GenDL Documentation

version 1581 2012-05-24

Table of Contents

v

1 Introduction

The acronym GenDL stands for General-Purpose Declarative Language. By General-Purpose it is meant that the language may be used to create a wide spectrum of end results. It is a superior platform for creating and deploying web-centric Knowledge-based Engineering and Business applications. A particular strength of the platform is the high efficiency in representing and evolving the definitions of complex systems, including three-dimensional geometric models.

Existing and potential examples with geometry include automotive or airplane wiring and hose systems, sheet metal surfaces, weld optimization, baggage delivery carousels, fluid storage tanks and boilers, airplane fuselages, wings, cabin configurations, other components, a web-based e-learning application for model race cars, "green" cars, and bridge structures, with a 3D virtual world gaming component — plus many others.

Examples without geometry include a Trucking company's national delivery scheduling, a multi-national corporate Patent Tracking system, an individual's computer Diary system.

1.1 What is GenDL/GWL

GenDL is a superset of ANSI Common Lisp (with case-sensitive "modern mode" extensions), and consists mainly of automatic code-expanding extensions to Common Lisp, implemented in the form of macros. When you write, for example, 20 lines in GenDL, you might be writing the equivalent of 200 lines of Common Lisp. Of course, since GenDL is a superset of Common Lisp, you still have the full power of the CL language at your fingertips whenever you are working in GenDL.

Since GenDL code expands into CL, everything you write in GenDL will be compiled "down to the metal" to machine code with all the optimizations and safety that the tested-and-true CL compiler provides. This is an important distinction when contrasted to some other so-called KB or KBE systems on the market, which are really nothing more than interpreted scripting languages which quickly meet their limitations when pushed to compute something more demanding than simple parameter-passing.

GenDL is also a true declarative language. When you put together a GenDL application, you write and think mainly in terms of objects and their properties, and how they depend on one another in a direct sense. You do not have to track in your mind explicitly how one object or property will call another object or property, in what order this will happen, etc. Those details are taken care of for you automatically by the language. Because GenDL is object-oriented, you have all the features you would normally expect from an object-oriented language, such as:

- Separation between the definition of an object and an instance of an object
- High levels of data abstraction
- The ability for one object to inherit from others
- The ability to use an object without concern for its under-the-hood implementation

GenDL supports the message-passing paradigm of object orientation, with some extensions. Since full-blown ANSI CLOS (Common Lisp Object System) is always available as well, the Generic Function paradigm is supported as well. Do not be concerned at this point if you are not fully aware of the differences between these two paradigms.

GenDL ships also with a standard web-development-component, the GWL (Generative Web Language), which consists essentially of a set of mixins and a few functions which provide a convenient mechanism to present KB objects defined in GenDL through a standard HTTP/HTML web user interface. GWL is designed to operate in conjunction with AllegroServe and a compatible HTML generating facility such as cl-who or htmlgen.

1.2 Packages and Licensing

Genworks offers GenDL in a variety of packages, from Personal/Trial Editions and Student Editions through to high-end Commercial packages. The most popular packages are presented in the next sections.

1.2.1 Enterprise Edition Development Environment

The Enterprise Edition is for use in developing, building, and deploying runtime applications in a networked corporate Enterprise environment. It. includes:

- *All GenDL/GWL Professional Edition components*
- *Ability to generate standalone runtime single-user and (web) server-based GenDL/GWL applications, as well as shared-library (.DLL or .so) applications.*
- *Distributed GenDL (dGenDL), allowing child objects of GenDL instance trees to exist on separate processes on local or remote server(s).*
- *Access to Relational Databases*
- *GenDL integrated object-oriented relational database support, including table definition compiler and caching and dependency-tracking for database table and row objects*
- *GWL relational database user interface, providing default web-based access to insert, update, and delete database table rows*
- *Secure Socket Layer for building secure web server and client applications*
- *Enterprise customers are automatically licensed to generate and use Runtime Applications for noncommercial activities (e.g. deployment testing, demonstrations, etc.)*

1.2.2 Professional Edition Development Environment

The Professional Edition is for use in the early phases of application development. It does not include the capability to generate runtime applications. It includes:

- *GenDL Compiler and Development Environment (integrated with CL/CLOS compiler)*
- *Library of GenDL Objects, Functions, and Macros*
- *GWL Web Application Framework, integrated with a fully-featured web server*
- *GWL-based GenDL Object Tree Browser and Inspector*
- *Fully cross-referenced web-based GenDL/GWL reference documentation, tutorials, and a set of functioning sample applications*
- *Base CL/CLOS compiler and development system with profiler and debugger*

1.2.3 Evaluation and Student Edition Development Environment

The Evaluation and Student Editions are for evaluation or education purposes. They do not include the capability to generate runtime applications or direct support from Genworks *(the support is provided via Genworks Google Group)*.

For additional Genworks products, pricing and support information please visit the Website `http://genworks.com`

2 Installation and tips

This Chapter is applicable if you are running a GenDL version newer than May 2012. Detailed installation instructions are available on www.genworks.com/newsite.

2.1 Basic Installation

This section describes how to install GenDL. To download the software and retrieve a license key, follow the next steps:

1. Visit the download section of the next URL: http://genworks.com/newsite
2. Enter your email address. If you don't have an email address on file with Genworks, send email to: licensing@genworks.com. Read and accept the applicable license agreement and click the checkbox.
3. Click the link to download the .zip file, and start the download to a known location on your computer.
4. Click the "Retrieve License Key" link, to have your license key file(s) emailed to you.

GenDL is currently distributed for all the platforms as a self-contained "zip" file which does not require official administrator installation. To install the downloaded software follow the instructions received in the license email.

NOTE: For the Windows platform, you will need to download the gpl directory.

2.2 Becoming an effective GenDL developer

Becoming effective with GenDL requires three (3) basic skills:

1. Editing text and managing files with Gnu Emacs (yes, other editors will work, but Emacs provides special support and only Emacs is supported by Genworks);
2. Very basic Common Lisp programming;
3. Writing Object Definitions in GenDL itself, using the define-object macro.

Beyond this, if you want to create nice Web applications with GenDL, it helps to know something about HTML and server-based computing. But these concepts can be picked up over time and are not necessary to get started or to make simple applications or 3D geometric objects.

3 Getting Started with Emacs

Assuming you followed the steps described in Section 2.1 [Basic Installation], page 5, at this moment you now have Emacs running on your PC. If not, start Emacs with the GenDL environment with the "run-gdl.bat" script (Windows) or the "run-gdl" shell-script (Linux). Emacs should come up with instructions for:

1. How to use Emacs 'basic tutorial'

2. How to start the GenDL process

3. Instructions for using special keychord shortcuts in GenDL

3.1 How to use Emacs 'basic tutorial'

GNU Emacs is an extensible, customizable text editor and more. At its core is an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing, for more details see http://www.gnu.org/software/emacs/. Note that Emacs Lisp is a *different* Lisp process (memory space) and dialect from the Common Lisp which hosts GenDL. However the existence of Emacs Lisp makes Emacs a very "Lisp-aware" environment.

The best way to become proficient in Emacs quickly is to invest approximately 20-40 minutes and perform the built-in interactive tutorial. This can be started (also in many languages alternative to English) from the "Help" menu at the top of the Emacs screen.

To summarize the tutorial here: working in Emacs involves the use of a CONTROL key (sometimes labeled CTRL or CTL) or the META key (sometimes labeled EDIT or ALT). Rather than write that in full each time, in Emacs the following abbreviations are used:

- *C-<chr> means hold the CONTROL key while typing the character <chr>*

  For example; C-f would be: hold the CONTROL key and type f.

- *M-<chr> means hold the META or EDIT or ALT key down while typing <chr>*

Emacs is highly customizable and extensible, but it is not necessary to master its full capabilities in order to become highly proficient in GenDL development. If you can commit the following commands to memory in your first few days, it will greatly speed up your future development:

1. Viewing screen commands:
   - *C-g Cancel any pending command – very useful especially at the beginning, always remember to use this if you are unsure what is happening in Emacs*
   - *C-v Move forward one screenful*
   - *M-v Move backward one screenful*
   - *C-x 2 Splits the screen into two windows*
   - *M-x make-frame Makes a new frame to appear on your screen*

2. Cursor-moving commands:
   - *C-f Move forward a character*
   - *C-b Move backward a character*
   - *M-f Move forward a word*

- *M-b Move backward a word*
- *C-n Move to next line*
- *C-p Move to previous line*
- *C-a Move to beginning of line*
- *C-e Move to end of line*
- *M-a Move back to beginning of sentence*
- *M-e Move forward to end of sentence*

3. Delete operation commands:
    - *<Delback> Delete the character just before the cursor*
    - *C-d Delete the next character after the cursor*
    - *M-<Delback> Kill the word immediately before the cursor*
    - *M-d Kill the next word after the cursor*
    - *C-k Kill from the cursor position to end of line*
    - *M-k Kill to the end of the current sentence*

4. Other commands:
    - *C-x C-f Find file*
    - *C-x C-s Save file*
    - *C-x s Save some buffers*
    - *C-x C-b List buffers*
    - *C-x b Switch buffer*
    - *C-x C-c Quit Emacs*
    - *C-x 1 Delete all but one window*
    - *C-x u Undo*

For more commands see the Emacs help menu or the GenDL readme.txt which is automatically displayed at startup.

3.2 How to start and exit the GenDL process

There are two ways to start the Gendl environment. The first is Slime, the Superior Lisp Interaction Mode for Emacs. Slime enjoys wide community support and works across all major OS platforms and CL implementations. The second is Franz ELI (Emacs Lisp Interface). ELI works with Allegro CL-based Gendl environments (currently supported by Genworks on Linux and Windows) and receives commercial support from Franz Inc.

Starting Slime can be achieved by typing *M-x slime* (that is, hold down the Meta (or Alt) key, and type x, then type 'slime' and hit Enter.)

Quitting Slime can be achieved as follow:

1. type M-x slime-quit-lisp (and hit Return) from anywhere in Emacs, to kill the GenDL process.
2. (optional) type C-x C-c (that's Control-x, Control-c) to kill the Emacs process and exit the window.

3. If you kept Emacs up and running, it is possible to restart the Gendl session with either ELI or Slime, by following the instructions under "Starting."

Starting ELI can be achieved by typing M-x gdl (that is, hold down the Meta (or Alt) key, and type x, then type 'gdl' and hit Enter.)

Quitting ELI can be achieved as follow:

1. type :exit (and hit Return) at the command prompt in the *gdl toplevel* buffer to kill the GenDL process.

2. type C-x C-c (that's Control-x, Control-c) to kill the Emacs process and exit the window.

3. If you kept Emacs up and running, it is possible to restart the Gendl session with either ELI or Slime, by following the instructions under "Starting."

