Length(v)</code>	Returns the length of the vector v . [length]
<code>rxt_Lerp</code>	<code>rxt_Lerp(a, b, s)</code>	Returns $a + s(b - a)$. This linearly interpolates between a and b , such that the return value is a when s is 0, and b when s is 1. [lerp]
<code>rxt_Lit</code>	<code>rxt_Lit(n • l, n • h, m)</code>	Returns a lighting vector (ambient, diffuse, specular, 1): ambient = 1; diffuse = $(n \cdot l < 0) ? 0 : n \cdot l$; specular = $(n \cdot l < 0) \parallel (n \cdot h < 0) ? 0 : (n \cdot h * m)$; [lit]
<code>rxt_Log</code>	<code>rxt_Log(x)</code>	Returns the base- e logarithm of x . If x is negative, the function returns indefinite. If x is 0, the function returns $+\text{INF}$. [log]
<code>rxt_Log10</code>	<code>rxt_Log10(x)</code>	Returns the base-10 logarithm of x . If x is negative, the function returns indefinite. If x is 0, the function returns $+\text{INF}$. [log10]
<code>rxt_Log2</code>	<code>rxt_Log2(x)</code>	Returns the base-2 logarithm of x . If x is negative, the function returns indefinite. If x is 0, the function returns $+\text{INF}$. [log2]
<code>rxt_Max</code>	<code>rxt_Max(a, b)</code>	Selects the greater of a and b . [max]
<code>rxt_Min</code>	<code>rxt_Min(a, b)</code>	Selects the lesser of a and b . [min]
<code>rxt_Modf</code>	<code>rxt_Modf(x, out ip)</code>	Splits the value x into fractional and integer parts, each of which has the same sign and x . The signed fractional portion of x is returned. The integer portion is stored in the output parameter <code>ip</code> . [modf]

rxt_Mul	rxt_Mul(a, b)	Performs matrix multiplication between a and b. If a is a vector, it is treated as a row vector. If b is a vector, it is treated as a column vector. The inner dimension acolumns and brows must be equal. The result has the dimension arows x bcolumns. [mul]
rxt_Normalize	rxt_Normalize(v)	Returns the normalized vector $v / \text{length}(v)$. If the length of v is 0, the result is indefinite. [normalize]
rxt_Pow	rxt_Pow(x, y)	Returns xy . [pow]
rxt_Radians	rxt_Radians(x)	Converts x from degrees to radians. [radians]
rxt_Reflect	rxt_Reflect(i, n)	Returns the reflection vector v, given the entering ray direction i, and the surface normal n, such that $v = i - 2 * (i \cdot n) * n$. [reflect]
rxt_Refract	rxt_Refract(i, n, ?)	Returns the refraction vector v, given the entering ray direction i, the surface normal n, and the relative index of refraction ?. If the angle between i and n is too great for a given ?, refract returns (0,0,0). [refract]
rxt_Round	rxt_Round(x)	Rounds x to the nearest integer. [round]
rxt_Rsqrt	rxt_Rsqrt(x)	Returns $1/\text{rtx_Sqrt}(x)$. [rsqrt]
rxt_Saturate	rxt_Saturate(x)	Clamps x to the range [0, 1]. [saturate]
rxt_Sign	rxt_Sign(x)	Computes the sign of x. Returns -1 if x is less than 0, 0 if x equals 0, and 1 if x is greater than zero. [sign]
rxt_Sin	rxt_Sin(x)	Returns the sine of x. [sin]
rxt_SinCos	rxt_SinCos(x, out s, out c)	Returns the sine and cosine of x. sin(x) is stored in the output parameter s. cos(x) is stored in the output parameter c. [sincos]
rxt_Sinh	rxt_Sinh(x)	Returns the hyperbolic sine of x. [sinh]
rxt_Smoothstep	rxt_Smoothstep (min, max, x)	Returns 0 if $x < \text{min}$. Returns 1 if $x > \text{max}$. Returns a smooth Hermite interpolation between 0 and 1, if x is in the range [min, max]. [smoothstep]
rxt_Sqrt	rxt_Sqrt(a)	Square root (per component). [sqrt]
rxt_Step	rxt_Step(a, x)	Returns $(x \geq a) ? 1 : 0$. [step]
rxt_Tan	rxt_Tan(x)	Returns the tangent of x. [tan]
rxt_Tanh	rxt_Tanh(x)	Returns the hyperbolic tangent of x. [tanh]
rxt_Tex1D	rxt_Tex1D(s, t)	1-D texture lookup. s is a sampler or a sampler1D object. t is a scalar. [tex1D]
rxt_Tex1D	rxt_Tex1D(s, t, ddx, ddy)	1-D texture lookup, with derivatives. S is a sampler or sampler1D object. t, ddx, and ddy are scalars. [tex1D]

<code>rxl_Tex1DProj</code>	<code>rxl_Tex1DProj(s, t)</code>	1-D projective texture lookup. <i>s</i> is a sampler or sampler1D object. <i>t</i> is a 4-D vector. <i>t</i> is divided by its last component before the lookup takes place. [tex1Dproj]
<code>rxl_Tex1DBias</code>	<code>rxl_Tex1DBias(s, t)</code>	1-D biased texture lookup. <i>S</i> is a sampler or sampler1D object. <i>t</i> is a 4-D vector. The mip level is biased by <i>t.w</i> before the lookup takes place. [tex1Dbias]
<code>rxl_Tex2D</code>	<code>rxl_Tex2D(s, t)</code>	2-D texture lookup. <i>s</i> is a sampler or a sampler2D object. <i>t</i> is a 2-D texture coordinate. [tex2D]
<code>rxl_Tex2D</code>	<code>rxl_Tex2D(s, t, ddx, ddy)</code>	2-D texture lookup, with derivatives. <i>s</i> is a sampler or sampler2D object. <i>t</i> , <i>ddx</i> , and <i>ddy</i> are 2-D vectors. [tex2D]
<code>rxl_Tex2DProj</code>	<code>rxl_Tex2DProj(s, t)</code>	2-D projective texture lookup. <i>s</i> is a sampler or sampler2D object. <i>t</i> is a 4-D vector. <i>t</i> is divided by its last component before the lookup takes place. [tex2Dproj]
<code>rxl_Tex2DBias</code>	<code>rxl_Tex2DBias(s, t)</code>	2-D biased texture lookup. <i>s</i> is a sampler or sampler2D object. <i>t</i> is a 4-D vector. The mip level is biased by <i>t.w</i> before the lookup takes place. [tex2Dbias]
<code>rxl_Transpose</code>	<code>rxl_Transpose(m)</code>	Returns the transpose of the matrix <i>m</i> . If the source is dimension <i>mrows</i> x <i>mcolums</i> , the result is dimension <i>mcolums</i> x <i>mrows</i> . [transpose]
<code>rxl_Modulate</code>	<code>rxl_Modulate(a,b)</code>	Returns per component <i>a</i> * <i>b</i> . [*]
<code>rxl_Add</code>	<code>rxl_Add(a,b)</code>	Returns per component <i>a</i> + <i>b</i> . [+]
<code>rxl_Sub</code>	<code>rxl_Sub(a,b)</code>	Returns per component <i>a</i> – <i>b</i> . [-]
<code>rxl_Divide</code>	<code>rxl_Divide(a,b)</code>	Returns per component <i>a</i> / <i>b</i> . [/]
<code>rxl_Modulus</code>	<code>rxl_Modulus(a,b)</code>	Returns per component <i>a</i> modulus <i>b</i> . [%]
<code>rxl_Negate</code>	<code>rxl_Negate(a)</code>	Returns per component – <i>a</i> . [-]
<code>rxl_Not</code>	<code>rxl_Not(a)</code>	Returns per component logical not <i>a</i> . [!]

trueSpace HLSL Shader Components

HLSL shader components are well-defined pieces of shader code that can be combined together in the Material Editor to create a shader. The core of a shader component is a function with a name that is the same as the name of the component. This function has a custom set of input and output parameters. trueSpace represents this function as a visible object in the Material Editor and you can connect the object to other shader components to create your material.

Shader components have several important constraints:

- The shader component function cannot return a value using the *return* keyword – all shader components use functions with a return type of *void*.

- Function parameters can only use *in* or *out* access specifiers. They cannot include *inout* access specifiers.
- Function parameters can be only be one of the previously specified trueSpace HLSL data types (i.e. RtFloat, RtFloat3x3, etc.). Other HLSL datatypes (like float2) can be only specified inside of the function as local variables.
- If you declare helper functions to be used by the main shader component function, you can use any data types for parameters or the return value. The name of these functions must, however, contain the name of the main function as a suffix.
- Violation of these rules will, in most cases, result in uncompileable shader code.

Predefined Shader Components

The following shader components are available in the context-sensitive shader component library.

Compound

Name	Alpha Texture HL
Description	This component returns alpha stored in the red channel of the texture
Attributes	
in RtSampler2D AlphaSampler	Texture that contains alpha values in the red channel
in RtFloat3 TextureCoordinates	Sampling texture coordinates
in RtFloat AlphaTexScaleX	Horizontal scale of the texture coordinates
in RtFloat AlphaTexScaleY	Vertical scale of the texture coordinates
in RtFloat AlphaTexMoveX	Horizontal translate of the texture coordinates
in RtFloat AlphaTexMoveY	Vertical translate of the texture coordinates
out RtFloat Alpha	resulting alpha value

Name	Anisotropic lighting
Description	Simple anisotropic lighting model
Attributes	
in RtFloat4 DiffuseColor	Source diffuse color
in RtFloat3 NormalVector	Current normal vector of the point
in RtFloat3 EyeDirection	Current eye direction
in RtFloat3 LightDirection	Current light direction
in RtFloat4 LightColor	Color of the light
in RtSampler2D AnisotropicMap	Special texture that represents the actual amount of lighting it is stored in the file <installdir>/ts/scripts/d3d/Aniso.png
out RtFloat4 ResultColor	final color of the point

Name	Color texture	
Description	This brick returns color stored in a texture	
Attributes		
in RtSampler2D ColorSampler	Source 2D texture	
in RtFloat3 TextureCoordinates	texture coordinates	
out RtFloat4 Color	resulting color	

Name	Color Texture HL	
Description	This brick returns color stored in a texture with modified texture coordinates	
Attributes		
in RtSampler2D ColorSampler	Source 2D texture	
in RtFloat3 TextureCoordinates	texture coordinates	
in RtFloat ColorTexScaleX	Horizontal scale of the texture coordinates	
in RtFloat ColorTexScaleY	Vertical scale of the texture coordinates	
in RtFloat ColorTexMoveX	Horizontal translate of the texture coordinates	
in RtFloat ColorTexMoveY	Vertical translate of the texture coordinates	
out RtFloat4 Color	resulting color value	

Name	Cook-Torrance lighting	
Description	Brick computes the Cook-Torrance lighting model which is good for reflective surfaces such as metal. On Pixel Shader 2.0 hardware it falls-back to Phong model with equal the same and diffuse color components which produces similar metallic effect.	
Attributes		
in RtFloat Roughness	The roughness of the surface. Expected values should be in range 0...1 and higher roughness means less shiny surface and less intensive highlights.	
in RtFloat RefractionIndex	This parameter controls the amount of reflected and refracted light. Expected values are in range 0...1. Low values mean smaller reflectivity with more account to real Fresnel term.	
in RtFloat4 DiffuseColor	Diffuse color of the point	
in RtFloat3 NormalVector	Normal vector of the point	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	Cook-Torrance 2
Description	Brick computes the Cook-Torrance lighting model which is good for reflective surfaces such as metal. On Pixel Shader 2.0 hardware it falls-back to Phong model with equal the same and diffuse color components which produces similar metallic effect. The difference from previous brick is that most of the computation is stored in a special function which saves instructions.
Attributes	
in RtSampler2D Precomputation	Fresnel term and Becman distribution function stored in a HDRI texture <installdir>/ts/scripts/d3d/RtD3D_BeckmannDistributionFresnel.dds. Cards without support for HDRI textures will not display this material correctly.
in RtFloat Roughness	The roughness of the surface. Expected values should be in range 0...1 and higher roughness means less shiny surface and less intensive highlights.
in RtFloat RefractionIndex	This parameter controls the amount of reflected and refracted light. Expected values are in range 0...1. Low values mean smaller reflectivity with more account to real Fresnel term.
in RtFloat4 DiffuseColor	Diffuse color of the point
in RtFloat3 NormalVector	Normal vector of the point
in RtFloat3 EyeDirection	Current eye direction
in RtFloat3 LightDirection	Current light direction
in RtFloat4 LightColor	Light color
out RtFloat4 ResultColor	Final, computed color

Name	Gooch lighting	
Description	The Gooch lighting and shading model was developed to better show geometrical properties of objects. It works best with single light setup.	
Attributes		
in RtFloat4 CoolColor	Cool color is used in areas with high angles to the viewer	
in RtFloat4 WarmColor	Warm color is used in areas with small angles to the viewer	
in RtFloat CoolModifier	This modifier changes the amount of diffuse color in cool areas.	
in RtFloat WarmModifier	This modifier changes the amount of diffuse color in warm areas.	
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat4 SpecularColor	Specular color of the material.	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 NormalVector	Current normal vector	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	Gooch lighting 2	
Description	The Gooch lighting and shading model was developed to better show geometrical properties of objects. This brick provides simpler, better controlled model that works better in multilight environments and shadows.	
Attributes		
in RtFloat4 CoolColor	Cool color is used in areas with high angles to the viewer	
in RtFloat4 WarmColor	Warm color is used in areas with small angles to the viewer	
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat4 SpecularColor	Specular color of the material.	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 NormalVector	Current normal vector	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	Hair Shader	
Description	This brick computes Ward's anisotropic lighting model that is used for example for hair simulation.	
Attributes		
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat4 SpecularColor	Specular color of the material.	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 TangentVector	Tangent vector used for computing the surface orientation	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	Normal-Map	
Description	This brick reads normals stored in a normal map and returns these normals transformed into the world space	
Attributes		
in RtSampler2D NormalMap	Normal map texture	
in RtFloat3 TextureCoords	Sampling coordinates	
in RtFloat3x3 TangentToWorld	Matrix that transforms vectors from tangent space to the world space.	
out RtFloat3 NormalVector	Final normal vector	

Name	Normal-Map HL	
Description	This brick reads normal stored in a normal map and returns these normals transformed into the world space. Additional texture coordinates transformation are possible.	
Attributes		
in RtSampler2D NormalMap	Normal map texture	
in RtFloat3 TextureCoords	Sampling coordinates	
in RtFloat3x3 TangentToWorld	Matrix that transforms vectors from tangent space to the world space.	
out RtFloat3 NormalVector	Final normal vector	
in RtFloat NormalTexScaleX	Horizontal scale of the texture coordinates	
in RtFloat NormalTexScaleY	Vertical scale of the texture coordinates	
in RtFloat NormalTexMoveX	Horizontal translate of the texture coordinates	
in RtFloat NormalTexMoveY	Vertical translate of the texture coordinates	

Name	Phong Lighting	
Description	The phong lighting model	
Attributes		
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat4 SpecularColor	Specular color of the material.	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 NormalVector	Current normal vector	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	TS Phong Lighting	
Description	The phong lighting model similar to that used in TS 6.x	
Attributes		
in RtFloat DiffuseStrength	Strength of the diffuse component in range 0...1	
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat SpecularStrength	Strength of the specular component in range 0...1	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 NormalVector	Current normal vector	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
out RtFloat4 ResultColor	Final, computed color	

Name	Thin Film	
Description	The phong lighting model with thin oil film effect on the surface.	
Attributes		
in RtFloat4 DiffuseColor	Current diffuse color.	
in RtFloat4 SpecularColor	Specular color of the material.	
in RtFloat Shininess	Shininess of the material	
in RtFloat3 NormalVector	Current normal vector	
in RtFloat3 EyeDirection	Current eye direction	
in RtFloat3 LightDirection	Current light direction	
in RtFloat4 LightColor	Light color	
in RtSampler1D ThinFilmSample	1D texture with the film coat samples. <installdir>/ts/scripts/d3d/thinfilm.dds.	
in RtFloat FilmDepth	Depth of the film layer	
out RtFloat4 ResultColor	Final, computed color	

Name	Hatching NPR
Description	The hatching non photo realistic rendering which uses a texture for the hatch and stroke simulation and chooses hatch level based on the intensity.
Attributes	
in RtFloat4 Intensity	Intensity of the pixel for example the result of lighting model.
in RtFloat3 Texcoords	Texture coordinates of the object for reading of the texture.
in RtSampler2D HatchingTexture	Texture with strokes of different densities in each channel. <installdir>/ts/scripts/d3d/threshold_hatching2.dds.
out RtFloat4 ResultColor	Final, computed color

Functions

All XML files from <installdir>/ts/scripts/MaterialEditor/Functions folder.

Logical

All XML files from <installdir>/ts/scripts/MaterialEditor/Logical folder.

Matrices & Vectors

All XML files from <installdir>/ts/scripts/MaterialEditor/ Matrices & Vectors folder.

Operators

All XML files from <installdir>/ts/scripts/MaterialEditor/ Operators folder.

Texturing

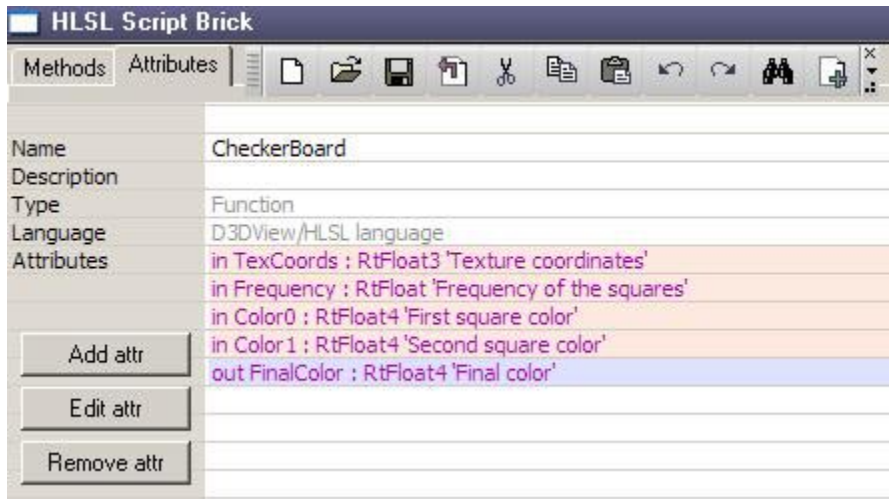
All XML files from <installdir>/ts/scripts/MaterialEditor/ Texturing folder.

Tutorial: A Simple Shader Component with HLSL

This tutorial shows you how to create a shader component that uses texture coordinates to create a color shader outputting alternating color squares like a checker board. After you create the HLSL color shader component you will use it to modify an existing material and apply it to an object.

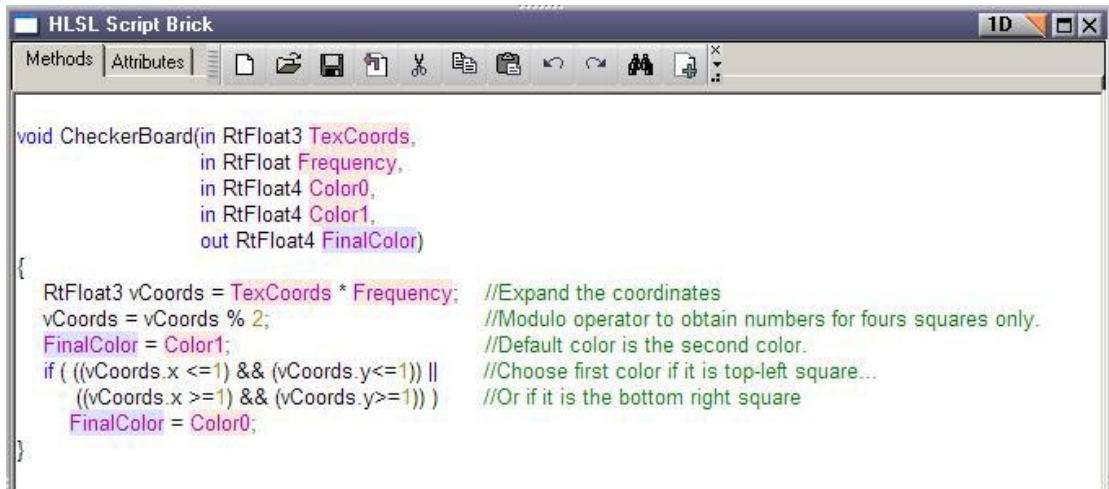
First locate the Cube object, found in the Base panel of the Objects library, and drag it into a new scene. Using the Link Editor, enter the object and then enter the LayeredPlastic material. Finally, enter the TextureBlendColorShader, as shown in the image below.

First you must define the attributes of the shader component. This component requires access to texture coordinates and needs two color inputs (for the checker board colors) and a number representing the frequency of the squares. It also needs one output color. The following image shows the complete list of attributes and their types. Rename the object to something descriptive like *CheckerBoard*.



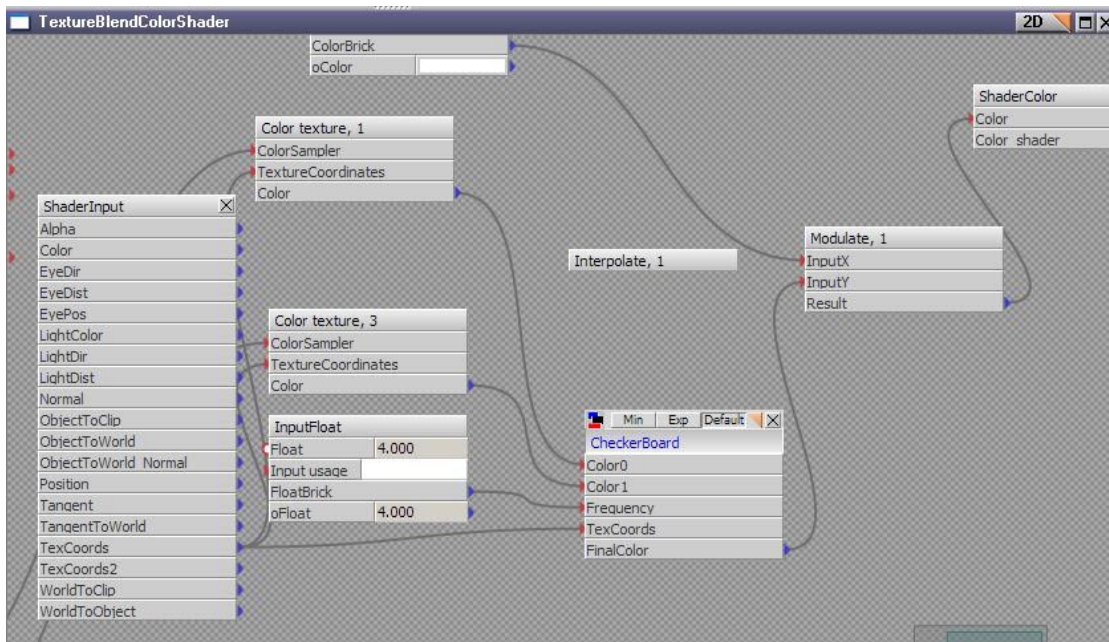
Attributes for the CheckerBoard shader component

The HLSL code to generate the checkerboard pattern is actually fairly simple. First, multiply the texture coordinates by the frequency. This changes the numbers from an initial range of zero to one to a range of zero to the value of frequency. Then use the Modulo operator (%) to determine which color to output for a given texture coordinate. If the -x and -y coordinates are both less than or equal to one or are both greater than one, then Color0 is used, otherwise Color1 is used. The complete script is shown in the following image.



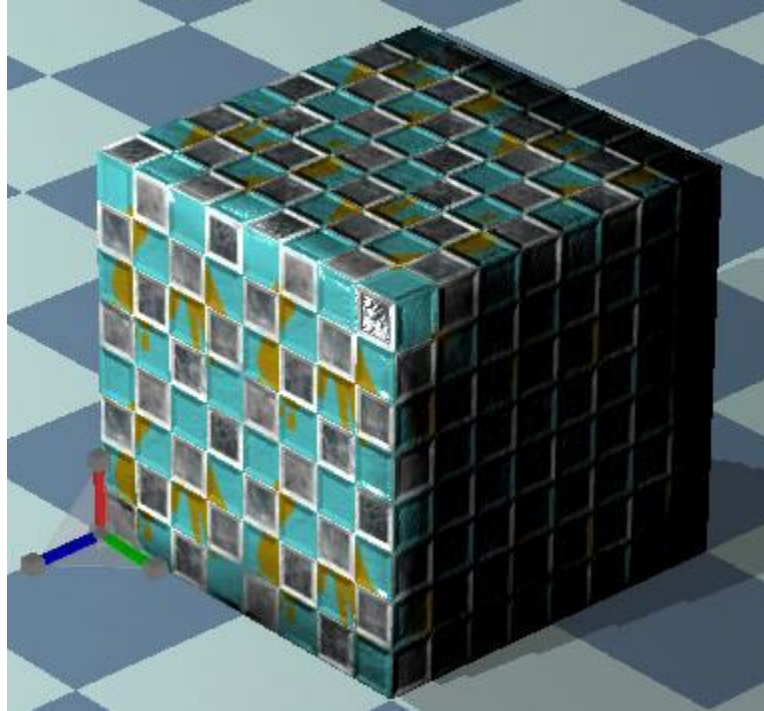
The completed Checkerboard shader component script

After you exit the script editor you can disconnect the Interpolate object, which currently controls texture blending, and connect your new HLSL script instead. Connect FinalColor to InputY on the modulate object and connect the Color outputs of the two Color Texture objects to the Color0 and Color1 inputs of the script. Finally, connect the FloatBrick output of the InputFloat object to the Frequency input on the script object. The connections are shown in the following image.



Connecting the Checkerboard shader component script in the Material Editor

Now, the BlendFactor attribute on the outer LayeredPlastic panel will control the frequency of the checkerboard pattern. Change the value to 4 and observe the results -- you should see something similar to the image below.



Using a BlendFactor of 4 results in a dense two-texture checkerboard

Writing Super LightWorks Material Scripts

Materials for the LightWorks renderer can be created either using the Material Editor or using the LUA scripting language. Currently there is no trueSpace script object for creating LUA scripts for Super LightWorks materials. They must be created as an external file.

What is LUA

LUA is a scripting language that is similar to the C programming language in syntax and is widely used for application automation and other scriptable tasks. LUA is designed to support general procedural programming with data description facilities. Documentation for the LUA scripting language can be found at <http://www.lua.org/docs.html>.

Some general information about LUA

- There are eight basic data types: nil, boolean, number, string, function, userdata, thread, and table.
- LUA has the control structures “if”, “while”, “repeat”, and “for” statements. For example:

```
if( pp== 0 ) then
for I = 0 5 1 do
    tssDiffuseTerm(diff_color[i])
end
end
```

- Arithmetic (+, -, *, /), relational (== ~= < > <= >=) and logical (and or not) expressions are all available in the LUA scripting language
- LUA functions can have more than one return value. You can for example, use `a, b = method(2)` where *a* is the first return value and *b* is the second return value.

```
function method (id)
    tsxSetHelpMessage( "Start shader script" )
    datan = LDataLt.new("color_ME_ColorShader")
    return -1, datan
end
```

LUA in trueSpace

trueSpace uses LUA scripts to describe Super LightWorks materials. The LUA/trueSpace shader pipeline is as follows:

- Initialization of the LUA virtual machine (loads when trueSpace starts up).
- Calling of the LUA script (loads when the shader is selected).
- Calling of an appropriate LUA script initialization (loads when the shader is selected).
- Calling of an appropriate LUA executable function (loads when the shader is used for an object).
- De-initialization of LUA and LUA shaders (during trueSpace shut down).

Shader Data Types

These parameters are specified as global variables in the shaders and are uniform during rendering. LightWorks-LUA shaders have the following parameters.

- RtFloat real number
- RtFloat4 4D vector (point, vector, color)
- RtFloat4x4 4D matrix

It is possible to use operators with mixed data types, for example.

```
RtFloat = float + RtFloat
or RtFloat = RtFloat / float
```

Initializing Shader Scripts

Currently, a LUA shader script must be placed in the trueSpace/Shaders/Material, and it must have the suffix “.lua”. The design of the scripting engine is similar to the implementation of custom shaders in the trueSpace API and every function of that API is available through LUA shader scripts.

The connection between trueSpace and LUA is made by the function *TSXShaderInitialize*. This function must return when called in the LUA script, and it is called automatically when a script is loaded.