3.3 Compiling and loading a buffer in GenDL

A buffer can be compiled and loaded while its content is edited. Depending on how the GenDL process was started (using Slime or ELI) there are two options to compile and load a buffer for each starting modes: one using the Emacs graphical interface and the other using a set of shortcut keys; see below description:

Compile and load a buffer using Slime

Shortcut key option:

- *C-c C-c compile/load current top-level form (within which the cursor is)*

- *C-c C-k compile/load file*

- *C-x C-e evaluates previous form (cursor should be after the last paren)*

Note: If the buffer is not saved, the compiler will offer the option to save the file (yes or no).

The **graphical option** is available from the Slime menu at the top of the Emacs screen; see the screenshot illustrated below:
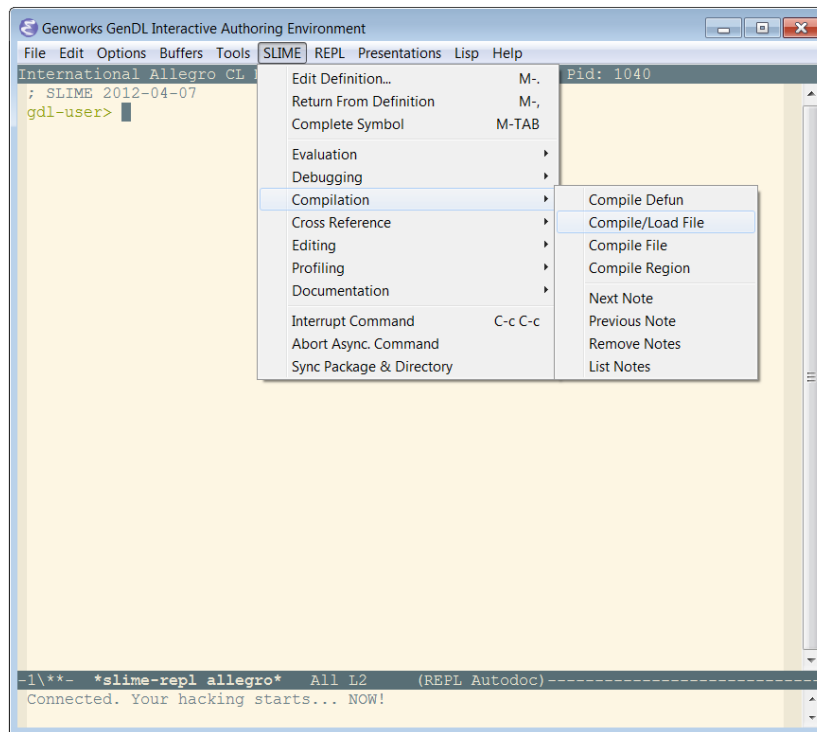
Compile and load a buffer using ELI

The **graphical option** is available by right clicking on the buffer screen; see the screenshot illustrated below:
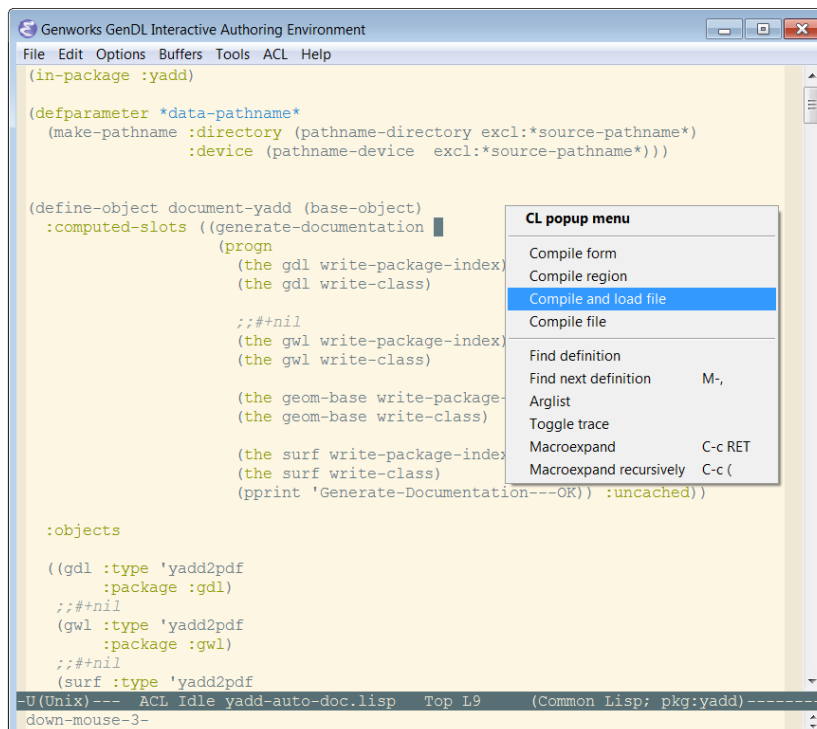
Note: The buffer must be saved and have the extension .lisp

Shortcut key option:

- C-x C-s – save current top-level form.
- C-c C-b – compile/load current toplevel form (within which the cursor is)

4 Getting Started with GenDL Itself

The Genworks GenDL language is a superset of ANSI Common Lisp (CL). If you are new to the CL language, we recommend that you supplement this chapter with other resources. Resources for Common Lisp abound in bookstores and on the Web[1]. One example of a concise resource to get started is Basic Lisp Techniques[2], which also includes a small Genworks GenDL example.

In the meantime, this chapter will provide a condensed overview of the language. Please note, however, that this book is intended as a summary, and will not delve into some of the more subtle and powerful techniques possible with Common Lisp.

4.1 Overview of CL and its Syntax

The first thing you should observe about GenDL (and most languages in the Lisp family) is that it uses a generalized *prefix* notation.

One of the most frequent actions in a CL program, or at the toplevel *read-eval-print* loop, is to call a *function*. This is most often done by writing an *expression* which names the function, followed by its arguments. Here is an example:

(+ 2 2)

This expression consists of the function named by the symbol '**+**', followed by the arguments 2 and another 2. As you may have guessed, when this expression is evaluated it will return the value 4.

Try it: Try typing this expression at your command prompt, and see the return-value being printed on the console.

What is actually happening here? When CL is asked to *evaluate* an *expression* (as in the toplevel *read-eval-print* loop), it evaluates the expression according to the following rules:

1. If the expression is a number (i.e. looks like a number), it simply evaluates to itself (a number):

   gdl-user(1): 99
   99

2. If the expression looks like a *string* (i.e. is surrounded by double-quotes), it also simply evaluates to itself:

   gdl-user(2): "Our golden rule is simplicity"
   "Our golden rule is simplicity"

3. If the expression looks like a literal *symbol*, it will simply evaluate to that symbol

   gdl-user(3): 'my-symbol

---

[1] http://www.cliki.net, http://www.common-lisp.net
[2] http://www.genworks.com/downloads/blt-2011.pdf

my-symbol

4. If the expression looks like a list (i.e. is surrounded by parentheses), CL assumes that the *first* element in this list is a *symbol* which names a *function* or a *macro*, and the *rest* of the elements in the list represent the *arguments* to the function or macro. (We will discuss functions first, macros later). A function can take zero or more arguments, and can return zero or more return-values. Often a function only returns one return-value:

   gdl-user(4): (expt 2 5)

   32

Try it: Try typing the following functional expressions at your command prompt, and convince yourself that the printed return-values make sense:

(+ 2 5)

(+ 2)

2

(+ (+ 2 2) (+ 3 3))

(+ (+ 2 2))

(sys:user-name)

4.2 Fundamental CL Data Types

Common Lisp natively supports many data types common to other languages, such as numbers, strings, and arrays. Native to CL is also a set of types which you may not have come across in other languages, such as lists, symbols, and hash tables. In this overview we will focus on numbers, strings, symbols and lists.

Regarding data types, CL follows a paradigm called dynamic typing. Basically this means that values have type, but variables do not necessarily have type, and typically variables are not "pre-declared" to be of a particular type.

4.2.1 Numbers

Numbers in CL form a hierarchy of types, which includes Integers, Ratios, Floating Point, and Complex numbers. For many purposes, you only need to think of a value as a "number" without getting any more specific than that. Most arithmetic operations, such as +, -, *, /, etc, will automaticaly do any necessary type coercion on their arguments and will return a number of the appropriate type.

CL supports a full range of floating-point decimal numbers, as well as true Ratios, which means that 1/3 is a true one-third, not 0.333333333 rounded off at some arbitrary precision.

As we have seen, numbers in CL are a native data type which simply evaluate to themselves when entered at the toplevel or included in an expression.

4.2.2 Strings

Strings are actually a specialized kind of array, namely a one-dimensional array (vector) made up of characters. These characters can be letters, numbers, or punctuation, and in some cases can include characters from international character sets (e.g. Unicode) such as Chinese Hanzi or Japanese Kanji. The string delimiter in CL is the double-quote character.

As we have seen, strings in CL are a native data type which simply evaluate to themselves when included in an expression.

4.2.3 Symbols

Symbols are such an important data structure in CL, that people sometimes refer to CL as a "Symbolic Computing Language." Symbols are a type of CL object which provides your program with a built-in mechanism to store and retrieve values and functions, as well as being useful in their own right. A symbol is most often known by its name (actually a string), but in fact there is much more to a symbol than its name. In addition to the name, symbols also contain a *function* slot, a *value* slot, and an open-ended *property-list* slot in which you can store an arbitrary number of named properties.

For a named function such as + the function-slot of the symbol + contains the actual function object. The value-slot of a symbol can contain any value, allowing the symbol to act as a global variable, or *parameter*. And the property-list, also known as the *plist* slot, can contain an arbitrary amount of information.

This separation of the symbol data structure into function, value, and plist slots is one obvious distinction between Common Lisp and most other Lisp dialects. Most other dialects allow only one (1) "thing" to be stored in the symbol data structure, other than its name (e.g. either a function or a value, but not both at the same time). Because Common Lisp does not impose this restriction, it is not necessary to contrive names, for example for your variables, to avoid conflicting with existing "reserved words" in the system. For example, **list** is the name of a built-in function in CL. But you may freely use **list** as a variable name as well. There is no need to contrive arbitrary abbreviations such as "lst."

How symbols are evaluated depends on where they occur in an expression. As we have seen, if a symbol appears first in a list expression, as with the + in **(+ 2 2)**, the symbol is evaluated for its function slot. If the first element of an expression indeed has a function in its function slot, then any subsequent symbol in the expression is taken as a variable, and it is evaluated for its global or local value, depending on its scope (more on variables and scope later).

As noted in Section 3.1.3, if you want a literal symbol itself, one way to achieve this is to "quote" the symbol name:

'a

Another way is for the symbol to appear within a quoted list expression:

'(a b c)

'(a (b c) d)

Note that the quote (') applies across everything in the list expression, including any sub- expressions.

4.2.4 Lists

Lisp takes its name from its strong support for the list data structure. The list concept is important to CL for more than this reason alone — most notably, lists are important because *all CL programs are themselves lists.*

Having the list as a native data structure, as well as the form of all programs, means that it is straightforward for CL programs to compute and generate other CL programs. Likewise, CL programs can read and manipulate other CL programs in a natural manner. This cannot be said of most other languages, and is one of the primary distinguishing characteristics of Lisp as a language.