- Type of shader text strings: COLOR, REFLECTION, DISPLACEMENT, TRANSPARENCY *or you can use the definitions* tsxSHCLASS_COLOR, tsxSHCLASS_TRANSPARENCY, tsxSHCLASS_REFLECTANCE, tsxSHCLASS_DISPLACEMENT.
- Name of scripting shader.
- Name of function, which is called when this shader is used for rendering.

For example:

```
MyTmp=tsxShaderInitialize("COLOR","CheckerBoard36","color_  
Checker36")
```

Shader Script Main Function

The main shader method is just another function whose name is specified during shader initialization. It is the entry point for shader execution and has no parameters, but it is allowed to use specific TSX functions that get and set LightWorks engine shader data (such as normal, color, etc.) for the current pixel. This function calls other methods of the shader to achieve the results.

```
function <function_name>(<param1>,<param2>,...<paramN>)  
...  
end
```

For example:

```
function rsxAdd(InputA, InputB, Output)  
    Output = InputA + InputB  
end
```

Lightworks Engine Shader Data

Lightworks defines the following data and fills them each time a shader is executed.

Name	Type	Description
alpha	RtFloat	Surface fractional coverage.
Ci	RtFloat4	Input colour.
Co	RtFloat4	Output colour.

dPds	RtFloat4	Texture space derivative vector with respect to s.
dPdt	RtFloat4	Texture space derivative vector with respect to t.
dPdu	RtFloat4	Parameter space derivative vector with respect to u.
dPdv	RtFloat4	Parameter space derivative vector with respect to v.
ds	RtFloat	Change in parameter space s-coordinate.
dt	RtFloat	Change in parameter space t-coordinate.
du	RtFloat	Change in parameter space u-coordinate.
dv	RtFloat	Change in parameter space v-coordinate.
E	RtFloat4	Camera eye ('from') point.
inside_solid	RtFloat	Inside solid flag.
I	RtFloat4	View direction vector of shading point.
modI	RtFloat	Length of I vector.
N	RtFloat4	Surface shading normal.
Ng	RtFloat4	Surface geometric normal.
P	RtFloat4	Shading point in local coordinate system.
Pw	RtFloat4	Shading point in world coordinate system.
s	RtFloat	Texture space s-coordinate.
t	RtFloat	Texture space t-coordinate.
Ti	RtFloat4	Input transparency.
To	RtFloat4	Output transparency.
u	RtFloat	Parameter space u-coordinate.
unitI	RtFloat4	Unitary version of I vector.
v	RtFloat	Parameter space v-coordinate.

These variables can be read by the shader and some can be written to by the shader.

The shader API provides a set of methods for accessing each of these parameters `TSXGet<Parameter>()` and `TSXSet<Parameter>(value)`. For example, for the `Ci` parameter you would use `TSXGetCi()` and `TSXSetCi(myLocalColor)`

All methods available in the LUA scripting engine are described in the document *trueSpaceCustom-ScriptingShaders.doc*. This document covers the trueSpace shader API, but function declaration and use is, except in a few cases, the same. Some methods in the API have parameters that also act as return values so these have been rewritten to move these parameters to the return value group.

Tutorial: Writing Shader Scripts – Chessboard Shader

The tutorial LUA sample files (*phong.lua* and *script.lua*) can be found in the trueSpace/Shaders/Material directory

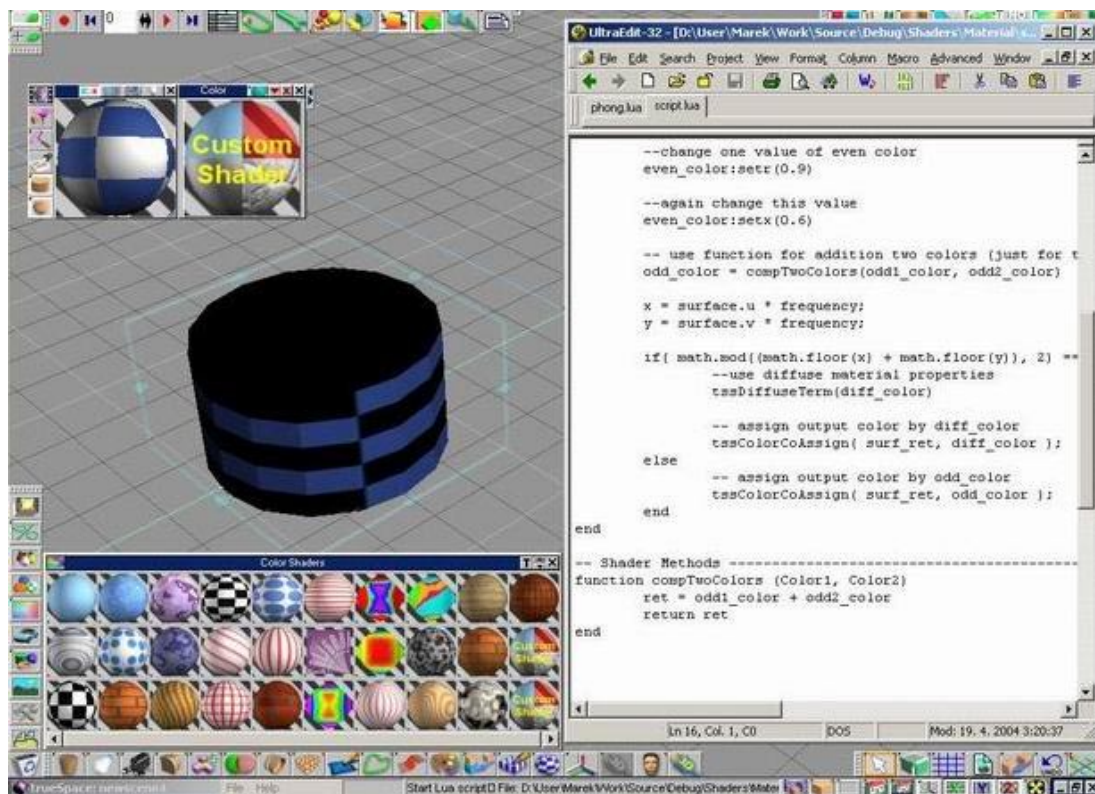


Figure1: New scripting shaders were added as custom shaders

First, your custom shader script must have an initialization method:

```
-- shader initialization -----
myTmp = tsxShaderInitialize(tsxSHCLASS_COLOR, "Shader-chessboard",
"color_ME_ColorShader")
```

Then you can define global variables.

```
even_color = RtFloat4.new(1.0, 0.5, 1.0)
odd1_color = RtFloat4.new(0.1, 0.0, 0.2)
odd2_color = RtFloat4.new(0.1, 0.3, 0.4)
diff_color = RtFloat4.new(0.0, 0.0, 0.0)
```

And now you can define the execute function

```
function color_ME_ColorShader ()
```

In this function, we first compute frequency

```
-- compute frequency
if (math.abs(size) < 0.0001) then
    frequency = 10000.0
else
    frequency = (1.0 / size)
end
```

Changing the value of the parameter is made by the method setXY()

```
--change one value of even color
even_color:setr(0.9)

--again change this value
even_color:setx(0.6)
```

After some modifications and changes you must call methods for changing the parameter Co in the surface structure by *new_color* value.

```
TSXSetCo ( new_color );
```

LUA/trueSpace Data Types

The following data types may be used in shader scripts created in LUA.

RtFloat

Create new data type

```
<VariableName> = RtFloat.new(<val1>)
<VariableName> = RtFloat.new() // Default init with val 0
```

Example:

```
val1 = 4.5
val1 = 4.5
val = val1 + val2 - 4 / 568
or val = val1:get() + val2 - 4 / 568
```

Standard methods for LightWorks data manipulation (e.g. tssPhongTerm) needed as input parameter *float* or *int* and not RtFloat. Therefore you must use *val1:get()* and not only *val1* for these functions

RtFloat4

Create new data type

```
<VariableName> = RtFloat4.new(<val1>, <val2>, <val3>, <val4>)
```

```
<VariableName> = RtFloat4.new() // Default init with val 0
```

To Get or Set value of RtFloat structure you can use two methods with different appendices

Set value of data type:

```
RtFloat4:set(<RtFloat4>)
```

Set 1. value of RtFloat4

```
RtFloat4:setx(<RtFloat>)  
RtFloat4:setr(<RtFloat>)
```

Set 2. value of RtFloat4

```
RtFloat4:sety(<RtFloat>)  
RtFloat4:setg(<RtFloat>)
```

Set 3. value of RtFloat4

```
RtFloat4:setz(<RtFloat>)  
RtFloat4.setb(<RtFloat>)
```

Set 4: value of RtFloat4

```
RtFloat4:seta(<RtFloat>)
```

Get value of data type:

```
<RtFloat> = RtFloat4:get()
```

Get 1. value of RtFloat4

```
<RtFloat> = RtFloat4:getx()  
<RtFloat> = RtFloat4:getr()
```

Get 2. value of RtFloat4

```
<RtFloat> = RtFloat4:gety()  
<RtFloat> = RtFloat4:getg()
```

Get 3. value of RtFloat4

```
<RtFloat> = RtFloat4:getz()  
<RtFloat> = RtFloat4:getb()
```

Get 4. value of RtFloat4

```
<RtFloat> = RtFloat4:geta()
```

Operators:

Add: +

```
<RtFloat4> = <RtFloat4> + <RtFloat4>
```

```
example:
val1 = RtFloat4:new(3,5,4,2)
val2 = RtFloat4:new(3,5,4,2)
val = val1 + val2
```

Sub: -

```
<RtFloat4> = <RtFloat4> - <RtFloat4>
```

Mul: *

```
<RtFloat4> = <RtFloat4> * <RtFloat4>
```

Div: /

```
<RtFloat4> = <RtFloat4> / <RtFloat4>
```

Dot product: dot

```
<RtFloat4> = <RtFloat4> dot <RtFloat4>
```

RtFloat4x4

Create new *data* type

```
<VariableName> = RtFloat4x4.new(
    <val11>, <val12>, <val13>, <val14>,
    <val21>, <val22>, <val23>, <val24>,
    <val31>, <val32>, <val33>, <val34>,
    <val41>, <val42>, <val43>, <val44> )
<VariableName> = RtFloat4x4.new() // Default init with val 0 and
diagonal values are 1
```

Set value of data type:

Get RtFloat4x4

```
RtFloat4x4:set(<RtFloat4x4>)
```

Set one value of RtFloat4x4

```
RtFloat4:set(colX, rowX, <RtFloat>)
```

Where XX are num of column and row of matrix

Example:

- set value in 3. column and 4: row by 5.6 value

```
val = RtFloat4x4.new()
val:set(3, 4, 5.6)
```

Get value of data type:

Get RtFloat4x4

```
<RtFloat4x4> = RtFloat4x4:get()
```

Get one value of RtFloat4x4

```
<RtFloat> = RtFloat4:get(X, X)
```

Where XX are num of column and row of matrix

Example:

- get value in 3. column and 4. row

```
val = RtFloat4x4.new()
```

```
flval = val:get(3, 4)
```

Operators:

Add: +

```
<RtFloat4x4> = <RtFloat4x4> + <RtFloat4x4>
```

Sub: -

```
<RtFloat4x4> = <RtFloat4x4> - <RtFloat4x4>
```

Mul: *

```
<RtFloat4x4> = <RtFloat4x4> * <RtFloat4x4>
```


Chapter 4

Scripting

4.1 Scripting in trueSpace

The trueSpace Script Editor provides a robust environment for creating procedural scripts to accomplish nearly any task. Scripts are interpreted, i.e. each command is executed by trueSpace as it is encountered, and can be edited while the program is running.

Through scripts, you can create, access, and interact with almost any trueSpace object from meshes to shaders. You can choose from several common scripting languages to create your script. trueSpace currently supports VBScript and Jscript, but new languages that are compatible with the Windows Script engine can easily be added.

There are two main methods of accessing the scripting capabilities of trueSpace. These are: Script Commands and Script Objects. Both are covered in this chapter.

Note: Scripts can also be used to create custom shaders in trueSpace using the LUA scripting language. Unlike Script Commands and Script Objects, these scripts are placed in a special HLSL Script object available only in the Material Editor. See “Writing Shader Scripts” in **DEVELOPER GUIDE CHAPTER 3: MATERIALS**.

Script Commands and Objects

Script Commands and Script Objects are very similar. Both are created by dragging the appropriate script object into the trueSpace Link Editor and then attaching input and output attributes and writing a script to act on those attributes. As noted previously you can create your script in languages including VBScript and JScript.

There are also important differences between Script Commands and Script Objects. You might think of Script Commands as ‘verbs’ or commands. They directly interact with other objects in your scene, requesting modifications, creating new objects, and they can act as a direct participant in control flow in the Link Editor. Script Objects can be thought of as ‘nouns’. They are stand-alone objects that manipulate their own data and state and respond to requests from other objects in the Link Editor and the system itself.

Scripting Possibilities

Scripts can help you accomplish a wide range of tasks. You can write a script to create your own primitive objects (a geosphere, for instance) or to modify an existing object (perhaps processing the objects vertices algorithmically to change its shape). You can use scripts to create behaviors for your objects (for example, causing a door to open when a certain condition is met) or to define how objects will interact (say, controlling a group of spheres to create a molecular simulation.)

In fact, there are so many possibilities that we can't even begin to name them all here. Think of it as a really large chest of tools and parts and start to imagine the things that you can build!

Getting Started with Scripting

In the following sections we will examine several scripting examples that you can use and modify to create your own scripts. Also, be sure to examine the many sample scripts that can be found in the trueSpace libraries. The best way to learn scripting is to explore and modify existing scripts.

To learn about native trueSpace objects, their attributes, and the methods that operate on them, please consult the Reference Guide. Finally, to learn what commands and facilities, such as libraries and built-in objects, are available in your chosen scripting language please visit the sites listed below.

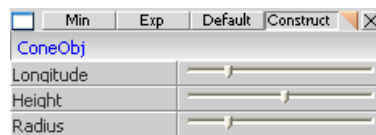
JScript, VBScript - <http://www.microsoft.com/scripting>

Also, if you have not already done so, you may want to read through section 2.12 of **ARTIST GUIDE CHAPTER 2: INTERFACE** for an overview of the Script Editor interface and a brief look at how scripts work in trueSpace before moving on to the more in-depth discussion in the rest of this chapter.

Tutorial: Modifying a Script

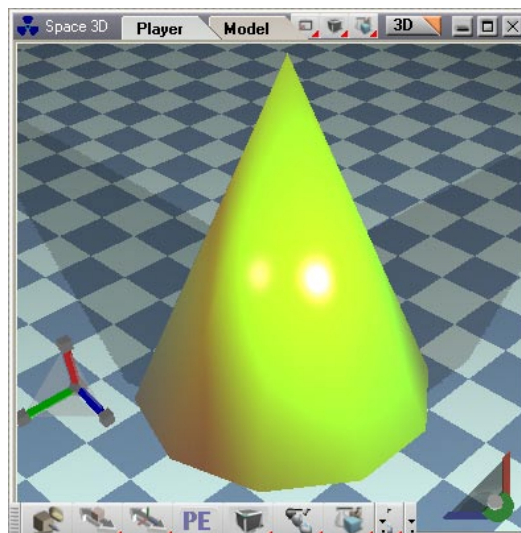
As a quick introduction to working with scripts in trueSpace, follow along with the example below to change the way the cone in the ConeObj object is created. In this example we will change just one line of script to dramatically alter the way the final cone geometry is laid out.

To get started open the Object library and click on the Script Objects panel to view its contents. Locate the ConeObj object and drag it into Space 3D in the Link Editor. You should see the object's panel, complete with three sliders for controlling the construction of the cone.



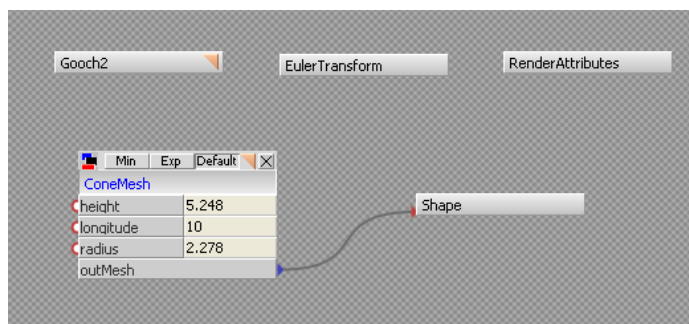
The ConeObj object as it appears in the Link Editor

The player window will show a greenish cone. Try moving the sliders on the ConeObj's panel. You should see the shape of the cone change as you move them back and forth. These sliders are hooked directly to attributes that the script is using to define how the cone mesh itself is constructed.



The Player window shows the results of the script as you change the sliders

When you are done experimenting with the sliders enter the ConeObj object by clicking on the orange enter triangle on the upper-right side of the object's panel. This will take you inside of the object, the contents of which are shown in the image below.

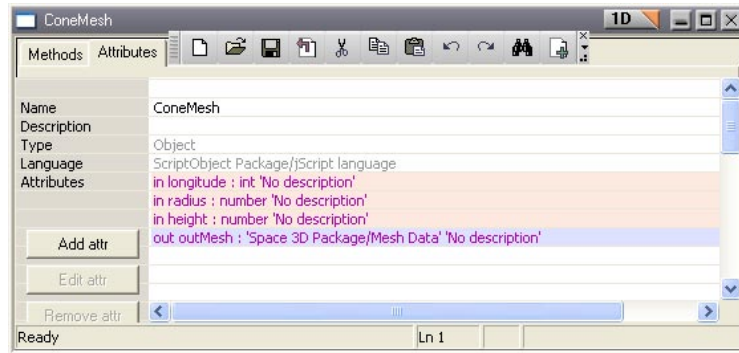


Inside the ConeObj we find a Script Object called ConeMesh

The object contains some additional objects that control how the object itself is displayed;— a Gooch2 material object, a Euler Transform object, and a RenderAttributes objects. You can explore these objects if you like but the most interesting object here is a Script Object called ConeMesh. You can see that this object has three input attributes: height, longitude, and radius and that these are exported, in fact these are the parameters that are being changed when you move the sliders on the main panel.

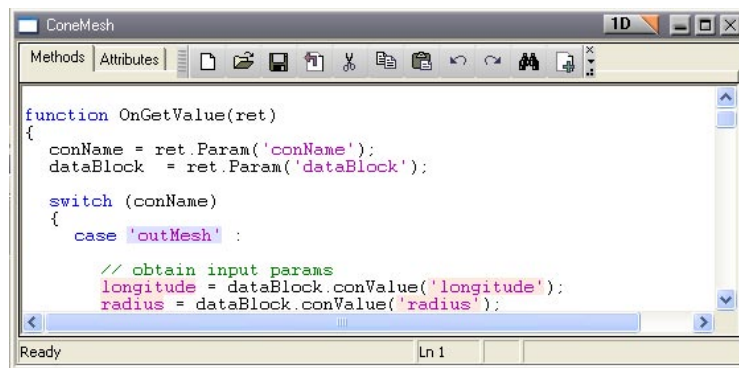
ConeMesh also has an output attribute called `outMesh` which is connected to the Shape object. `outMesh` is where the script inside ConeMesh deposits its finished product. The mesh data is then passed on to a Shape object allowing the cone to be rendered by trueSpace.

Enter ConeMesh by clicking on the orange enter. This will open the Script Editor. Now, click on the Attributes tab to see the Script Attributes view. This is another view of the attributes on the outside panel. Longitude, radius, and height are shown as input attributes and `outMesh` is shown as an output attribute.



ConeMesh's attributes as seen in the Script Attributes view

Now, click the Methods tab to open the Script Methods view. This will give us a view of the script in ConeMesh as shown in the image below.



The ConeMesh script in the Script Methods view

There are two functions in this script, and these functions are how trueSpace objects interact with the contents of the script. The first, and most important for our purposes, is `OnComputeOutputs()`. This function is executed any time another object or the system requests the script to provide a value at one of its output connectors. In the case of ConeMesh there is only one output attribute, `outMesh`.

As you might guess, then, this function is executed each time the system requests the script to provide a mesh 'value' for `outMesh`. This is very convenient, of course, because that is just the time that we would

want to create and alter the mesh's geometry. All of the complex looking code in this function is devoted to creating a mesh from vertices and faces and then providing that mesh data to the outMesh connector for the system to use. Take a closer look at the main part of this function, included below.

```
// obtain input params
longitude = params.conValue('longitude');
radius = params.conValue('radius');
height = params.conValue('height');
```

Here we simply retrieve the values of the other attributes to use in the script.

```
nVert = longitude + 2;
nFaces = 2 * longitude;
```

Now, we create two new variables and fill them with a calculation,. In this case we are determining how many vertices and faces the object will have based on the input attributes.

```
// define vertices
dV = System.CreateDO("Space 3D Package/Vertex Stream Data");
dV.SetNumVertices(nVert);
```

This creates a special type of data object, designed to hold vertex data, and then fills in the number of vertices calculated above.

```
// ... add vertices for bottom circle
angle = 0.0;
angleStep = 2.0 * Math.PI / longitude;

for (i = 0; i < longitude; i++)
{
    dV.x(i) = Math.cos(angle) * radius;
    dV.y(i) = Math.sin(angle) * radius;
    dV.z(i) = 0;

    angle += angleStep ;
}
```

Getting a little more complex now, in this section we have a loop that steps around the outside of a circle adding vertices to the vertex data object created earlier. The loop only executed longitude times, in other words, it only adds as many vertices around the circle as the value input for longitude.

```
dV.x(longitude) = 0;
dV.y(longitude) = 0;
dV.z(longitude) = height;
```

```
dV.x(longitude + 1) = 0;
dV.y(longitude + 1) = 0;
dV.z(longitude + 1) = 0;
```

These six lines add two more vertices to our vertex data object. One at the bottom center and one at the top center as defined by the input attribute height.

```
// define faces
dF = System.CreateDO("Space 3D Package/Triangle Vertices
Stream Data");
dF.SetNumTripleIndices(nFaces);
```

Now for the faces of the cone, here we create a special type of data object designed to hold face data, which is then told how many faces to expect.

```
// ... define bottom faces
for (i = 0; i < longitude; i++)
{
    dF.i(i) = longitude + 1;
    dF.j(i) = (i + 1) % longitude;
    dF.k(i) = i;
}
```

Back to complex, we now have a loop to add faces to the bottom of the cone. Faces are specified by listing the three vertices, in counter-clockwise order, around the edge of the face. You do not have to understand this to work with the script but be sure to explore the rest of this chapter and the reference guide for more information on advanced scripts, and working with special trueSpace data types if you are interested.

```
// ... define upper faces
for (i = 0; i < longitude; i++)
{
    dF.i(i+longitude) = longitude;
    dF.j(i+longitude) = i;
    dF.k(i+longitude) = (i + 1) % longitude;
}
```

Now we add the faces on the upper part of the cone between the top vertex and those around the circle we defined earlier.

```
// create mesh
dM = System.CreateDO("Space 3D Package/Mesh Data");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);
```

With the cone's vertices and faces fully defined, we create another special data object; this one designed

to hold mesh data and attach the vertex and face data to it.

```
params.conValue( 'outMesh' ) = dM;
```

We output the resulting data to outMesh and complete the function. It is a lot to take at a glance but, if you didn't quite understand something the first time, just take another read through the script listing. It sometimes helps if you just think of certain parts of the code as a mysterious black box; give it this value and it gives you this result back. You don't have to know how the math library sine function works; you just have to know that if you give it an angle it will return the sine of that angle!

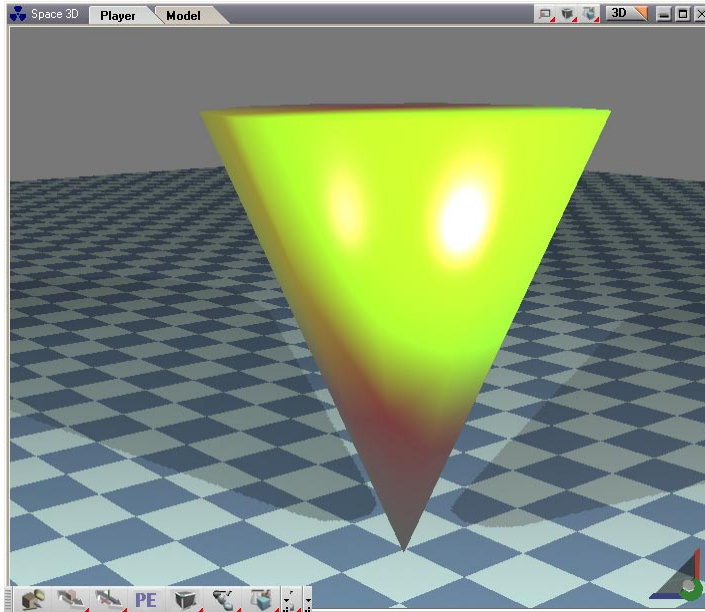
Upright cones are all well and good, but let's see what happens when we tweak some values. Find the section of the script that defines the vertices around the outer circumference of the cone. It should look like this:

```
// ... add vertices for bottom circle
angle = 0.0;
angleStep = 2.0 * Math.PI / longitude;

for (i = 0; i < longitude; i++)
{
    dV.x(i) = Math.cos(angle) * radius;
    dV.y(i) = Math.sin(angle) * radius;
    dV.z(i) = 0;

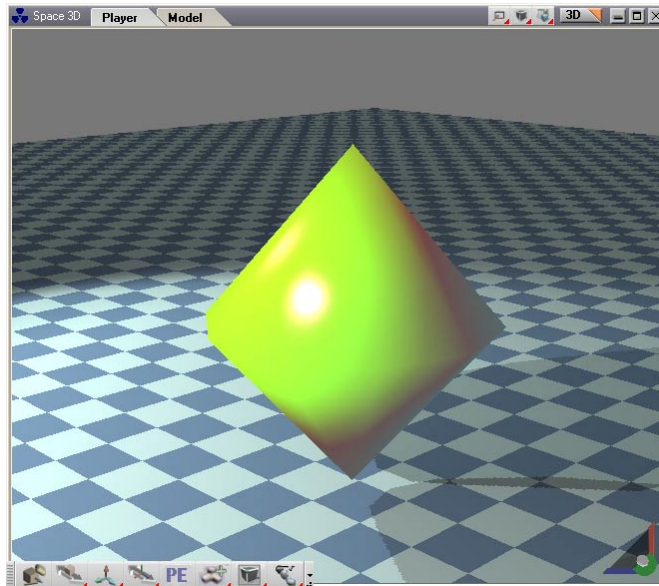
    angle += angleStep ;
}
```

Now change `dV.z(i) = 0;` to `dV.z(i) = height;` and press the Commit icon. You will probably see something like the image below in your player window. All we did here is change the z value of the vertices being created around the circumference to the cone so that they would be created at the top of the cone instead.



Changing just one value can have a major effect on the result

Now, try changing `dV.z(i) = 0;` to `dV.z(i) = height / 2;` and press the Commit icon. Again, we see another amazing result for such a small change. Here we created the vertices along the middle of the 'cone' mesh instead of at the bottom or the top.



Now our cone has become a top

Try changing other values to see what happens. If you break the script do not worry – just get a new one from the library and try again. When you feel that you want to move on and learn more about scripting read through the next two sections and begin exploring how Script Commands and Script Objects work!

4.2 Script Commands

Script Commands in trueSpace

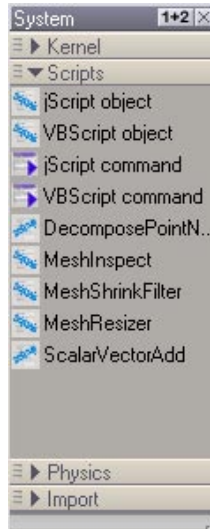
Script Commands are scripts that are executed *once*. They are activated by a Control signal in the trueSpace Link Editor or by direct request from another object. You can use a Script Command to create actions that react to a specific event, such as when you push a button or in response to a Control signal from another object.

Creating a Script Command Object

You create a Script Command object by dragging and dropping it from the System Library panel into the Link Editor. If the System Library panel is not visible you can open it by clicking the System Library icon.



The Script Command objects are identified by the word “command” after the name of the language that the object handles. For instance, if you want to create a JScript Script Command object simply left-click on the item named “JScript command” in the System Library panel, hold down the mouse button, and drag that object into the Link Editor.



The System library, Scripts panel contains starter script-based objects

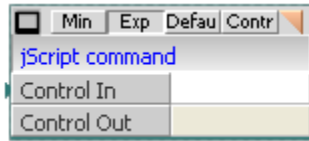
You can place the Script Command object at any level of hierarchy in the Link Editor but you will usually want to place it either within Space 3D or perhaps inside of another object with which you want the script to interact. Script Command objects can be Encapsulated within the Link Editor just like other objects (ref to Link Editor chapter for more information).