Textually, a list is defined as zero or more elements surrounded by parentheses. The elements can be objects of any valid CL data types, such as numbers, strings, symbols, lists, or other kinds of objects. As we have seen, you must quote a literal list to evaluate it or CL will assume you are calling a function. Now look at the following list:

(defun hello () (write-string "Hello, World!"))

This list also happens to be a valid CL program (function definition, in this case). Don't worry about analyzing the function right now, but do take a few moments to convince yourself that it meets the requirements for a list.

What are the types of the elements in this list?

In addition to using the quote (') to produce a literal list, another way to produce a list is to call the function **list**. The function **list** takes any number of arguments, and returns a list made up from the result of evaluating each argument. As with all functions, the arguments to the **list** function get evaluated, from left to right, before being passed into the function. For example:

(list a b (+ 2 2))

will return the list

(a b 4)

The two quoted symbols evaluate to symbols, and the function call **(+ 2 2)** evaluates to the number 4.

4.3 Functions

Functions form the basic building blocks of CL. Here we will give a brief overview on how to define a function; later we will go in more detail on what a function actually is.

A common way to define named functions in CL is with the macro **defun**, which stands for *DEFinition* of a *FUNction*. Defun takes as arguments a symbol, an argument list, and a body:

gdl-user(5): (defun my-first-lisp-function () (list 'hello 'world))

Because defun is a macro, rather than a function, it does not automatically evaluate all its arguments as expressions — specifically, the symbol which names the function does not have to be quoted, nor does the argument list. These are taken as a literal symbol and a literal list, respectively.

Once the function has been defined with defun, you can call it just as you would call any other function, by wrapping it in parentheses together with its arguments:

```
gdl-user(6): (my-first-lisp-function)

(hello world)

gdl-user(7): (defun square(x) (* x x))

square

gdl-user(8): (square 4)

16
```

Note: Declaring the types of the arguments to a function is not required.

FLAG – Add sections on required, optional, and keyword arguments to functions!!

4.4 Macros

Macros in CL provide a very powerful and flexible method of extending CL syntax. A CL macro is like a function that takes Lisp forms or objects as input, and typically generates code to be then compiled and executed. This happens before runtime, in a phase called macroexpansion time. Macros can perform arbitrary computation during expansion, using the full CL language.

One use of macros is transforming input representing arbitrary source code into a version specified in terms of known definitions. In other words, macros can add new syntax to the original language (this is known as syntactic abstraction).

This enables easily embedding domain-specific languages, since specialized syntax can be added before compile-time.

The chief benefit of macros is that they add power by letting the programmer express intent clearly and with less code. Particularly, one can add new features to the language that appear as if they were built-in. In addition, when used to pre-emptively compute data or initialize state, macros may aid in performance optimisation. *(source)*.

Although we introduce macros to familiarize you with the concept, you typically will have no need to author new macros when using GenDL. In fact, the GenDL *kernel* (i.e. core software component) already provides a general-purpose macro which covers the most typical requirements for a generative application. This macro is called *define-object*, introduced in the next section.

4.5 GenDL Syntax

4.5.1 Define-Object

The *define-object* macro acts as the basic operator for defining objects in GenDL. An *object definition* acts a generic "recipe" or "blueprint," from which specific *object instances* (also simply called *objects*) may be generated.

The define-object macro takes three basic arguments:

- *a name*, which is a symbol;
- *a mixin-list*, which is a list of symbols naming other objects from which the current object will *inherit* characteristics;
- *a specification-plist*, which is spliced in (i.e. does not have its own surrounding parentheses) after the mixin-list, and describes the object model by specifying properties of the object (messages, contained objects, etc.) The specification-plist typically makes up the bulk of the object definition.

Here are descriptions of the most common keywords which can be used in the specification-plist:

input-slots specify information to be passed into the object instance when it is created.

computed-slots specify "formulas" to compute a value on-demand.

objects specify the types and inputs for other objects which can be instantiated on-demand and contained within this object.

functions specify (uncached) Lisp functions which take the current object as the first argument and can also take additional arguments, just like a normal CL function defined with **defun**.

```
(define-object hello ()

  :input-slots
  (first-name last-name)

  :computed-slots
  ((greeting (format nil "Hello, ~a ~a!!"
                     (the first-name)
                     (the last-name)))))
```

Figure 1: Example of Simple Object Definition

Figure 2.1 shows a simple example, which contains two input-slots, **first-name** and **last-name**, and a single computed-slot, **greeting**. As you can see, a GenDL Object is analogous in some ways to a **defun**, where the input-slots are like arguments to the function, and the computed-slots are like return values. But seen another way, each computed-slot in a GenDL object is like a function in its own right.

The referencing macro **the** shadows CLs **the** (which is a seldom-used type declaration operator). **the** in GenDL is used to reference the value of other messages within the same object or within contained objects. In the above example, we are using **the** to refer to the values of the messages (input-slots) named **first-name** and **last-name**. Note that messages used with **the** are given as symbols. These symbols are unaffected by the current Lisp

package, so they can be specified either as plain unquoted symbols or as *keyword* symbols (i.e. symbols preceded by a colon), and the **the** macro will process them identically.

In the previous example, the mixin-list is an empty list. As a consequence, the object hello is a "stand-alone" object; it does not inherit characteristics from any other definition within GenDL[3]. Figure 2 shows an example containing the definition of an object named my-circle-sample which inherit the characteristics of the circle object defined in GenDL.

```
(define-object my-circle-sample (circle)
  :computed-slots
  ((radius 10)))
```

Figure 2: Example of Simple Circle Object Definition

4.5.2 Making Instances and Sending Messages

Once we have defined an object such as the example above, we can use the constructor function **make-object** in order to create an *instance* of it. Below we create an instance of hello with specified values for first-name and last-name (the required input-slots), and assign this instance as the value of the symbol my-instance:

```
gdl-user(9): (setq my-instance
                (make-object 'hello
                             :first-name "John"
                             :last-name "Doe"))

#<HELLO  #x218f39c2>
```

As you can see, *keyword* symbols are used to tag the input values, and the return value is an instance of class hello (recall that a keyword symbol is a symbol whose name is preceded by a colon (":")). Now that we have an instance, we can use the macro **the-object** to send messages to this instance:

```
gdl-user(9): (the-object my-instance greeting)

"Hello, John Doe!!"
```

The-object is similar to the, but as its first argument it takes an expression which evaluates to an object instance (typically, as in the example above, this will just be a symbol which is acting as a variable). **the**, by contrast, assumes that it is being used within a particular object definition (i.e. within the code of a particular **define-object**), and the message is assumed to be available in that definition.

---

[3] Actually, this is a simplification. GenDL contains a built-in object definition called **vanilla-mixin**, which is inherited automatically by all definitions created with **define-object**.

```
(define-object city ()
  :computed-slots
  ((total-water-usage (+ (the hotel water-usage)
                         (the bank water-usage))))
  :objects
  ((hotel :type 'hotel
          :size :large)
   (bank :type 'bank
          :size :medium)))
```

Figure 3: Object Containing Child Objects

Similarly to **the**, **the-object** evaluates all (but the first) of its arguments as package-immune symbols, so although keyword (preceded by colon) symbols may be used, plain unquoted symbols will work just as well.

For convenience, you can also set the special variable **self** manually at the CL Command Prompt, and use **the** instead of **the-object** for referencing:

```
gdl-user(10): (setq self
                 (make-object 'hello :first-name "John"
                                     :last-name "Doe"))

#<HELLO  #x218f406a>

gdl-user(11): (the greeting)

"Hello, John Doe!!"
```

It may help to conceptualize that

```
(the ...)
```

is equivalent to

```
(the-object self ...)
```

4.5.3 Objects

The **:objects** keyword specifies a list of "contained" instances, where each instance is considered to be a "child" object of the current object. Each child object is of a specified type, which itself must be defined with define-object before the child object can be instantiated. Input values to each instance are specified as a plist of keywords and value expressions, spliced in (i.e. not inside their own set of parentheses) after the objects name and type specification. These inputs must match the inputs protocol (i.e. the **input-slots**) of the object being instantiated.

Figure 2.2 shows an example of an object which contains some child objects. In this example, **hotel** and **bank** are presumed to be already (or soon to be) defined as objects themselves, which each answer the **water-usage** message. The *reference chains:*

```
(the hotel water-usage)
```

and

```
(the bank water-usage)
```

provide the mechanism to access messages within the child object instances.

These child objects become instantiated *on demand*, meaning that the first time they or any of their messages are referenced, the actual instance will be created and cached appropriately for future reference.

4.5.4 Sequences of Objects and Input-slots with a Default Expression

```
(defparameter *presidents-data*
    '((:name
        "Carter"
        :term 1976)
      (:name "Reagan"
        :term 1980)
      (:name "Bush"
        :term 1988)
      (:name "Clinton"
        :term 1992)))

(define-object presidents-container ()
  :input-slots
  ((data *presidents-data*))

  :objects
  ((presidents :type president
               :sequence (:size (length (the data)))
               :name (getf (nth (the-child index) (the data)) :name)
               :term (getf (nth (the-child index) (the data)) :term))))
```

Figure 4: Sample Data and Object Definition to Contain U.S. Presidents

Objects may be *sequenced* , to specify, in effect, an array or list of object instances. The most common type of sequence is called a *standard* sequence. See Figure 4 for an example of an object which contains a sequenced set of instances representing U.S. presidents. Each member of the sequenced set is fed inputs from a list of plists, which simulates a relational database table (essentially a "list of rows").

Note the following from this example:

- In order to sequence an object, the input keyword **:sequence** is added, with a list consisting of the keyword **:size** followed by an expression which must evaluate to a number.

- In the input-slots, **data** is specified together with a default expression. Used this way, input-slots function as a hybrid of computed-slots and input-slots, allowing a *default expression* as with computed-slots, but allowing a value to be passed in on instantiation or from the parent, as with an input-slot which has no default expression. Note that a passed-in value will always override the default expression.

This GenDL syntax overview has been kept purposely brief, covering the fundamentals of the language in a dense manner. On one hand, it is not meant to be a comprehensive language reference; on the other hand, do not be concerned if you are still unsure about some of the terminology. The upcoming chapter will revisit and further expand many of the topics covered here, and at some point a coherent picture should begin to emerge.

4.6 Development of Projects using Files and Packages

While in the previous chapter most of the coding was done at the prompt level, in this chapter the coding is done by using a project management approach, in separate files. Each time you start a new GenDL project, it is recommended to structure your applications in a modular fashion. This allows you to incrementally compile and load your applications. To illustrate the best practice in creating a new GenDL project, we will describe in detail a small application. This application is created and loaded following the next three steps:

Note: To avoid confusion, at the beginning of each sample code the file name will be mentioned with the following notation: **;;–file-name.lisp–** *and in some cases, comments will be added.*

Step 1. Create a new folder in your HOME folder, named **shock-absorber**. Within this folder create a new folder named **source**.