A JScript Command object as it appears in the trueSpace Link Editor is shown below. The object, by default, shows its Control aspect, which contains buttons for starting and stopping control flow in the link editor. Each time you press the Start button, the script in your JScript Command object will execute once.



A JScript Command object as it appears in the Link Editor

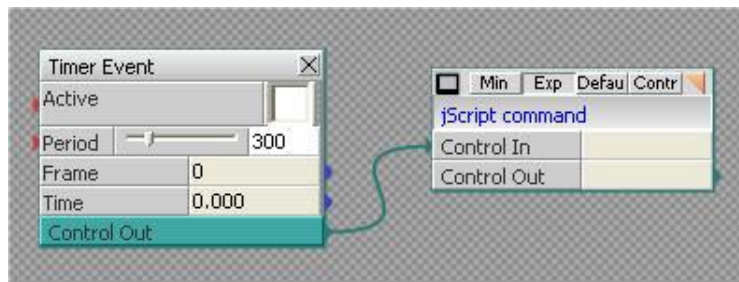
If you change to the Exp aspect you will see two items on the object's panel: an input connector called *Control In* and an output connector called *Control Out*. As you add Attributes to your Script Command object additional connectors representing those Attributes will appear on this panel, allowing you to connect them to other objects in the trueSpace Link Editor.



A JScript Command object with Exp aspect

Executing a Script Command

To activate a Script Command object you simply press the Start button on the panel when the object is in the Control aspect. You can also activate a Script Command by providing a control signal from another object connected to the objects Control In connector. This control signal could come from any number of Link Editor activity objects such as the one shown below.



Setting up a JScript Command to execute on a Timer Event

Alternatively you can execute a Script Command with a direct call from another Script Command or from an interface element, such as a button, using the `Activity.Run(<full name of command> command.)`

Inside the Script Command Object

To access the script and attributes of your Script Command object you will need to click the orange Enter icon in the upper right-hand side of the Script Command object in the Link Editor. Once you click the icon the Script Editor will open to either the Script Attributes view or the Script Methods view.

You can access the Script Methods view, if it is not visible, by clicking the Methods tab at the top-left of the window. This is where you will write your script code. You can access the Script Attributes view by clicking the Attributes tab. This is where you create input and output attributes for your Script Command object to use.

The Script Command Methods Window

The Methods window shows the textual portion of the script in your Script Command object. This window works the same as any simple text application, such as Windows Notepad. Simply type your script directly into the window, following the syntax guidelines of whichever script language you have chosen to create your script. Whitespace, i.e. tabs, spaces, and line spaces are ignored by the script interpreter.

There are several icons in the toolbar of this window, providing functions for editing (Cut, Copy, Paste, Undo, Redo, and Find), file management (New, Open, Save). These commands operate similar to those found in any text application. There are also two additional icons on the toolbar: Commit and Add Advanced Handler.

**Commit Icon****Add Advanced Handler Icon**

Clicking the Commit icon tells trueSpace to update the script with any changes you have made. It will also check for errors in your script at this time. Clicking the Add Advanced Handler Icon creates additional methods for your script that will allow you to handle more advanced interaction with the system.

The Script Command Attributes Window

The Attributes window is where you will create attributes or parameters that your Script Command object will use. If you have ever written a program using C, C++, Java, or even modern BASIC you are probably familiar with functions. Attributes are similar to the parameters passed into or retrieved from functions in these languages except in trueSpace these attributes are shared with other objects visually in the Link Editor.

By clicking the Add Attr button at the left of the window you can add an attribute to your Script Command object. When you click this button you will see the Add New Attribute dialog, pictured below.

The Add New Attribute dialog box

This dialog allows you create a new attribute by specifying its name, the direction the attribute will work (IN or OUT), its type (e.g. integer, number, or one of the special data types provided by trueSpace), and a text description that will tell users what the attribute is for and how it is used. The Control Flow checkbox allows you to create a control parameter, which can be triggered by or trigger other objects with control input and output connectors. The Registered checkbox declares the attribute to be one of the 'registered' data types that can be accepted by other objects like meshes and normals.

If your Script Command object already has attributes you can select them and use the Edit Attr or Remove Attr buttons at the left of the window to make changes or remove these attributes from your object. Please note that you will often be referencing the attributes you create here from within your script so changes made here will need to be reflected in your script. In other words, do not delete an attribute from the Attributes window and still refer to it from within your script or you will receive an error message.

Writing Script Commands

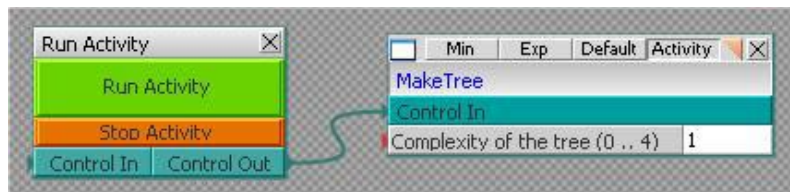
A trueSpace Script Command script has only one default function, called `Execute()`. This function is called each time the Script Command object receives an activation signal at its Control In connector or when the script is called directly by pressing the object's Start button or via a RunActivity command. You will place most of your script code within this function, though you may create additional functions and call them from within `Execute()`. Note that only script commands within the curly braces following `Execute()` will be carried out by the script interpreter, unless you call another function explicitly.

Forming complex activities from commands

If you want to create complex project, e.g. create game or define special behavior for your scene, you will most likely use commands as a 'building bricks' to achieve your goal, and you have to connect these bricks to one big working entity.

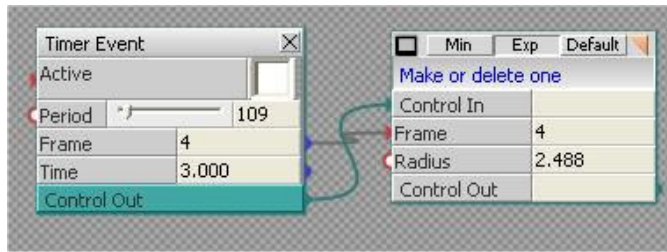
We can outline here several most frequently used schemes how to form final activity:

- **One shoot activity**



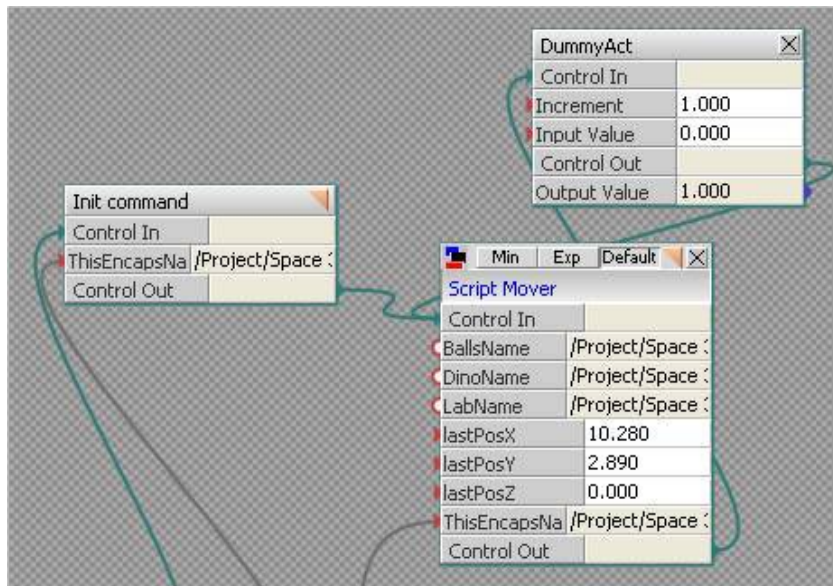
One Shoot activity is the simplest form. It consists of only one or few commands (bricks) connected together without loops. You can run it; it performs action and stop immediately. (Example: MakeTree command connected to the Run Activity starter)

- **Timer based loop**



The Timer based loop is the more complex form. It is useful when you want to create a loop with exact timing. Look at the example taken from the Crazy circle example (found in Activities library), the Timer event component fires a connected command (Make or delete one) each period of time, till the user stops it. The benefit of this scheme is that you can control the speed of executing by adjusting timer period

- **Infinity loop**



Alternatively you can form a loop without a timer simply by forming a loop on commands. In this example (taken from the PacMan demo), the Script Mover command is connected to DummyAct and back to make a loop. The user can stop the infinity loop by pressing the Stop button on the main control panel of the game. The benefit here is that activity is running in top speed, and you can also form complex behaviors with decision blocks, states etc.

Note: To control an execution speed of this kind of loops, or ensure similar speed on various machines, it is strongly recommended to connect the Pause Activity element to the loop. This way you can create CPU-friendly loops with easily controlled execution speed.

Tutorial: Writing a Simple Script Command

To start learning about writing Script Command scripts we will go through the steps required to create a very basic Script Command object in trueSpace. This Tutorial will show you how to use a script command to change the orientation of an object, in this case, to turn the needle of a compass to face a random direction when you click a button.

To get started create a new scene and drag-and-drop a new JScript Command object into Space 3D in the Link Editor. You should always name your objects something descriptive. To do this, click on the JScript Command object's drop-down menu box and select Rename Object from the menu. Call it something simple like Random Direction.

Now we will create an attribute. Left-click the Enter icon on the Random Direction object to access the Methods and Attributes windows. Then, click on the Attributes tab to view the Attributes window. Click on the Add Attr button and fill in the entries to match the image below. When you are done press the OK button.



The Add New Attribute dialog box filled in for oDirection

This will create a new attribute called `oDirection`, of the type `number` (which can represent any real number). Attributes should also have descriptive names, such as `direction`, `location`, and so on so that users of your object can easily tell what they are used for. In this case we also prefaced the name with 'o' to remind us that it is an output attribute while we are writing the script.

Now, we need to write a script that will fill `oDirection` with a value representing an angle from 0 to 359 degrees. To do this, click on the Methods tab to access the Methods window. In the text area type in the following script:

```
// Execute
// Called to execute the command
function Execute(params)
{
```

```

// Get a random value between 0 and 359
value = Math.random() * 359

// Pass on output value
params.conValue('oDirection') = value;
}

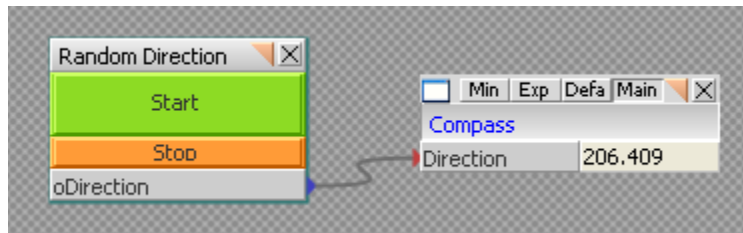
```

This script simply accesses the JScript Math library to return a pseudorandom number between 0 and 1 and then multiplies that number by 359. This gives us a random number in the range of 0 and 359, which we then send to `oDirection` using the `ret.Param()` statement. So, each time `Execute()` is called `oDirection` will be filled with a random value ranging from 0 to 359, which just happens to be the number of degrees in a full circle, a handy thing to know of course.

Now, our simple script is completed. Click the Commit icon at the top of the Methods window to tell trueSpace to update the script and check for any errors. If you have entered the script as shown above you should not have any problems. If you do receive an error just check your script against the one above. trueSpace should give you a hint as to where the problem lies.

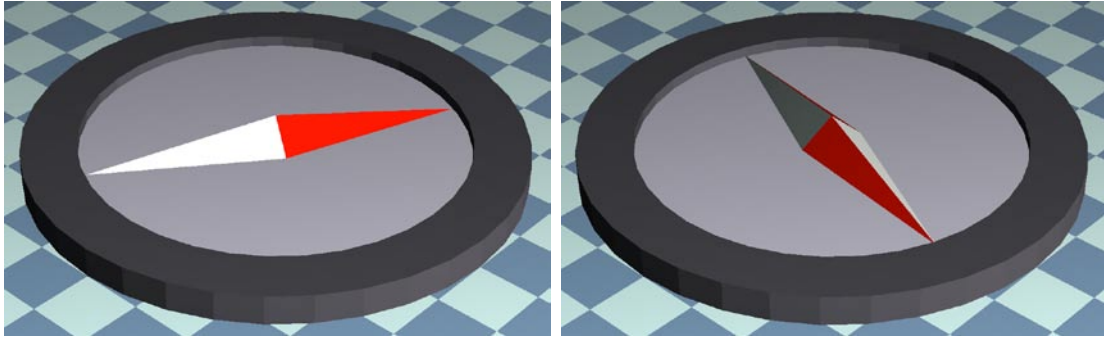
If your script checks out just click the orange triangle to exit the script and re-enter the Link Editor. Now, change to the Control aspect of your Random Direction object and edit the panel to include the newly created `oDirection` attribute.

After that, drag-and-drop the Compass object into the Link Editor. You can find this object under the Objects Library in the Tutorial Objects panel. Next, connect the `oDirection` connector to the Direction connector on the Compass object. The final configuration is shown below.



Random Direction connected to Compass in the Link Editor

Assuming you have connected everything correctly you should be able to left-click the Run Activity button on the Run Activity object and watch the needle of the Compass object change each time to a random direction. Your player window should look like the image below.



The needle points to a random direction when you execute Random Direction

While this example is not particularly useful, in the real world you can easily expand on it to do more useful work within the `Execute()` method. A more-advanced example might be calculating X, Y, and Z values for a spherical orbit using Kepler's formulae. You might need to add additional input and output parameters, but doing this is just a matter of following the steps above.

Tutorial: Creating Advanced Script Commands

We can build on what has been learned in the previous tutorial by creating a Script Command object and an activity loop to make an object perform a slightly more advanced trick – move in a complete circle. To do this we will just use basic trigonometric formulae to calculate points on a circle based on angle and radius, leaving the more complex Keplerian method or orbit calculation as an exercise for the user.

First off, as with the last tutorial, create a new scene and drag-and-drop a new JScript Command object into Space 3D in the Link Editor. Rename it something like Orbit Calculator.