Step 2. In the **source** directory create the next three lisp files (the file names used below follow the normal convention, but your file names can potentially be different):

The first file will be named **package.lisp** and will contain your package definition as mentioned below:

```
;;--package.lisp--
(in-package :gdl-user)
(define-package :shock-absorber ;;the name of your new package as key word
    (:nicknames :shock :picasso ) ;;the nicknames of your new package
  (:use ) ;;the packages used by your new package
  (:export #:assembly
   ;; Exported symbols
   ))
```

It is also possible to skip the package specification **(in-package :gdl-user)** as presented in this example code, if you refer to the package definition macro as **gdl:define-package**.

Note that you can work directly in the **:gdl-user** package, which is an empty pre-defined package for your use, if you do not wish to make a new package just for scratch work.

For real projects, however, it is recommended that you make and work in your own GenDL package.

In GenDL, the macro **define-package** is used to set up a new working package.

Packages defined with **gdl:define-package** will implicitly **:use** the GenDL package and the Common-Lisp package, so you will have access to all exported symbols in these packages without prefixing them with their package name. You may extend this behavior by calling **gdl:define-package** and adding additional packages to use with **(:use ...)**. For example, if you want to work in a package with access to GenDL exported symbols, Common Lisp exported symbols, and symbols which are defined in other packages like **:my-utilities**, you could set it up as follows:

```
;;--package.lisp--
(gdl:define-package :shock-absorber
    (:nicknames :shock  :picasso )
  (:use :my-utilities)
  (:export #:assembly
   ;; Exported symbols
   ))
```

Note: The "Exported symbols" option will be addressed in the next chapter on a real case. However, the fundamental concept is that symbols are contained in packages, and can be marked as "internal" or "external," which is roughly analogous to "private" and "public" in languages like C++ and Java — that is, internal symbols should only be accessed from inside the same package where they are defined. In the above example, the symbol **assembly** *is exported from the* **:shock-absorber** *package. Note that symbols named in the export list should be prepended with "#:" — this is for "package hygiene" reasons which are not important to understand at this point.*

The second file will be named **assembly.lisp** and will contain the definition for the toplevel "assembly" (i.e. object) of your application. As you have already seen, this file will contain a GenDL object definition so it will have the following structure:

As stated in previous chapters, **define-object** is the basic macro for defining objects (i.e. creating classes) in GenDL. A GenDL object definition in which:

- **class-name** is any non-keyword symbol [i.e. a symbol not preceded by a colon (":")]. A GenDL Class will be generated for this symbol, so that any name you use will override a previous class if one is already defined with the same name.
- **mixin-list** is a list of other class-names (i.e. object definition names) from which this object will inherit. Note that the standard mixin **gdl:vanilla-mixin** gets mixed in automatically with any GenDL object and carries some of the basic GenDL functionalities (messages).
- **specification-plist** is a plist made up of pairs made from special keywords and expression lists.

For the moment we will exemplify the use of the specification-plist by defining a cylinder child object named **my-cylinder** contained by the **new-object** as mentioned below:

```
;;-- assembly.lisp--
(in-package :shock-absorber)

(define-object assembly (base-object)

  :input-slots ()
  :trickle-down-slots ()
  :computed-slots ()

  :objects ((my-cylinder :type 'cylinder
                         :length 10
                         :radius 3))

  :hidden-objects ()
  :functions ()
  :methods ())
```

The **:objects** and **:hidden-objects** are used to specify a list of instance specifications, where each instance is considered to be a "child" object of the current object, **:hidden-objects** serves the same purpose and has the same syntax, but hidden objects are considered **hidden-children** rather than **children** (so they are not returned by a call to (the children), for example).The inputs to each object are specified as a plist of inputs and value expressions, spliced in after the object's name and type specification; in the presented case **:length** and **:radius** for the my-cylinder.

The third file will be named **file-ordering.isc** and is used to enforce a certain ordering on the files when the directory is compiled and loaded. Here is the content of an example for the above application:

```
;;--file-ordering.isc--
("package" "assembly")
```

Step 3. Compiling and loading the application can be achieved by invoking the **cl-lite** function. This is the usual function used to compile and load a directory tree. The function can be invoked at the prompt as follows, assuming you have in your HOME folder a folder "gdl-projects" containing "my-project," which contains the "source/" folder as described above:

```
gdl-user(11): (cl-lite "~/gdl-projects/shock-absorber/")
```

The **cl-lite** function will traverse the pathname in an alphabetical depth-first order (if a **file-ordering.isc** is not defined), compiling and loading any lisp files found in **source/subdirectories**. A lisp source file will only be compiled if it is newer than the corresponding compiled fasl binary file, or if the corresponding compiled fasl binary file does not exist. A **bin/source/** will be created, as a sibling to each **source/ subdirectory**, to contain the compiled fasl files. If the **:create-fasl?** keyword argument is specified as non-nil, a concatenated fasl file, named after the last directory component of pathname, will be created in the temporary directory which defaults to a folder called "tmp/" in your HOME folder. In practice, most of the time you will invoke the **cl-lite** function in a more simple fashion:

```
gdl-user(12): (cl-lite "d:/new-folder/")

!!!in case your pathname in windows is d:\new-folder\!!!
```

Once the application is compiled and loaded, it is possible to interact with the assembly at prompt. To achieve that, change the prompt package to **shock-absorber** by typing:

```
gdl-user(13): (in-package :shock-absorber)

!!!and set self to the new-object by typing:

shock (14): (setq self (make-object 'new-object))

or the shorthand version of the above:

shock (14): (make-self 'new-object)

#<assembly  #x223c71a2>
```

At this point the global variable **self** is set to the **assembly** instance, and you can ask for the object **my-cylinder** or for the **length** or **radius** of the object **my-cylinder** as follows:

```
shock(15):  (the my-cylinder)

#<cylinder  #x223cbefa>

shock(16):  (the my-cylinder length)

10

shock(16):  (the my-cylinder radius)

3
```

Basically, the presented approach is the root of a typical GenDL application. In the next chapter, we will add more contour to this application using the complete **specification-plist** *(:input-slots :trickle-down-slots :computed-slots etc.)* in conjunction with the introduction of a set of other GenDL features. The intent is to present the necessary steps in modeling a simple shock absorber, as the one presented in figure 5.
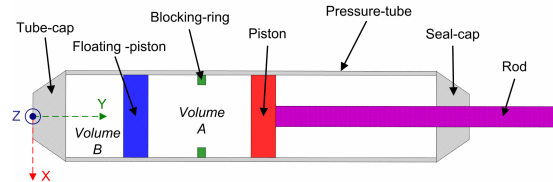
Figure 5: Shock absorber

The example code for this geometrical model is further presented:

```
;;-- assembly.lisp--
;; this is a new version of the file assembly.lisp
(in-package :shock-absorber)
(define-object  assembly (base-object)
  :input-slots ()
  :trickle-down-slots ()
  :computed-slots ()
  :objects
  ((pressure-tube :type 'cone
                  :center (make-point 0 70 0)
                  :length 120
                  :radius-1 13
                  :inner-radius-1 12
                  :radius-2 13
                  :inner-radius-2 12)

   (tube-cap :type 'cone
             :center (make-point 0 5 0)
             :length 10
             :radius-1 5
             :inner-radius-1 0
             :radius-2 13
             :inner-radius-2 0)

 (seal-cap :type 'cone
             :center (make-point 0 135 0)
             :length 10
             :radius-1 13
             :inner-radius-1 2.5
             :radius-2 5
             :inner-radius-2 2.5)
```

```
  (floating-piston :type 'cylinder
                   :center (make-point 0 35 0)
                   :radius 12
                   :length 10)

  (blocking-ring :type 'cone
                 :center (make-point 0 42.5 0)
                 :length 5
                 :radius-1 12
                 :inner-radius-1 10
                 :radius-2 12
                 :inner-radius-2 10)

  (piston :type 'cylinder
          :center (make-point 0 125 0)
          :radius 12
          :length 10)

  (rod :type 'cylinder
       :center (make-point 0 175 0)
       :radius 2.5
       :length 90)
  )

:hidden-objects ()

:functions ()

:methods ())
```

As presented above, this geometric model is relatively simple; it is built by using just a cone and cylinder from predefined GenDL primitive classes. However, even at this level of simplicity, it is not easy for the user to foresee how each defined object will become instantiated to form the shock absorber assembly. In order to allow for incremental "discovery" during the development cycle, GenDL features a dedicated inspection utility called **"Tasty"**, designed for graphical development and testing purposes.

4.7 The Tasty Interface

Tasty[4] is a web based testing and tracking utility. Note that Tasty is designed for developers of GenDL applications — it is not intended as an end-user application interface (see the [FLAG – gdlAjax section] for the recommended steps to create end-user interfaces).

Tasty allows you to visualize and inspect any object defined in GenDL, which mixes at least **base-object** into its root-level part[5] To access Tasty, point your web browser to the URL:

---

[4] Tasty is a tortured acronym of acronyms - it stands for TAtu (ta2) with STYle (sheets), or Testing And tracking utility with STYle.

[5] base-object is the core mixin for all geometric objects and gives them a coordinate system, length, width, and height. This restriction in tasty will be removed in a future GenDL release so you will be able to instantiate non-geometric root-level objects in tasty as well, for example to inspect objects which generate a web page but no geometry.

```
http://<host>:<port>/tasty

!!!by default this URL is:

http://localhost:9000/tasty
```

This will bring up the start-up page (see figure 6). To access a specific object you must specify the class package and the object type, separated by a colon (":") or a double-colon ("::") in case the symbol naming the type is not exported from the package. For example, for the previous presented shock-absorber model, the specification will be: **shock-absorber:assembly**, or by using the package nickname **shock** or **picasso** the specification would be **shock:assembly** or **picasso:assembly**.
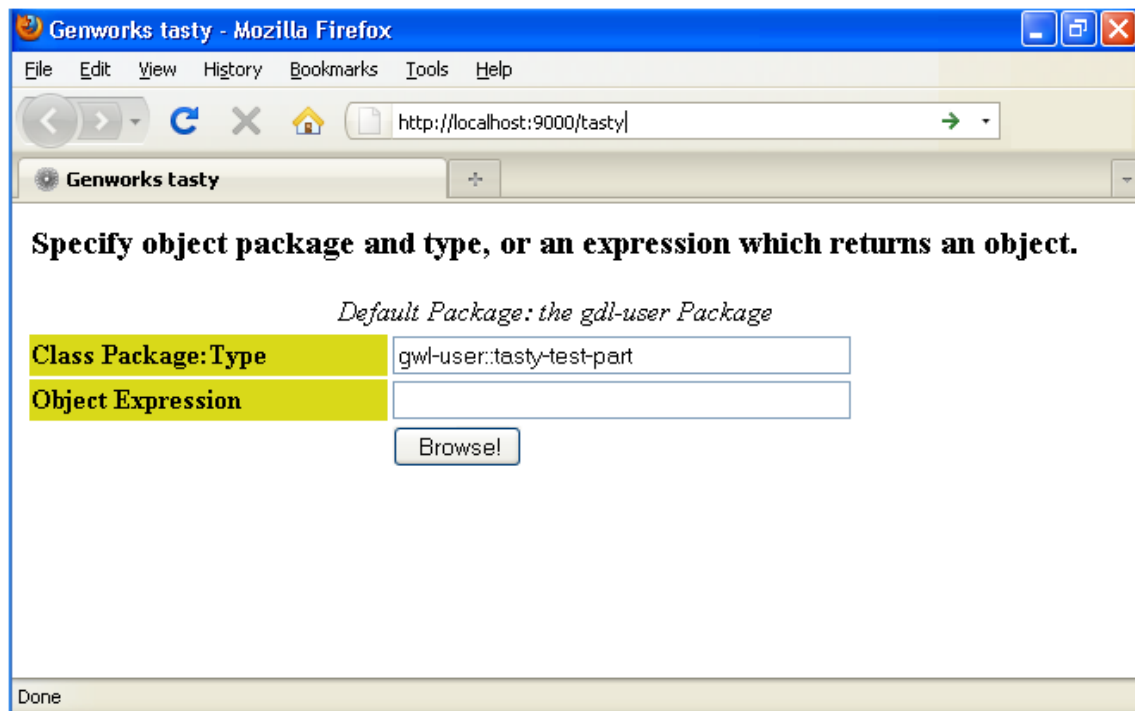
Figure 6: Tasty start up.