We will use four attributes in our script: `iRadius`, `iAngle`, `oX`, and `oY`. To create these, left-click the Enter icon on the Orbit Calculator object to access the Methods and Attributes windows and click on the Attributes tab to view the Attributes window. Use the Add Attr button to create entries for each of these attributes as shown below.

```
in iRadius : number 'Radius of orbit'
in iAngle : int 'Current angle along orbit'
out oX : number 'Calculated X position of planet'
out oY : number 'Calculated Y position of planet'
```

The integer (int in the Add New Attribute dialog) attribute `iAngle` will be used to input the current angle of rotation for a planet object. The number attribute `iRadius` will control the size of the circle traveled by the planet object. Finally, the two integer output attributes `oX` and `oY` will contain the X and Y values that we will calculate to specify the location of the planet object as it travels around its orbit.

Now that we have all of our attributes created it is time to write the script. The basic procedure is simple: get `iRadius` and `iAngle` and calculate the planet object's position along the arc of the circle using the built-in `JScriptMath` library functions. The general formula for this is $x = \sin(\text{angle}) * \text{radius}$ and $y = \cos(\text{angle}) * \text{radius}$. The completed script is shown below.

```
// Execute
// Called to execute the command
function Execute(params)
{
    // Get radius and angle
    radius = params.conValue('iRadius');
    angle = params.conValue('iAngle') * (Math.PI / 180);

    // Calculate new position
    x = Math.sin(angle) * radius;
    y = Math.cos(angle) * radius;

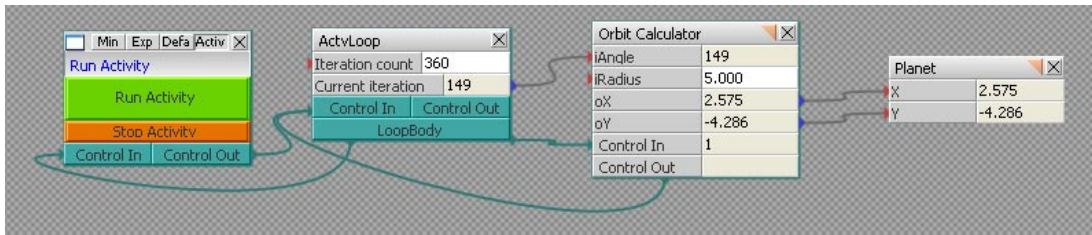
    // Output calculated value
    params.conValue("oX") = x;
    params.conValue("oY") = y;
}
```

One item of note is that we must convert `iAngle` to radians before we can use it with the `JScriptMath` library functions. To do this we multiply it by π ($\approx 3.1415926\dots$) divided by 180, because there are 2π radians in a circle. The value of π is provided to us via `Math.PI`. After the conversion we can perform our calculation and output the X and Y coordinates for the position of the planet.

Now, there is a bit more to do before we can see our script in action. In the previous tutorial we use the Start button on the Script Command's panel to execute the script. This time we will use a Run Activity to power our object and we will add an `ActvLoop` object, both of which can be found in the Activity library.

The `ActvLoop` object will generate a continuous control signal for a certain number of iterations, specified by the Iteration count attribute. Drag and drop a Run Activity object and an `Actv Loop` object into Space 3D in the Link Editor.

You will also need to add in the Planet object from the Tutorial Objects panel of the Objects Library. Now that you have all four objects placed in the Link Editor you will need to hook up their connectors as shown in the diagram below.



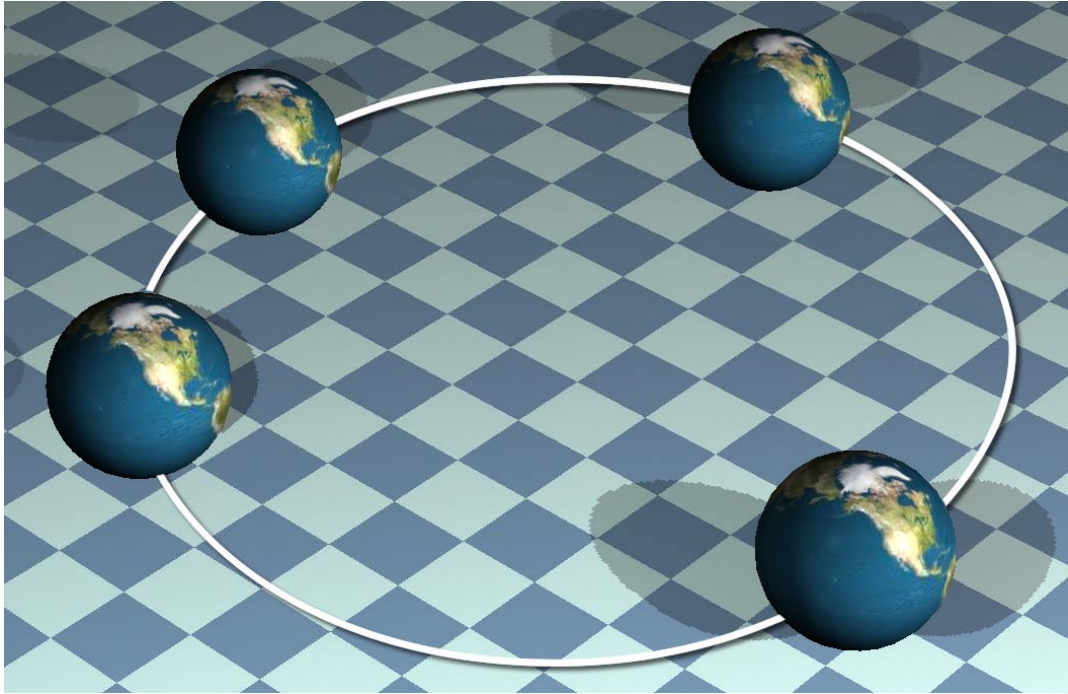
The completed activity loop in the Link Editor

Briefly, connect the `Current iteration` connector on the `ActvLoop` object to the `iAngle` connector on the `Orbit Calculator` object. Then, connect `oX` and `oY` on the `Script Command` to `X` and `Y` on the `Planet` object. Now we turn to control flow. First connect `Control Out` on `Run Activity` to `Control In` on `ActvLoop`. Then connect `LoopBody` to `Control In` on your `Orbit Calculator` object. After that, close the loop by connecting `Control Out` on `Orbit Calculator` back into `Control In` on the `ActvLoop` object. Finally, to make the activity repeat when it is done, connect `Control Out` on `ActvLoop` to `Control In` on `Run Activity`.

As a last step, type in 360 in the text box for `Iteration count` on the `ActvLoop` object and then type in a small value in the text box for `iRadius` on your `Orbit Calculator` object – we used 5.0. If you make `iRadius` too large the planet may be transported outside of your view range.

If you have followed the preceding steps correctly you should be looking at a pretty complex network of blue-green control connections, but do not worry, we will follow the flow step-by-step to see how simple the final product really is! First, when you click the `Run Activity` button on the `Run Activity` object a control signal is sent to the `ActvLoop` object, starting the process. `ActvLoop` starts its iterations at 0 and counts up to 360 each time.

Now, after receiving the control signal, `ActvLoop` sends a control signal of its own via `LoopBody`, which triggers the `Execute()` method of your `Orbit Calculator` script. When it executes the script requests values for `iRadius` and `iAngle` and calculates values for `oX` and `oY`, which are sent to the `Planet` to transform its location. When the script is complete it generates its own control signal which is sent back to `ActvLoop` starting the process over again. Finally, when `ActvLoop` has completed its 360 iterations it sends a control signal back to `Run Activity` starting the entire loop over from zero.



The planet object follows a circular path when you click the Run Activity button

With this basic loop control apparatus you are armed with the tools to create some pretty sophisticated loop-based animations and simulations.

Tutorial: Script Commands in More Complex Objects

To see how Script Command objects can be used in more complex objects we will take a look at the inner workings of the Terrain System object. You can find this object in the Objects Library under the Script Objects Panel. The icon is shown below, simply drag-and-drop this object into the Space 3D in the Link Editor to get started.

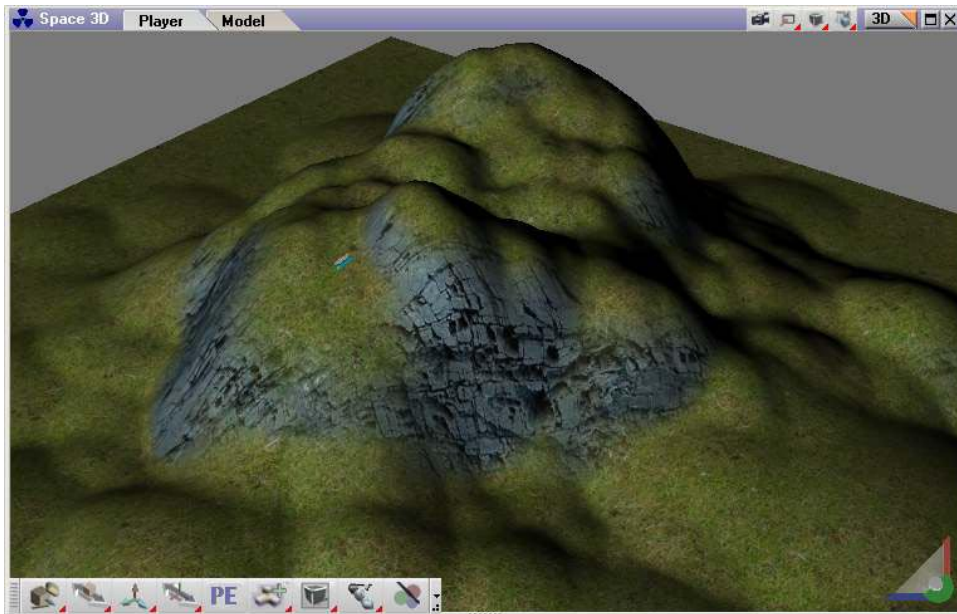


You should see the panel pictured below in the Link Editor. This panel has two buttons, several sliders, and a check box that control the functioning of the object.



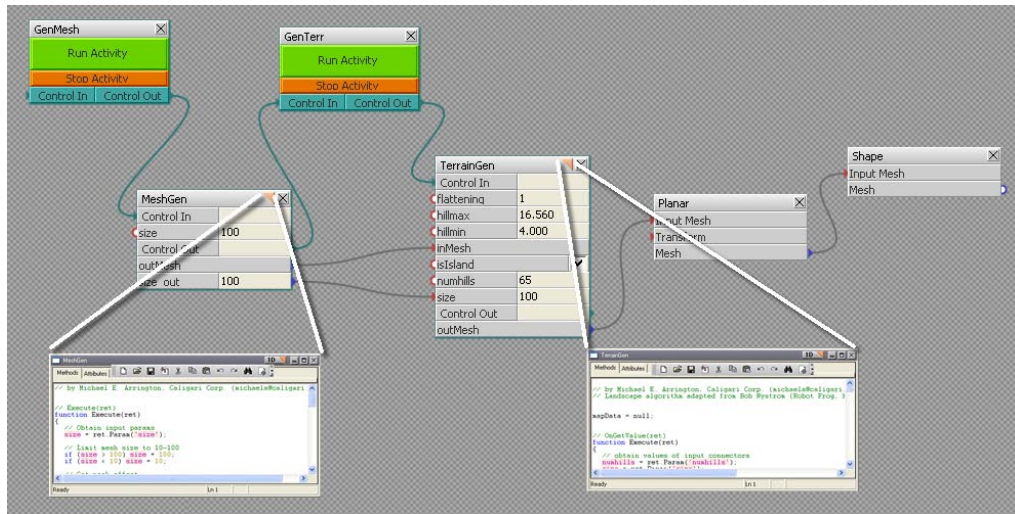
The Terrain System object panel

The image below shows the Player Window view of the Terrain System objects with default values. You can see that the object creates a mesh that resembles rolling hills in shape and coloration.



The terrain as shown in the Player view

So, how does it all work? To find out, enter the Terrain System object by clicking the orange triangle on the upper right-hand side of the panel. Inside you should see several objects including a Euler Transform, a modified LayeredPlastic shader called TerrainMaterial, and a RenderAttributes object. These control the position, orientation, size, and position of the object and are interesting in their own right but the real meat of the object is the collection of objects pictured in the diagram below.



Inside the terrain object are two Script Commands, shown zoomed

Here you see two Run Activity objects, two JScript Command objects, a default UV coordinate mapper object and a Shape object. The flow is actually quite simple. When the user presses the Make/Remake Mesh button on the main object panel a control signal is sent from the GenMesh Run Activity object, which triggers the script in the MeshGen Script Command. This script, shown below, creates a square grid mesh object based on the `size` parameter, passed in from the slider on the main panel.

```
// Execute(params)
function Execute(params)
{
    // Obtain input params
    size = params.conValue('size');

    // Limit mesh size to 10-100
    if (size > 100) size = 100;
    if (size < 10) size = 10;

    // Get mesh offset
    var nHalf = size / 2.0;

    // Define vertices
    var nVerts = (size + 1) * (size + 1);
    dV = new ActiveXObject("Space3D.RdVertexStream");
    dV.SetNumVertices(nVerts);

    // Define faces
    var nFaces = size * size * 2;
```

```

dF = new ActiveXObject("Space3D.RdTriangleStream");
dF.SetNumTripleIndices(nFaces);

// Create verts
for (y = 0; y <= size; y++)
{
    for (x = 0; x <= size; x++)
    {
        dV.x(y * (size + 1) + x) = x - nHalf;
        dV.y(y * (size + 1) + x) = y - nHalf;
        dV.z(y * (size + 1) + x) = 0.01;
    }
}

// Create faces
var i = 0;
for (y = 0; y < size; y++)
{
    for (x = 0; x < size ; x++)
    {
        // Create first tri in grid square
        dF.i(i) = (y) * (size + 1) + (x);
        dF.j(i) = (y + 1) * (size + 1) + (x + 1);
        dF.k(i) = (y + 1) * (size + 1) + (x);
        // Create second tri in grid square
        dF.i(i + 1) = (y) * (size + 1) + (x);
        dF.j(i + 1) = (y) * (size + 1) + (x + 1);
        dF.k(i + 1) = (y + 1) * (size + 1) + (x + 1);
        i = i + 2;
    }
}

// Create the mesh and output it
dM = new ActiveXObject("Space3D.RdMesh");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);
params.conValue('outMesh') = dM;

// Pass on output values
params.conValue('size_out') = size;
}

```

The script in MeshGen retrieves the value passed in for `size` and checks to make sure that it is between 10 and 100. Then it creates vertex and triangle stream objects (see the reference guide for more information on these objects and how they work) and then adds the vertices and triangles that make up the mesh.

When the information has been assembled these streams are attached to a new mesh data object and passed out to the `outMesh` connector. The `size` parameter is also passed on at this time.

Back out of the script we can see that the control signal is also passed on from the MeshGen Script Command object to the GenTerr Run Activity object, forcing that object to pass on the control signal to activate the script in the TerrGen Script Command object. You can see that this object has a number of attributes which retrieve their values from sliders on the main object panel. Two attributes, `inMesh` and `size`, are retrieved from the MeshGen Script Command object.

When TerrGen receives the control signal the following script is executed.

```
mapData = null;
```

```
// Execute(params)
function Execute(params)
{
    // obtain values of input connectors
    numhills = params.conValue('numhills');
    size = params.conValue('size');
    hillmin = params.conValue('hillmin');
    hillmax = params.conValue('hillmax');
    isIsland = params.conValue('isIsland');
    flattening = params.conValue('flattening');

    // Get input mesh and vertex stream
    dM = params.conValue('inMesh');
    nVert = dM.GetNumVertices();
    dV = dM.GetVertices();

    // Initialize map data array and generate terrain
    mapData = new Array(nVert);
    GenTerrain();

    // Modify z value of vertices based on map data array
    for( i = 0; i < nVert; i++)
    {
        dV.z(i) = 0.1 + mapData[i];
    }

    // Reattach vertex stream and output the mesh
    dM.AttachVerticesStream(dV);
    params.conValue('outMesh') = dM;
```

```
}
```

First off the script defines `mapData` at the global scope, i.e. outside of any function. This is so that the variable can be used throughout the script (see the reference guide for JScript for more information about variable scope). When the `Execute()` function is called the script first obtains all of the data it will act on, including the mesh previously created by `MeshGen`. It also uses a couple of the mesh object's methods to determine the number of vertices that are attached to that mesh and to get a copy of those vertices.

Next the script creates an array in the variable `mapData`, sized so that there is an entry for each vertex in the provided mesh. Then the script calls `GenTerrain()` to fill that array with terrain data (`GenTerrain()` and other terrain generation functions are not listed here but you can examine them by opening the Terrain System object and entering the `TerrGen Script Command` object.)

After generating the data, the script iterates through each vertex in the mesh and changes the height (z coordinate) to the value stored in the `mapData` array. Finally the new vertex stream is attached to the mesh and it is passed out as `outMesh`.

Back in the Link Editor the finished, modified mesh has a planar UV mapping applied and is then fed into a Shape object so that it can be rendered by trueSpace. Following all of these steps the Player window shows a nicely formed terrain, complete with rocky hillsides and rolling grassy plains.

You may have noticed that the description above did not mention the second button on the main object panel – `Generate Terrain`. When you press the `Make/Remake Mesh` button the object runs through the entire sequence above: creating a mesh, assigning data, and outputting the mesh. When you press the `Generate Terrain` button the process skips over the mesh creation phase, by sending a control signal to `GenTerr`. This is so that a new mesh only needs to be created if you want to change the size of the mesh, not each time one of the terrain-specific parameters is changed.

Though it appears complex at first glance, the Terrain System object actually has a simple, understandable flow and can be used as a basic model for a number of different mesh creation tasks with a few simple modifications. Try changing some of the parameters or altering the algorithms used for mesh creation to see what other possibilities exist. The same basic setup could be used, for instance, to create a spherical mesh covered in craters or to create a city-like area covered with buildings. Your imagination is the only limit.

4.2 Script Objects

Script Objects in trueSpace

Script Objects are scripts that create *persistent* objects, reacting to changes by input and output parameters as they happen. You should use a Script Object when you want to have a script that regularly interacts with your scene or activity as the values that are provided to it change.

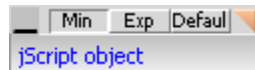
Creating a Script Object

You create a Script Object the same way you create a Script Command object, by dragging and dropping it from the System Library panel into the Link Editor.

A Script Object is identified by the word “object” after the name of the language that the object handles. For instance, if you want to create a JScript Script Object simply left-click on the item named “JScript object” in the System Library panel, hold down the mouse button, and drag that object into the Link Editor.

As with Script Command objects you can place your Script Object at any level of hierarchy in the Link Editor but you will usually want to place it either within Space 3D or perhaps inside of another object, one with which you want the script to interact. Script Objects can be encapsulated within the Link Editor just like other objects (ref to Link Editor chapter for more information}.

A JScript Script Object as it appears in the trueSpace Link Editor is shown below. Notice that there are not any items on its panel when the object is created. As you add Attributes to your Script Object additional connectors representing those Attributes will appear on the panel.



A JScript Script Object in the as it appears in the Link Editor

Inside the Script Object

As with a Script Command object you can access the script and attributes of your Script Object by clicking the orange Enter icon in the upper right-hand side of the Script Object in the Link Editor. Once you click the icon the Link Editor will show either the Methods window or the Attributes window.

You can easily switch between these two views by clicking the Methods and Attributes tabs at the top-left of the window. The Methods window is where you will write your script code and the Attributes window is where you create input and output attributes for your Script Object to use. For more on these windows please see the script command section above.

Writing Script Object Scripts

A trueSpace Script Object has one basic method that controls the way the script interacts with other objects in the Link Editor. The method, also called a handler because it handles requests for interaction, is `OnComputeOutputs(params)`. Below you will find a quick overview of this method. Please note that all of the samples in this section are written in JScript.

OnComputeOutputs(params)

This method is called whenever the value of any output connector is requested by another object in the Link Editor. You can compute values of all output connectors at once here using input connector values. You have access to all connector values through the generic `params` parameter.

```
function OnComputeOutputs (params)
{
    inValue1 = params.conValue('inValue1');
    inValue2 = params.conValue('inValue2');

    params.conValue('outValue1') = inValue1 + 1;
    params.conValue('outValue2') = inValue2 + 1;
}
```

Our example Script Object has just two output values, `outValue1` and `outValue2`, showing that you can define the basic behavior of your object with just a few lines of code!

Advanced Handlers

For advanced users there are more handlers available to define more complex behavior. These advanced handlers include `OnSetValue(params)`, `OnDefaultValue(params)`, and `OnCreate(params)`. Below you will find a quick overview of these methods. Please note that all of the samples in this section are written in JScript.

OnSetValue(params)

This method is called whenever the value of one of the Script Object's input connectors is changed. You can use this method to react to changing values, perhaps by making sure that the value stays within a certain range. As with `OnComputeOutputs()` you can access two values from within this method, both provided through the generic `params` parameter. These are `conName` and `dataBlock`. Below you see a basic implementation of this method.

```
function OnSetValue (params)
{
    conName = params.Param('conName');
    dataBlock = params.Param('dataBlock');

    switch (conName)
    {
        case 'inValue1' :

            // perform some action
            if (dataBlock.conValue('inValue1') > 10)
                dataBlock.conValue('inValue1') = 10
    }
```

```

        break;

    default :
        break;
}
}

```

In this example implementation we are checking only one input attribute/connector, `inValue1`. If `conName` is `inValue1` then the statements directly under case `'inValue1'` : will execute. In this case we check to see if the new value is greater than 10 and, if so, set it equal to 10 again.

OnDefaultValue(params)

This method is used to set the default values for specific connectors. It is called when the Script Object is first created or whenever the default value for a connector is needed. Two values are provided through the generic `params` parameter. These are `conName` and `vtData`. A simple implementation of this method is shown below.

```

function OnDefaultValue(params)
{
    conName = params.Param('conName');

    switch (conName)
    {
        case 'height' :
            params.Param('vtData') = 3.0;
            break;
        case 'longitude' :
            params.Param('vtData') = 15;
            break;
        case 'radius' :
            params.Param('vtData') = 1.5;
            break;
    }
}

```

The parameter `conName` provides the name of the connector for which a default value is being requested. In this case there are three connectors (`height`, `longitude`, and `radius`) and we have used `switch-case` statements to find out which connector is being requested. Then we set `vtData` equal to our chosen default value.

You only need to implement method in your Script Object if you wish your connectors to have a set of default values when the object is created. The implementation above, for example, provides the basic construction parameters for a cone object. When the Script Object is created `height` will be set to 3, `longitude` will be set to 15, and `radius` will be set to 1.5.

OnCreate(params)

This method is called only once, when the Script Object is created and initialized. Perform any initialization for your script here. For instance, you could create a variable and set it equal to zero here, then add one to it each time `OnComputeOutputs()` is executed. Create any variables that you would like to use throughout your object here.

This method also sets the dependencies for the object's connectors. You can access one value from within this method, provided through the generic `params` parameter. This is the `connectors` parameter, as shown in the default implementation of this method below.

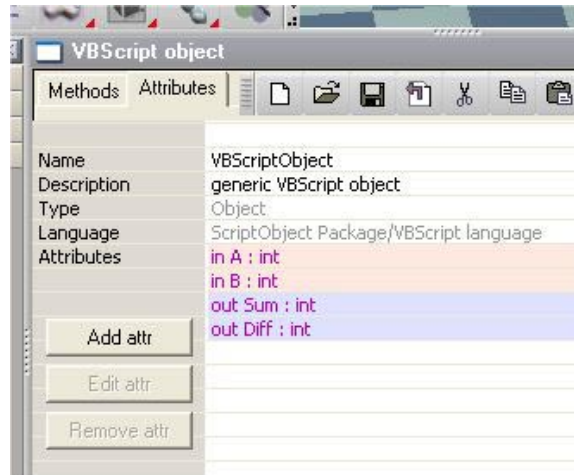
```
function OnLoadDefault(params)
{
    connectors = params.Param('connectors');

    // remove following line to set dependencies manually
    connectors.SetDefaultDependency();
}
```

Tutorial: Creating a Simple Math Object

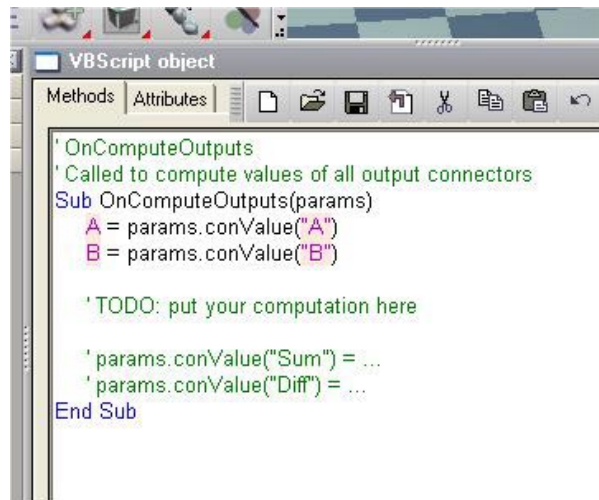
In this tutorial we will create very simple object to perform a basic calculation with only two inputs and two outputs. Though this script object is as simple as possible you should be able to see how this example can be applied to much more useful objects. Please note that this example is written in VBScript.

First open the System Components library and drag a VBScript Object into the Link Editor. Then enter the object by clicking the orange triangle on its panel to open the Script Editor. Click the Add Attr button and define two input attributes (A and B) as integers and two output attributes Sum and Diff as integers.



The math object will need four attributes

Now switch to the Methods tab and take a look at the script generated for you by trueSpace. You should always try to define all of an object's attributes first before beginning to work on the script to take advantage of the built-in script generator. You can, of course, always add attributes later but it is easier to let trueSpace write a skeleton script for you.



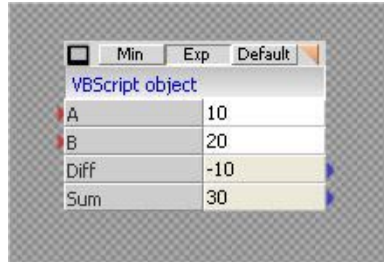
The math object script generated after you add attributes

Now, manually complete the OnComputeOutputs() method to figure the sum and difference of the values at attributes A and B, with these two lines. Hint, if you remove the apostrophe (') before the lines you will only have to replace the ellipses with 'A + B' and 'A - B'

```
Params.conValue("Sum") = A + B
```

```
Params.conValue("Diff") = A - B
```

That's all you need to do to define the Script Object. Now, press the orange exit triangle to return to the Link Editor. Switch the object to the Exp aspect and then change the values of A and B to see the results.

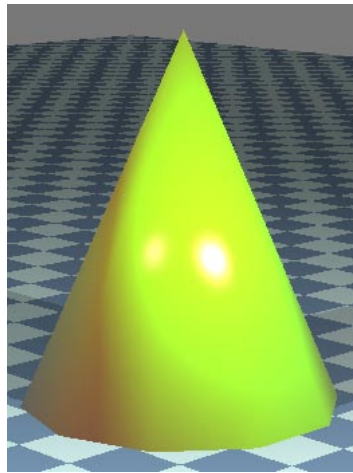


The final math object in the Link Editor

Tutorial: Examining a Simple Script Object

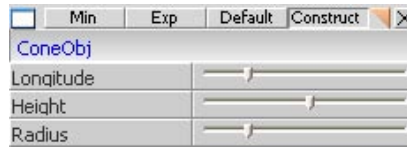
For this tutorial we will take a look at one of the more basic Script Objects, the ConeObj. This object creates a simple cone and provides front panel access to attributes including longitude, height, and radius using slider controls.

The ConeObj can be found in the Objects library under the Script Objects panel. Simply drag the item into Space 3D in the Link Editor. When completed you should see a cone in the Player view, as shown in the image below.



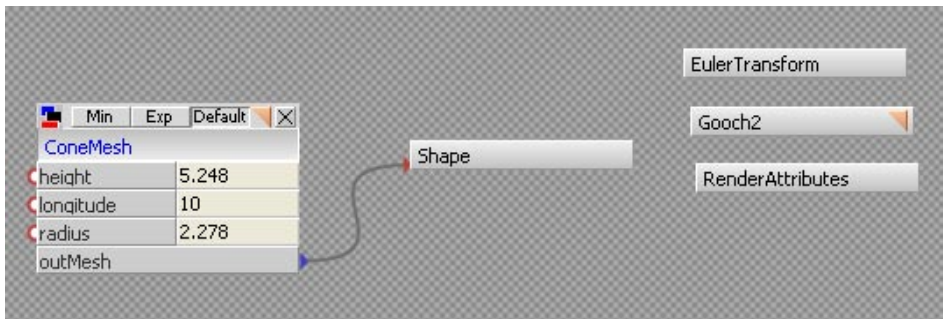
The output of ConeObj as seen in the Player view

In the Link Editor the ConeObj panel will appear as in the picture below. Try moving the sliders to see the effect they have on the shape and geometry of the cone in the Player window.