Note that if the **assembly** symbol had not been exported from the **:shock-absorber** package, then a double-colon would have been needed: **shock-absorber::assembly**[6].

After you specify the class package and the object type and press the "browse" button, the browser will bring up the utility interface with an instance of the specified type (see figure 7).

The utility interface by default is composed of three toolbars and three view frames (tree frame, inspector frame and viewport frame "graphical view port").

---

[6] use of a double-colon indicates dubious coding practice, because it means that the code in quesion is accessing the "internals" or "guts" of another package, which may not have been the intent of that other package's designer.
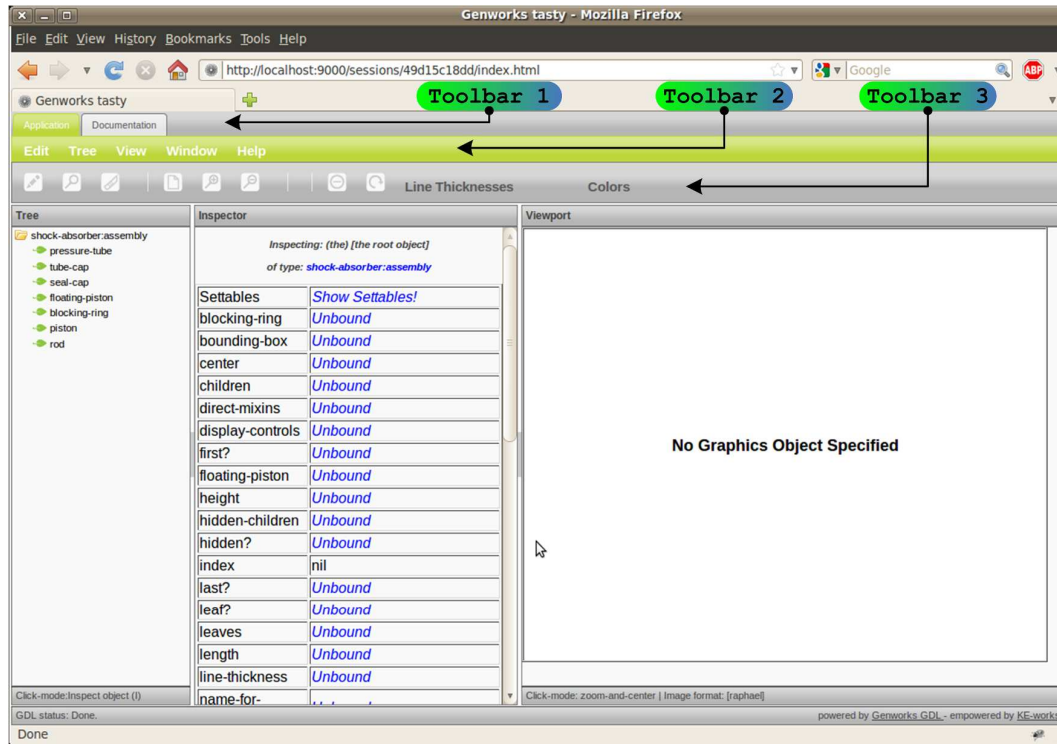
Figure 7: Tasty interface

4.7.1 The Toolbars

The first toolbar consists of two "tabs" which allow the user to select between the display of the application itself or the GenDL reference documentation.

The second toolbar is designed to select various "click modes" for objects and graphical viewing, and to customize the interface in other ways. It hosts five menus: edit, tree, view, windows and help[7].

The edit and window menu allows the user to customize the interface in various ways.

The tree menu allows the user to customize the "click mode" of the mouse (or "tap mode" for other pointing device) for objects in any of the tree, inspector, or viewport frames. The behavior follows the **select-and-match** paradigm – you first **select** a mode of operation with one of the buttons or menu items, then **match** that mode to any object in the tree frame or inspector frame by left-clicking (or tapping). These modes are as follows:

<<Graphical modes>>

Add Node (AN) — Add Node in graphics view port

Add Leaves (AL) — Add Leaves in graphics view port

Add Leaves indiv. (AL*) — Add Leaves individually (so they can be deleted individually).

Draw Node (DN) — Draw Node in graphics view port (replacing any existing).

Draw Leaves (DL) — Draw Leaves in graphics view port (replacing any existing).

Clear Leaves (DL) — Delete Leaves

---

[7] A File menu will be added in a future release, to facilitate saving and restoring of instance "snapshots" – at present, this can be done programmatically.

<<Inspect & debug modes>>

Inspect object (I) — Inspect (make the inspector frame to show the selected object).

Set self to Object (B) — Break on selected object.

Set Root to Object (SR) — Set displayed root in Tasty tree to selected object.

Up Root (UR!) — Set displayed root in Tasty tree up one level (this is grayed out if already on root).

Reset Root (RR!) — Reset displayed root in Tasty to to the true root of the tree (this is grayed out if already on root).

<<Tree frame navigation modes>>

Expand to Leaves (L) — Nodes expand to their deepest leaves when clicked.

Expand to Children (C) — Nodes expand to their direct children when clicked.

Auto Close (A) — When any node is clicked to expand, all other nodes close automatically.

Remember State (R) — Nodes expand to their previously expanded state when clicked.

The view menu allows the user to customize the graphics view port modes. These modes are as follows:

<<Viewport Actions>>

Fit to Window! — Fits to the graphics view port size the displayed objects (use after a Zoom).

Clear View! (CL!)— Clear all the objects displayed in the graphics view port.

<<Image Format>>

PNG — Sets the displayed format in the graphics view port to PNG (raster image with isoparametric curves for surfaces and brep faces).

JPEG — Sets the displayed format in the graphics view port to JPEG (raster image with isoparametric curves for surfaces and brep faces).

VRML/X3D — Sets the displayed format in the graphics view port to VRML with default lighting and viewpoint (these can be changed programmatically).

SVG/VML — Sets the displayed format in the graphics view port to SVG/VML[8] (vector graphics image with isoparametric curves for surfaces and brep faces).

<<Click Modes>>

Zoom in — Sets the mouse left-click in the graphics view port to zoom in.

Zoom out — Sets the mouse left-click in the graphics view port to zoom out.

Measure distance — Calculates the distance between two selected points from the graphics view port.

Get coordinates — Displays the coordinates of the selected point from the graphics view port.

---

[8] For complex objects with many display curves, SVG/VML can overwhelm the JavaScript engine in the web browser. Use PNG for these cases.

Select Object — Allows the user to select an object from the graphics view port (currently works for displayed curves and in SVG/VML mode only).

<<Perspective>>

Trimetric — Sets the displayed perspective in the graphics view port to trimetric.

Front — Sets the displayed perspective in the graphics view port to Front (-Y axis).

Rear — Sets the displayed perspective in the graphics view port to Rear (+Y axis).

Left — Sets the displayed perspective in the graphics view port to Left (-X axis).

Right — Sets the displayed perspective in the graphics view port to Right (+X axis).

Top — Sets the displayed perspective in the graphics view port to Top (+Z axis).

Bottom — Sets the displayed perspective in the graphics view port to Bottom (-Z axis).

Third toolbar hosts the most frequently used buttons. This buttons have tooltips which will pop up when you hover the mouse over them. However, these buttons are found in the second toolbar too, except line thickness and color buttons. The line thickness and color buttons[9] expand and contract when clicked on and allows the user to select a desired line thickness and color for the objects displayed in the graphics view port.

4.7.2 View Frames

The tree frame is a hierarchical representation of your defined object. For example for the shock-absorber **assembly** this will be as depicted in figure 8.
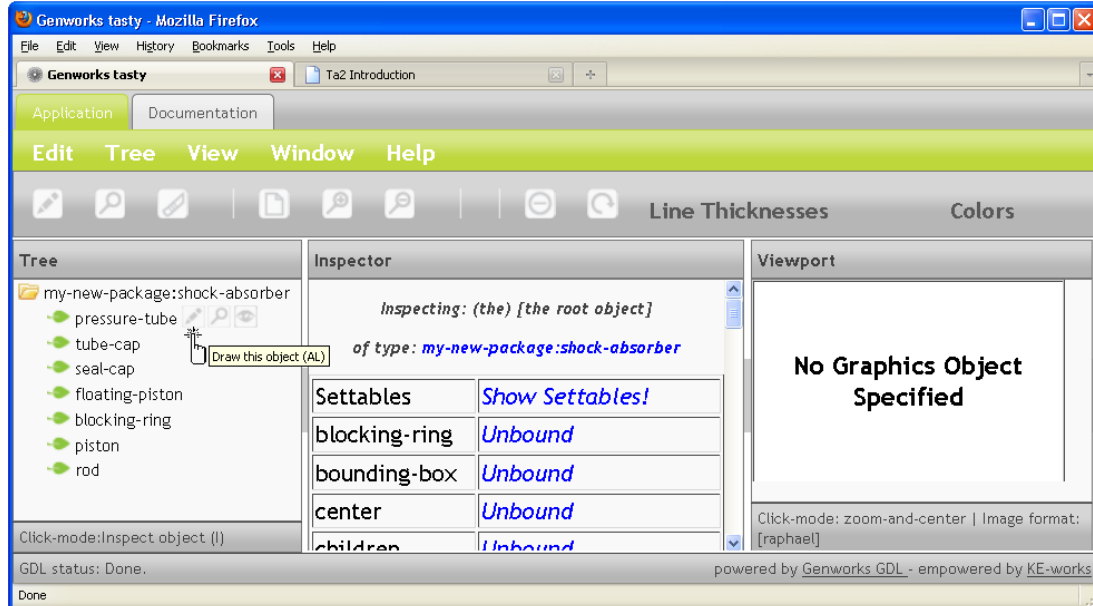
Figure 8: Tasty shock-absorber tree.

To draw the graphics (geometry) for the shock-absorber leaf-level objects, you can select the "Add Leaves (AL)" item from the Tree menu, then click the desired leaf to be displayed

---

[9] the design of the line thickness and color buttons is being refined and may appear different in your installation.

from the tree or select the *rapid* button from third toolbar which is symbolized by a pencil . Because this operation (draw leaves) is frequently used, the tree leaves has this operation directly available as a tooltip, which will pop up when you hover the mouse over them. To perform this operation on the fly, you simply have to click the pencil icon; it does not require a second click on the leaf in the tre (see figure 8).

The "on the fly" feature is available also for "inspect object"(second icon when you hover the mouse over a leaf) and "highlight object" (third icon when you hover the mouse over a leaf). For safety reasons highlight object requires a second click on the leaf (a confirmation that the user wants to remove that leaf from the graphics view port).

The inspector frame allows the user to inspect (and in some cases modify) object instance being inspected. Following are some examples of how the inspector frame may be used.

To prepare for the first example, we will modify the assembly definition by adding a settable **input-slot** for the piston radius. In GenDL, the **:input-slots** are made up of a list, each of whose elements is either a symbol (for required inputs) or a list expression beginning with a symbol (for optional inputs). In either case, the symbol represents a value which can be supplied either:

(a) into the toplevel object of an object hierarchy, when the object is instanted, or

(b) into a child object, using a **:objects** specification by definition in the parent.

Inputs are specified in the definition either as a symbol by itself (for required inputs), or as an expression whose **first** is a symbol and whose **second** is an expression which returns a value which will be the default value for the input slot.

Optionally, additional keywords can be supplied:

- the keyword **:defaulting**, which indicates that if a slot by this name is contained in any ancestor object's list of :trickle-down-slots[10], the value from the ancestor will take precedence over the local default expression.

- The keyword **:settable**, which indiates that the default value of this slot can be "bashed," or overridden, at runtime after the object has been instantiated, using the special object function **set-slot!**[11].

For this example, we will supply piston-radius as an input symbol with a default expression and with the optional keyword **:settable**. We will also pass the piston-radius down into the child **piston** object, rather than using a hard-coded value of 12 as previously. The new assembly definition is now:

---

[10] trickle-down slots will be introduced later

[11] Any other slots depending on settable slots (directly or indirectly) will become unbound when the **:set-slot!** function is called to change the slot's value, and these will be recomputed the next time they are demanded.

```
;;--assembly.lisp--
;; this is a new version V0.1 of the file  assembly.lisp
(in-package :shock-absorber)

(define-object  assembly (base-object)
  :input-slots ((piston-radius 12 :settable)) ;;;----modification
  :computed-slots ()
  :objects
  ((pressure-tube :type 'cone
                  :center (make-point 0 70 0)
                  :length 120
                  :radius-1 13
                  :inner-radius-1 12
                  :radius-2 13
                  :inner-radius-2 12)

   (tube-cap :type 'cone
             :center (make-point 0 5 0)
             :length 10
             :radius-1 5
             :inner-radius-1 0
             :radius-2 13
             :inner-radius-2 0)

   (seal-cap :type 'cone
             :center (make-point 0 135 0)
             :length 10
             :radius-1 13
             :inner-radius-1 2.5
             :radius-2 5
             :inner-radius-2 2.5)

   (floating-piston :type 'cylinder
                    :center (make-point 0 35 0)
                    :radius 12
                    :length 10)

   (blocking-ring :type 'cone
                  :center (make-point 0 42.5 0)
                  :length 5
                  :radius-1 12
                  :inner-radius-1 10
                  :radius-2 12
                  :inner-radius-2 10)

   (piston :type 'cylinder
           :center (make-point 0 125 0)
           :radius (the piston-radius) ;;;----modification
           :length 10)

   (rod :type 'cylinder
        :center (make-point 0 175 0)
        :radius 2.5
        :length 90))

  :hidden-objects ()
  :functions ()
  :methods ())
```

34

In this new version "V0.1" of the assembly, the piston radius is a settable slot, and its value can be modified (i.e. "bashed") as desired, either programmatically from the command-line, in an end-user application, or from **Tasty**.
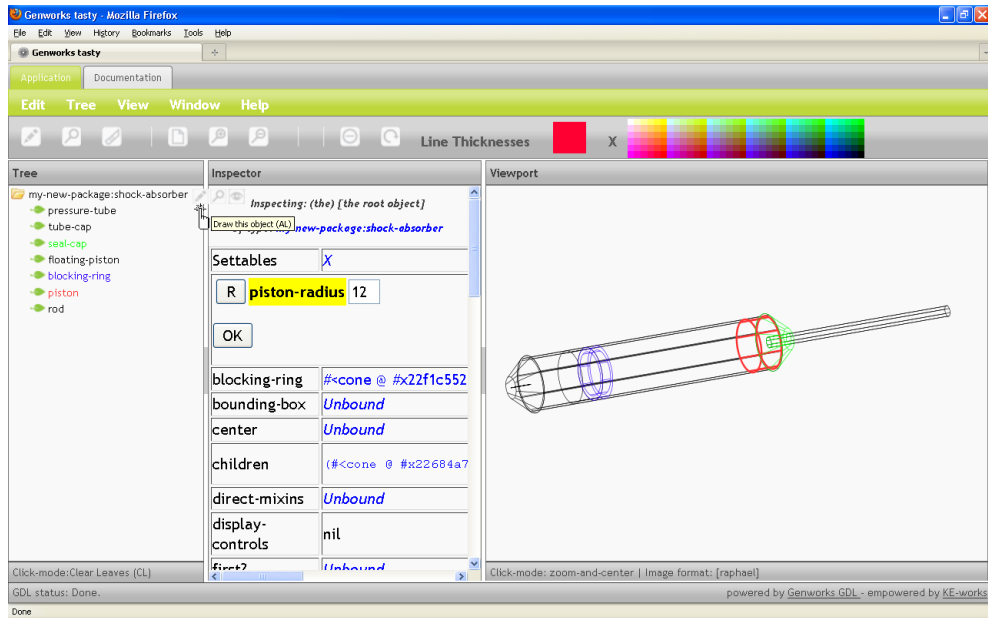
Figure 9: Tasty inspector.
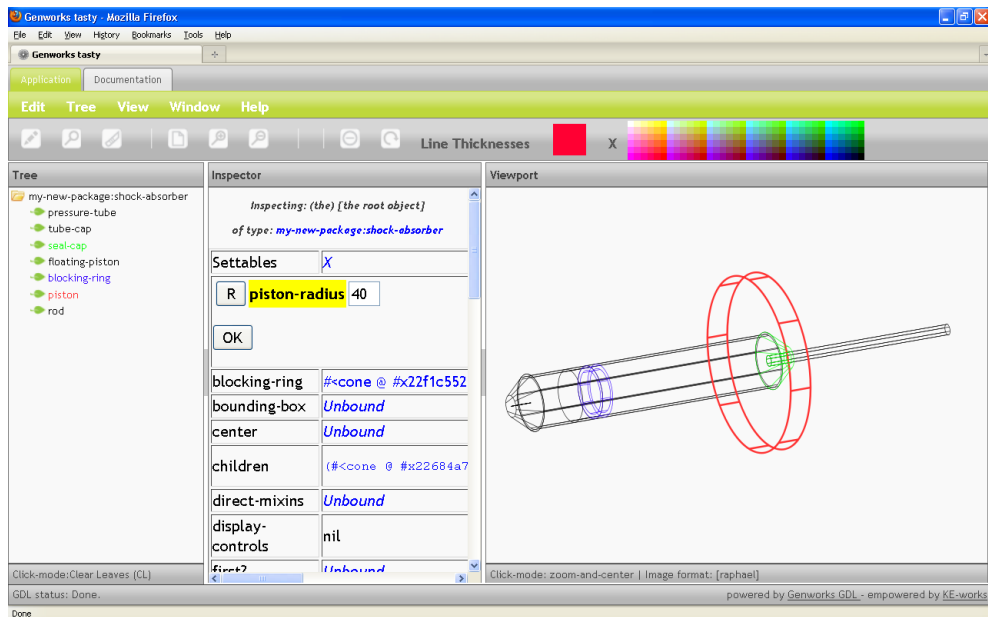
Figure 10: Tasty settable slots.

To modify the value in Tasty: select "Inspect" mode from the Tree menu, then select the root of the **assembly** tree to set the inspector on that object (see figure 9). Once the inspector is set to this object, it is possible to expand its settable slots by clicking on the

"Show Settables!" link. (use the "X" link to collapse the settable slots view). When the settable slots area is open the user may set the values as desired by inputting the new value and pressing the OK button (see figure 10).

As it can be observed from Figure 10, there is no dependency between the piston radius and the rest of the shock-absorber objects (components). If such a dependency is desired, the definition has to be modified to include it. For the moment the piston radius will be considered as the leading parameter, the rod radius constant and the piston Y position will be added as a settable input slot. In addition to this dimensional dependency, two computed slots will be added to the previous code in order to compute the volume A and B (see figure 5) as follows:

```
;;--assembly.lisp--
;; this is a new version V0.2 of the file assembly.lisp
(in-package :shock-absorber)
(define-object  assembly (base-object)

  :input-slots
  ((piston-radius 12 :settable)
   (piston-y-position 125 :settable))


  :computed-slots
  ((volume-a  (* 2 pi (- (* (the piston-radius)
                            (- (- (get-y (the piston center))
                                  (get-y (the floating-piston center)))
                               (/ (+ (the floating-piston length)
                                     (the piston length)) 2)))
                         (*  (the blocking-ring  length)
                             (- (the piston-radius)
                                (the blocking-ring  inner-radius-1))))))

   (volume-b (* (- (- (get-y (the floating-piston center))
                      (/ (the floating-piston length)2))
                   (+ (get-y (the tube-cap center))
                      (/ (the tube-cap length)2)))
                (* 2 pi (the  floating-piston radius )))))

  :objects
  ((pressure-tube :type 'cone
                  :center (make-point 0 70 0)
                  :length 120
                  :radius-1 (+ 1 (the piston-radius))
                  :inner-radius-1 (the piston-radius)
                  :radius-2 (+ 1 (the piston-radius))
                  :inner-radius-2 (the piston-radius))

   (tube-cap :type 'cone
             :center (make-point 0 5 0)
             :length 10
             :radius-1 5
             :inner-radius-1 0
             :radius-2 (+ 1 (the piston-radius))
             :inner-radius-2 0)
```

```
  (seal-cap :type 'cone
            :center (make-point 0 135 0)
            :length 10
            :radius-1 (+ 1 (the piston-radius))
            :inner-radius-1 2.5
            :radius-2 5
            :inner-radius-2 2.5)

  (floating-piston :type 'cylinder
                   :center (make-point 0 35 0)
                   :radius (the piston-radius)
                   :length 10)

  (blocking-ring :type 'cone
                 :center (make-point 0 42.5 0)
                 :length 5
                 :radius-1 (the piston-radius)
                 :inner-radius-1 (* 0.8 (the piston-radius))
                 :radius-2 (the piston-radius)
                 :inner-radius-2 (* 0.8 (the piston-radius)))

  (piston :type 'cylinder
          :center (make-point 0 (the piston-y-position ) 0)
          :radius (the piston-radius)
          :length 10)

  (rod :type 'cylinder
       :center (make-point 0 175 0)
       :radius 2.5
       :length 90))
:hidden-objects ()
:functions ()
:methods ())
```

The dimensional dependency is a straight forward operation similar to the **piston-radius** being passed in for the **piston radius** in v0.1. The only difference in this case is that, for some of the objects, a coefficient was added to account for a wall thickness (e.g. 0.8) which is multiplied with the **piston-radius**.