ConeObj as seen in the Link Editor

Now, click the orange triangle at the upper left of the ConeObj panel window in the Link Editor to enter the object. The ConeObj contains a Script Object called ConeMesh, along with a transform, a material, and a Render Attributes object as shown in the image below.



The interior of ConeObj as seen in the Link Editor

You can see that the ConeMesh script exports three attributes (height, longitude, and radius), and these are the values affected by the sliders on the main object panel. It also exports a mesh to the Shape object so that trueSpace can render the cone mesh.

Entering the ConeMesh script reveals the script, containing implementations of two basic methods, or handlers. The important method for this object is `OnComputeOutputs()`, shown below:

```
function OnComputeOutputs (params)
{
    // obtain input params
    longitude = params.conValue('longitude');
    radius = params.conValue('radius');
    height = params.conValue('height');

    nVert = longitude + 2;
    nFaces = 2 * longitude;

    // define vertices
    dV = System.CreateDO("Space 3D Package/Vertex Stream Data");
    dV.SetNumVertices(nVert);
```

```
// ... add vertices for bottom circle
angle = 0.0;
angleStep = 2.0 * Math.PI / longitude;

for (i = 0; i < longitude; i++)
{
    dV.x(i) = Math.cos(angle) * radius;
    dV.y(i) = Math.sin(angle) * radius;
    dV.z(i) = 0;

    angle += angleStep ;
}

// ... add top and bottom vertices
dV.x(longitude) = 0;
dV.y(longitude) = 0;
dV.z(longitude) = height;

dV.x(longitude + 1) = 0;
dV.y(longitude + 1) = 0;
dV.z(longitude + 1) = 0;

// define faces
dF = System.CreateDO("Space 3D Package/Triangle Vertices Stream
Data");
dF.SetNumTripleIndices(nFaces);

// ... define bottom faces
for (i = 0; i < longitude; i++)
{
    dF.i(i) = longitude + 1;
    dF.j(i) = (i + 1) % longitude;
    dF.k(i) = i;
}

// ... define upper faces
for (i = 0; i < longitude; i++)
{
    dF.i(i+longitude) = longitude;
    dF.j(i+longitude) = i;
    dF.k(i+longitude) = (i + 1) % longitude;
}

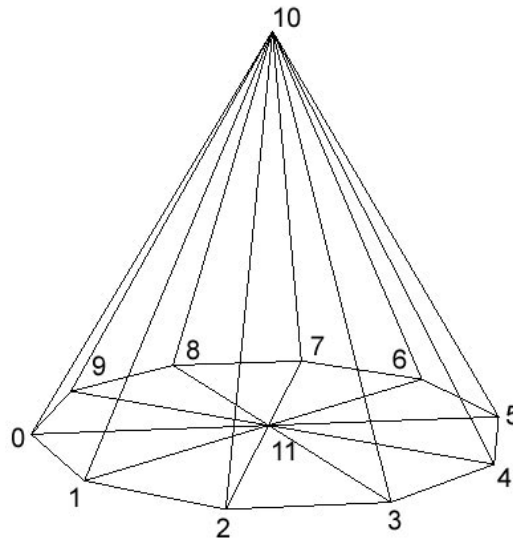
// create mesh
```

```
dM = System.CreateDO("Space 3D Package/Mesh Data");  
dM.AttachVerticesStream(dV);  
dM.AttachTrianglesStream(dF);  
  
params.conValue('outMesh') = dM;  
}
```

While this script may look complex it is actually very straightforward when taken piece by piece. Essentially, it reacts to a request for an updated mesh on the outMesh attribute. Then, the script obtains the input values for longitude, radius, and height and uses those values to construct a mesh representing a cone.

To do this it first determines the number of vertices and faces in the mesh. The number of vertices equals the longitude (or number of divisions around the circumference of the cone) plus two (one for the top of the cone and one for the bottom center). The number of faces is equal to two times the longitude (there are an equal number of faces on the bottom of the cone and along the circumference, between the base and the top).

It then, using a RDVertexStream object, creates the vertices for the bottom, using trigonometric functions to place vertices along the circumference of the cone. Then it creates a vertex at the bottom, center of the cone and another at the center, located at the height. The following image shows the order of vertex creation for a 10-sided cone mesh.



Creating vertices and faces to make up the cone mesh

Now, the script defines the faces for the cone, using a `RDTriangleStream` object. First it defines the bottom of the cone, followed by the upper faces. Faces are specified by identifying the indices of three vertices to be used as vertices for the triangle in counter-clockwise order. So, the first face would use vertices 11, 1, and 0, in that order. The second would use 11, 2, and 1.

Finally, the script creates a `RDMesh` object and attaches the vertices and triangles to it and then outputs that to the attribute `outMesh`. Each time a slider on the outer panel is changed the object is requested to update the value of `outMesh`, causing the script to execute.

You can use this basic structure to create almost any sort of geometry you can think of. For instance, with slightly different formulae you could create a sphere object, or a geosphere.

Advanced implementation notes:

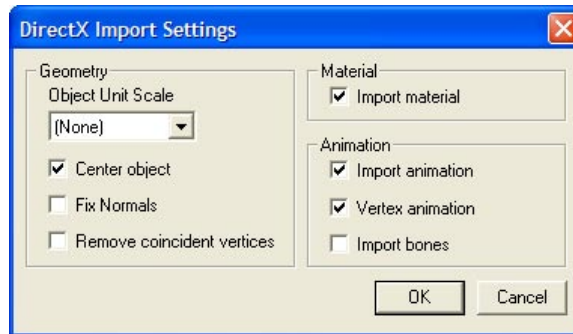
- As for output Mesh attribute that the mesh generator scripts produce, you have 2 possibilities here (both used in existing demos in libraries) – either you define output mesh connector as registered Mesh attribute (you have to check “Registered” checkbox in the attributes edit dialog), or use simple Mesh attribute – in the later case you have to add “Shape” component to your final object to allow tS renderers to recognize the mesh. (refer to the Cone object and Gear object in the Script objects library to compare both approaches)

Chapter 5

File Formats

5.1 DX import

The DirectX (*.x) file format supports most geometry, material, and animation templates. Skinning animation is converted to vertex animation, and at the frames' centers are created bones of the new objects.



Geometry

- **Object Unit Scale:** Converts the object to either the desired unit system, no unit system, or to fit on screen
- **Center object:** Places the scene at the world center (0,0,0). This is for placement of the whole scene, not each object.
- **Fix normals:** Use this option if objects appear to be imported inside out.
- **Remove coincident vertices:** Remove duplicate (coincident) vertices from the model.

Material

- **Import material:** Enable or disable material importing.

Animation

- **Import animation:** Enable or disable animation importing.
- **Vertex animation:** When enabled, vertex animation is imported. (*Import animation* must also be enabled.)
- **Import bones:** Import bones disconnected from the model and skin.

Note: Bones are loaded as separated objects. For vertex animation, the position of every vertex for every frame is computed and added to the mesh. The DirectX importer

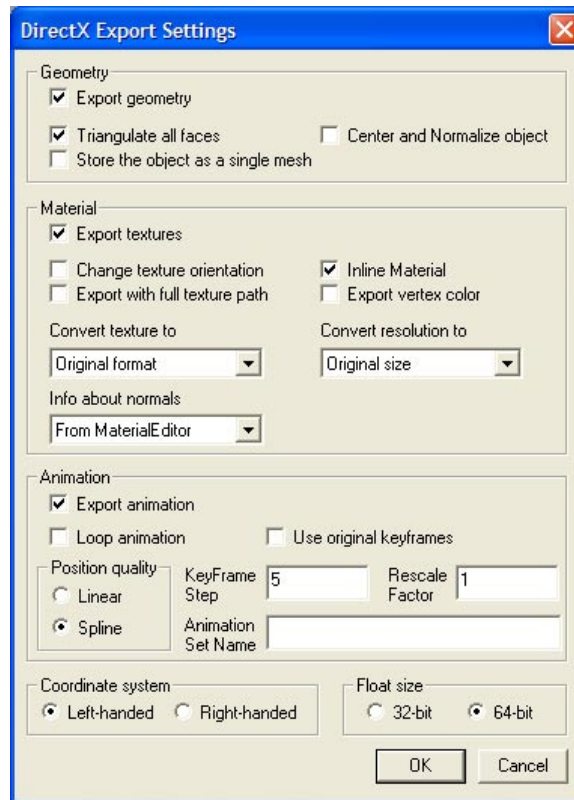
does not create the entire skeleton, but you can do it manually with the imported bones of the loaded X object. This can be done by importing without vertex animation but with bones (“Import bones” check box).

Limitations

- Loading files with matrix animation keys in template Animation-Key is not supported. Only rotation, scale and position are supported.
- Supported files are only text (ascii) or binary. If you have a compressed binary format you can transform it to ascii or binary using the "mview" program, which is a part of DirectX SDK.
- **The following templates are not supported:** CompressedAnimationSet, DeclData, Effect-Param-DWord, VertexDuplicationIndices, Bézier patches templates and FVFData template. These templates are skipped in loading process.

5.2 DX export

There are trueSpace features that cannot be exported into the .x file format because DirectX does not support them. These are limitations of DirectX and not of trueSpace’s exporter. DirectX does not support material animation. However, trueSpace exports the animation of IK objects, converting it into Move and Rotate keyframes of all sub-objects of the IK object in each. Vertex animation is not exported, either



Geometry Options

- **Export geometry:** Check box for exporting mesh data.
- **Triangulate all faces:** Triangulate all faces if you experience problems with the untriangulated geometry, in particular in the DirectX viewer.
- **Store object as a single mesh:** Hierarchical objects can be exported with a hierarchy, or they can be converted to a single mesh object. In the case of a single mesh export, the frame, including the transformation matrix information and animation, is not exported.
- **Center and normalize object:** Object is exported centered and with a normalized size for easier viewing in DirectX viewer.

Material Options

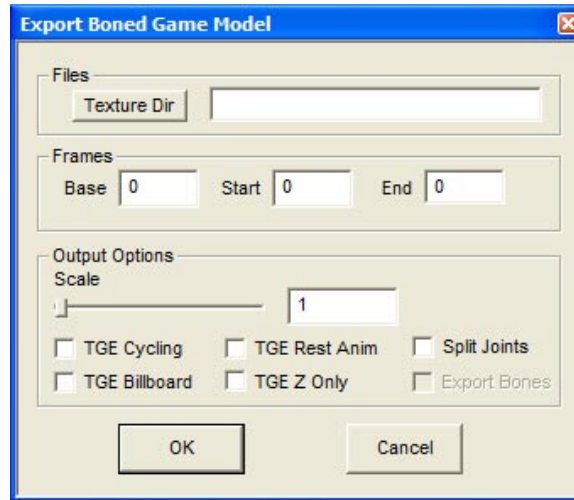
- **Export Material:** Check box for exporting material (texture) data.
- **Change texture orientation:** Old X format supported another orientation than new one DirectX 8.1. You have the option to select to change orientation of textures for the old DirectX format, or orientation by version 8.1 or newer.
- **Export with full texture path:** Export the texture bitmap with full texture path. If disabled, the texture file will be copied into the current object export directory.
- **Inline material:** You can choose define a material inside a mesh or before mesh declaration.

- **Export vertex color:** Add or remove vertex color from X file (template MeshVertexColors). When this template is in the exported file, some DirectX engines can mix this value from standard material (Material template)
- **Info about normals:** Since DirectX does not support autofacet materials, feature edges of the object must be exported explicitly by trueSpace at the expense of a larger file size for the exported object. Enable this option if your geometry has important feature edges that have to be exported precisely.
- **Convert textures to:** Type of exported format. You can choose from PNG, JPG, TGA, DDS or BMP. For example the oldest DirectX format supports only 8bit BMP textures with size a power of 2, so viewers using this format are not able to read another format. There is a necessary conversion of textures to 8bit bitmap that have a size of a power of 2.
- **Convert resolution to:** Type of exported resolution. You can choose from 512x512, 256x256, 128x128, 96x96, 64x64, 32x32 or find closest exponent calculation of number 2.

Animation Options

- **Export Animation:** Export selected object or scene with animation.
- **Animation Loop:** Loop animation sequence. Default value is off
- **Animation Set Name:** Name of Active Animation Channel
- **Rescale Factor:** Value for user-defined keyframe scale factor for animation export with range from 0 to 99999. Default value is 1.
- **Keyframe step value:** Value for user-defined keyframe step animation export with range from 0 to 999.
- **Use original keyframe:** Select if you want to export the original keyframes defined in trueSpace (from the scene editor).
- **Position Quality:** Linear or spline position quality
- **Float size:** Define floating point size for DirectX format. 32-bit or 64-bit.
- **Coordinate system:** Exported object is in Left-handed or Right-Handed coordinate system

5.3 Conitec A6 MDL export



3DGS A6: Generates a single mdl7 format model file using bones as the animation method. Creates one or more textures in 565 rgb mipmapped format in the mdl7 file

Files:

- **Texture Dir:** The directory where the texture file(s) (if any) will be placed, since it is frequently different from where the model file needs to go.

Frames: Defines the animation frames that will be exported.

- **Base:** The zero frame from which all movements are based. (Note: It does not have to be included in the animation.) It is also the frame that is used to assign points to bones, so a non-deformed frame may work better in some cases.
- **Start:** The first frame output.
- **End:** The last frame output. Number of frames = End - Start + 1.

Output options:

- **Scale:** Scales the model up or down to match your other game components. This is a multiplier, not a percentage. 10 is ten times bigger, not ten percent bigger. You can enter decimal places in the edit box.
- **Split Joints:** By default, all points in the joint area on the near end of a bone are assigned to that bone. With this switch on, it will split the joints and assign them to the closest bone. This is a user option as some models look better with this on, but some look better with it off, depending on the model and the movement in the specific animation.
- **Export Bones:** With the Conitec A6 mdl7 format, which supports vertex animation, the animation will be exported as a vertex animation if Export Bones is not selected. For Torque and BlitzBasic output, the Export Bones switch turned off will let you output a non-animated 3D

model. Bones carry a performance penalty in Torque, so turn this option off for all non-animated models. With the Export Bones option you can select either the skeleton (if it has one) or just the mesh object itself.