Regarding the volume A and B, these are computed in such a manner to account for any geometry variation, including relative movement of the piston and floating-piston. While the Lisp expression may not be immediately obvious, in practice it is easy to build up such an expression incrementally at the prompt level until a valid expression is obtained. See example below:

Assuming that the **assembly version V0.2** is compiled and loaded, you can update **Tasty** by clicking the "update" button ⟳.To start implementing the volume-a sequentially (although the volume-a and volume-b is already implemented in V0.2, this is a good exercise to go through), select the click mode to "break" ⊖ and "set self" to the **shock-absorber:assembly** by clicking the assembly root. This will allow you to interact directly with the shock-absorber assembly at the prompt level where you can type the following:

```
!!! see figure 11 for more details regarding R, H,  HFP, HSC, etc.

gdl-user(38): (setq R (the piston-radius))

12

gdl-user(39): (setq rsc (the blocking-ring  inner-radius-1))

9.600000000000001

gdl-user(40): (setq HSC  (the blocking-ring  length))

5

gdl-user(41): (setq HFP  (the floating-piston length))

10

gdl-user(42): (setq HP  (the piston length))

10

gdl-user(43): (setq H (- (- (get-y (the piston center))
                         (get-y (the floating-piston center)))
                      (/ (+ HFP HP) 2)))
80.0
```

Mathematically, volume-a is defined as follows:

$$V_A = [2\pi RH - (2\pi RH_{SC} - 2\pi r_{SC}H_{SC})]$$

or

$$V_A = 2\pi[RH - H_{SC}(R - r_{SC})]$$

Where H is:

$$H = [(Y^* - Y) - 2^{-1}(H_{FP} + H_P]$$
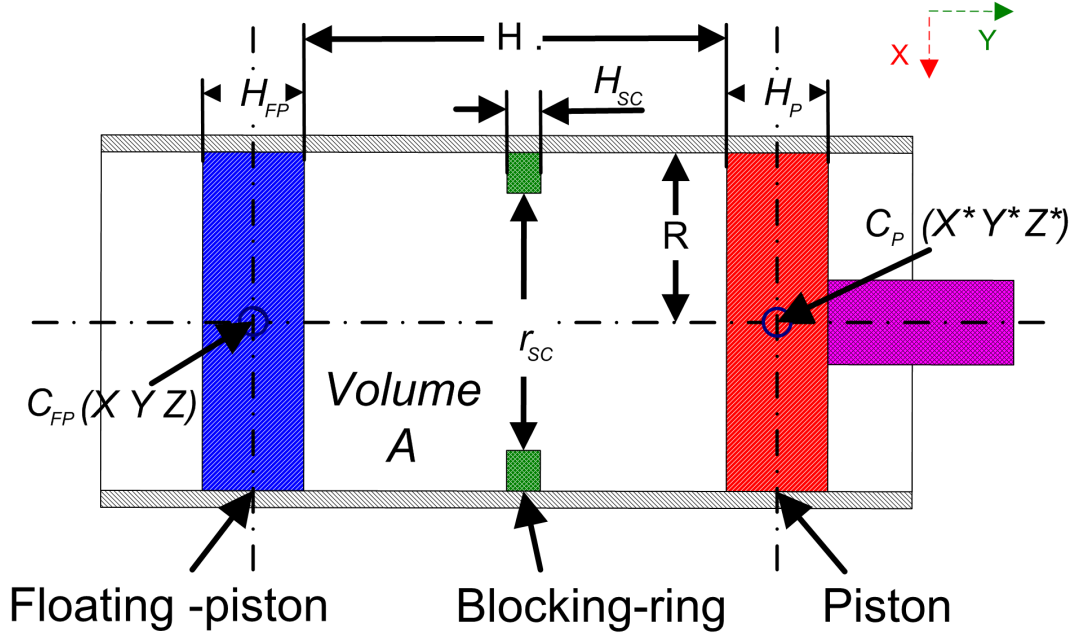
Figure 11: The definition of volume-a and volume-b.

In GenDL S-expression (i.e. Lisp) notation, the formula for volume-a is equivalent to:

```
gdl-user(35):  (* 2 pi (- (* R H) (* HSC (- R rsc))))

;; or expanded as implemented in the shock-absorber assembly version V0.2:

gdl-user(46): (* 2 pi (- (* (the piston-radius)
                            (- (- (get-y (the piston center))
                                  (get-y (the floating-piston center)))
                               (/ (+ (the floating-piston length)
                                     (the piston length)) 2)))
                         (* (the blocking-ring length)
                            (- (the piston-radius)
                               (the blocking-ring  inner-radius-1)))))))

;; for volume-b the approach is similar to volume-a
```

Note that the implementation of **volume-a** and **volume-b** in the shock-absorber assembly are non-settable **computed-slots**. This means that they are, in fact, messages of the object which are strictly computed based on their default expression.

Computed-slots will only be computed when called ("demanded"), then the resultant values will be cached (memorized)in memory. Only if another slot on which they depend becomes modified will they become unbound, then their values will be recomputed from their expressions when demanded next time.

At prompt you can ask for **volume-a** and **volume-b** value by typing:

```
gdl-user(47): (the volume-a)

5956.459671206248

or

gdl-user(48): (the volume-b)

1507.9644737231006
```

To test the volume dependency on relative movement of the piston, set the value of **piston-y-position** to 60 (see figure 12) and ask at prompt the value of volume-a.

```
gdl-user(52): (the volume-a)

2563.539605329271
```
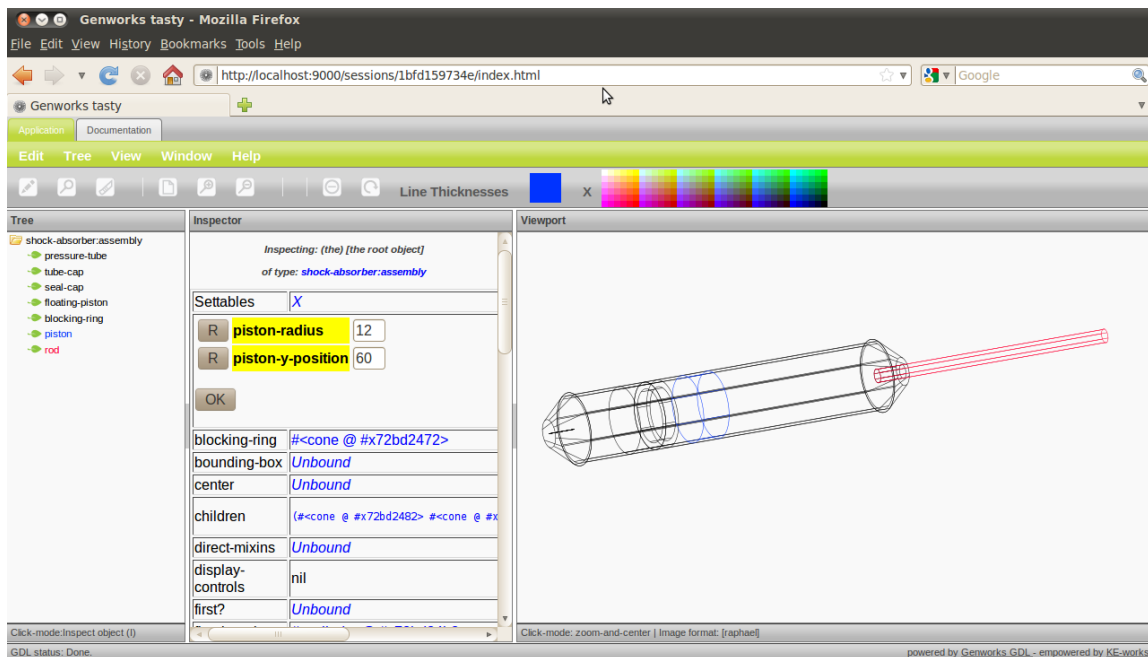

Figure 12: Relative movement of the piston in the Y direction

Adding a relative movement to the piston by using the slot **piston-y-position**, will implicitly alter the dimensional connection in the Y direction between the piston and the rod - see figure 12. To account for this, a dimensional dependency must be added in the y direction. The best practice for this particular case is to define a piston assembly as a separate object definition (and in a separate file — piston-assembly.lisp) which will contain both the definition of piston/rod and the dimensional dependency as presented below:

```
;;-- piston-assembly.lisp--

(in-package :shock-absorber)

(define-object piston-assembly (base-object)

  :input-slots (piston-length piston-radius rot-length piston-y-position)

  :computed-slots
  ((rot-center (make-point 0 (+ (get-y (the piston center ))
                                (/ (+ (the piston-length)
                                      (the rot-length)) 2)) 0)))

  :objects
  ((piston :type 'cylinder
           :center (make-point 0 (the piston-y-position ) 0)
           :radius (the piston-radius)
           :length (the piston-length ))

   (rod :type 'cylinder
        :center (the rot-center)
        :radius 2.5
        :length (the rot-length)))))
```

```
;;--assembly.lisp-- this is a new version V0.3 of the file
;; assembly.lisp

(in-package :shock-absorber)

(define-object  assembly (base-object)

  :input-slots

  ((piston-radius 12 :settable)
   (piston-y-position 125 :settable)
   (piston-length 10 :settable)
   (rot-length 90 :settable))

 :computed-slots
 ((volume-a
   (* 2 pi (- (* (the piston-radius)
                 (- (- (get-y (the rod-piston-assembly piston center))
                       (get-y (the rod-piston-assembly floating-piston center)))
                    (/ (+ (the rod-piston-assembly  floating-piston length)
                          (the rod-piston-assembly  piston length)) 2)))
              (*  (the blocking-ring  length)
                  (-  (the piston-radius)
                      (the blocking-ring  inner-radius-1)))))))))
```

```
(volume-b
 (* (- (- (get-y (the rod-piston-assembly floating-piston center))
          (/ (the  rod-piston-assembly floating-piston length)2))
       (+ (get-y (the tube-cap center)) (/ (the tube-cap length)2)))
    (* 2 pi (the  rod-piston-assembly  floating-piston radius )))))

:objects
((pressure-tube :type 'cone
                :center (make-point 0 70 0)
                :length 120
                :radius-1 (+ 1 (the piston-radius))
                :inner-radius-1 (the piston-radius)
                :radius-2 (+ 1 (the piston-radius))
                :inner-radius-2 (the piston-radius))
 (tube-cap :type 'cone
           :center (make-point 0 5 0)
           :length 10
           :radius-1 5
           :inner-radius-1 0
           :radius-2 (+ 1 (the piston-radius))
           :inner-radius-2 0)
 (seal-cap :type 'cone
           :center (make-point 0 135 0)
           :length 10
           :radius-1 (+ 1 (the piston-radius))
           :inner-radius-1 2.5
           :radius-2 5
           :inner-radius-2 2.5)
 (floating-piston :type 'cylinder
                  :center (make-point 0 35 0)
                  :radius (the piston-radius)
                  :length 10)
 (blocking-ring :type 'cone
                :center (make-point 0 42.5 0)
                :length 5
                :radius-1 (the piston-radius)
                :inner-radius-1 (* 0.8 (the piston-radius))
                :radius-2 (the piston-radius)
                :inner-radius-2 (* 0.8 (the piston-radius)))

 (rod-piston-assembly :type 'piston-assembly
                      :pass-down (piston-radius piston-y-position
                                  piston-length rot-length)))
```

In this version of the assembly, **:pass-down** was used to pass the **piston-radius**, **piston-y-position**, **piston-length**, and **rot-length** into the child piston-assembly. **:pass-down** is a shorthand way of passing inputs into a child object where the input-slots in the child definition have the same names as the messages in the parent. The following child specification would be exactly equivalent (but more verbose) to the above **:pass-down** clause:

```
(rod-piston-assembly :type 'piston-assembly
                     :piston-radius (the piston radius)
                     :piston-y-position (the piston-y-position)
                     :piston-length (the piston-length)
                     :rot-length (the rot-length))
```

Basically, the shock-absorber assembly is fully constrained in the above example, except for the floating piston position. In a real application, the piston and floating piston relative movement in the Y direction is determined by the absorbed energy at shock. However, we will not detail the full shock absorber physics in this manual. Instead, we will provide an "ideal case" example by adding a function to determine the equilibrium between the two pistons when loaded. The following ideal case will be considered:

The shock-absorber volume-a is filled with an ideal gas and experiences an isothermal transformation T=const, pV=const. In volume-b a incompressible liquid is used. Based on this assumptions the volume-a variation at load can be defined as follows:

$$pV = \ nRT = \ ct.$$

$$p_A V_A = \ p_{A0} V_{A0}$$

If :

$$p_A > 1.101325 \ X \ 10^5 \ N/m^2$$

Then:

$$V_{A0} = \left(\frac{p_A V_A}{p_A + \left(\frac{mg}{2\pi R} \right)} \right)$$

$$Y_0^* = Y^* - \left(\frac{1}{2\pi R} \right) (V_A - V_{A0})$$

Based on this, a function can be defined as follows, which will automatically compute the new piston y position when loaded:

```
;;-- main.lisp--
;; this is a new version V0.4 of the file main.lisp

(in-package :shock-absorber)

(define-object  assembly (base-object)

   :input-slots
  ((piston-radius 12 :settable);;[SI]
   (piston-y-position 125 :settable);;[SI]
   (piston-length 10 :settable);;[SI]
   (rot-length 90 :settable);;[SI]
   (pressure-a 1.5e5 :settable);;[SI]
   (loaded-mass 10e5 :settable);;[SI] )

  :computed-slots
  ((volume-a
    (* 2 pi (- (* (the piston-radius)
                  (- (- (get-y (the rod-piston-assembly piston center))
                        (get-y (the rod-piston-assembly floating-piston center)))
                     (/ (+ (the rod-piston-assembly  floating-piston length)
                           (the rod-piston-assembly  piston length)) 2)))
             (*  (the blocking-ring  length)
                 (-  (the piston-radius)
                     (the blocking-ring  inner-radius-1)))))))

   (volume-a0 (/ (* (the pressure-a) (the volume-a ))
                 (+ (the pressure-a) (/ (* (the loaded-mass) 9.8)
                                        (* 2 pi (the piston-radius))))))

   (volume-b
    (* (- (- (get-y (the rod-piston-assembly floating-piston center))
             (/ (the  rod-piston-assembly floating-piston length)2))
          (+ (get-y (the tube-cap center)) (/ (the tube-cap length)2)))
       (* 2 pi (the  rod-piston-assembly  floating-piston radius )))))

  :objects
  ((pressure-tube :type 'cone
                  :center (make-point 0 70 0)
                  :length 120
                  :radius-1 (+ 1 (the piston-radius))
                  :inner-radius-1 (the piston-radius)
                  :radius-2 (+ 1 (the piston-radius))
                  :inner-radius-2 (the piston-radius))

   (tube-cap :type 'cone
             :center (make-point 0 5 0)
             :length 10
             :radius-1 5
             :inner-radius-1 0
             :radius-2 (+ 1 (the piston-radius))
             :inner-radius-2 0)

   (seal-cap :type 'cone
             :center (make-point 0 135 0)
             :length 10
             :radius-1 (+ 1 (the piston-radius))
             :inner-radius-1 2.5
             :radius-2 5
             :inner-radius-2 2.5)
```

```
   (floating-piston :type 'cylinder
                     :center (make-point 0 35 0)
                     :radius (the piston-radius)
                     :length 10)

   (blocking-ring :type 'cone
                  :center (make-point 0 42.5 0)
                  :length 5
                  :radius-1 (the piston-radius)
                  :inner-radius-1 (* 0.8 (the piston-radius))
                  :radius-2 (the piston-radius)
                  :inner-radius-2 (* 0.8 (the piston-radius)))

   (rod-piston-assembly :type 'piston-assembly
                        :pass-down (piston-radius piston-y-position
                                                  piston-length rot-length))

   (loaded-piston :type 'piston-assembly
                  :piston-y-position (l_p_p (the piston-radius)
                                            (the piston-y-position)
                                            (the volume-a)
                                            (the volume-a0))
                  :pass-down (piston-radius
                              piston-length
                              rot-length)))

 :functions ((l_p_p ;;;loaded_piston_position
              (R Y-star Va Va0)
              (- Y-star (* (/ 1 (* 2 pi R)) (- Va Va0))))))
```

5 GenDL Output Formats and Drawing

5.1 GenDL Output Formats

In GenDL a series of output formats can be used in conjunction with the "with-format" macro to produce customized output from GenDL objects, as shown in this chapter.

If you create new GenDL objects, it is possible to create output-functions for your new object for various formats by using the define-lens macro. You can also define your own output-formats using the define-format macro. Please see the reference documentation for define-lens for more details.

GenDL has several output-formats and lenses already defined, which you can use via the **with-format** macro. The basic syntax for **with-format** is as follows:

Assume you have an object **my-obj** which contains geometry in its leaves. By setting self **my-obj** you can output this geometry in Iges format using:

```
(with-format (iges "/tmp/try.iges") (write-the cad-output-tree))
```

Assume you have an object **my-obj** which is itself some geometry (for example, a surface or brep solid). By setting self **my-obj** you can output its geometry using:

```
(with-format (iges "/tmp/try.iges") (write-the cad-output))
```

To summarize, **with-format** takes first a literal list with the name of the desired output-format followed by a stream or file name, followed by any number of optional keyword arguments for the format slots. Next it takes a body of code. The body uses the **write-the** macro to invoke an output-function.

The most frequently used outputs formats in GenDL are: PDF, DXF, VRML and IGES.

5.1.1 Portable Document Format (PDF)

PDF is the native graphics and document format of GenDL and as such is the most developed. Most graphical and textual objects in GenDL have PDF lenses defined for them with at least a "cad-output" method. The term "cad-output" is somewhat dated, as the output could refer to a paper report with charts and graphs as much as a drawing of mechanical parts, so these names may be refined in future GenDL releases.

Available slots for the PDF output-format and their defaults:

(page-width 612)

(page-length 792)

(view-transform (getf *standard-views* :top))

(view-center (make-point 0 0 0)) ;; in page coordinates

(view-scale 1)

Please see for examples.

5.1.2 Drawing Exchange Format (DXF)

DXF is AutoCAD's drawing exchange format. Currently most of the geometric objects and drawing and text objects have a cad-output method for this output-format. The GenDL DXF writer currently outputs a relatively old-style AutoCAD Release 11/12 DXF header. In future GenDL versions this will be switchable to be able to target different release levels of the DXF format.

Note that the DXF writer is currently 2D in nature and therefore it is only intended to work for 2D objects in GenDL, i.e. drawings.

Available slots for the DXF output-format and their defaults:

(view-transform (getf *standard-views* :top))

(view-center (make-point 0 0 0)) ;; in page coordinates

(view-scale 1)

Please see Section 5.4 [Drawing and Output Formats Examples], page 49 for examples.

5.1.3 Virtual Reality Modeling Language (VRML)

GenDL currently contains a rudimentary VRML writer which maps GenDL primitive geometric objects (boxes, spheres, etc) into their VRML equivalents. Future GenDL releases will include surfaces and curves to be outputted in the pure BSpline format which is supported by BSContact and some other VRML and X3D viewers. It will also include more object types and mechanisms to specify viewpoints, lighting, textures, etc.

Please see Section 5.4.6 [Example of a VRML Output Format], page 58 for a detailed exemplification

5.1.4 Initial Graphics Exchange Specification (IGES)

GenDL platform supplies an IGES output-format with the optional GenDL NURBS Surfaces Facility. The IGES format will also convert certain GenDL wireframe primitives into appropriate curve and surface objects for IGES output (e.g. l-lines into linear-curves, arcs into arc-curves).

Here are some examples of simple use:

```
;;
;; Writes the leaves of the current ``self'' object:
;;
(with-format (iges "/tmp/try.iges" :units :millimeters)
  (write-the cad-output-tree))


;;
;; Write the current ``self'' object but not children or leaves:
;;
(with-format (iges "/tmp/try.iges")
  (write-the cad-output))
```

- *Brep solids representation in iges.*

  For breps, you can output them as individual breps or as a bag of the faces (trimmed surfaces) making up the brep.

  This is done with the format slot :breps-format, which can have the value :breps (the default) or :surfaces, e.g.

  ```
  (with-format (iges "/tmp/try.iges" :breps-format :surfaces)  ...) ;; or

  (with-format (iges "/tmp/try.iges" :breps-format :breps)  ...)
  ```

- *Units in iges.*

  The iges units can be specified with the format variable :units, e.g:

  ```
  (with-format (iges "/tmp/try.iges" :units :millimeters) ...) ;; or

  (with-format (iges "/tmp/try.iges" :units :feet) ...)
  ```

  The allowed values for :units are the keyword symbols in the following list:

  (:inches :millimeters :feet :miles :meters :kilometers :mils :microns :centimeters :microinches :no-units-specified)

  The default is :inches.

5.1.5 HTML Format

The HTML output format is used extensively throughout GWL for generating dynamic web page hierarchies corresponding to GenDL object hierarchies. Please see Chapter 6 [Custom User Interfaces in GenDL], page 61 for extensive examples of using this output format.

5.2 Base Drawing

Base-drawing is the fundamental object which represents a physical drawing on a piece of paper. Although the metaphor is a piece of paper, a drawing can also be displayed in a GWL web application as an embedded, clickable, zoomable, pannable image. An empty drawing does not do much of anything. It has to contain at least one object of type base-view to be useful.

A Drawing is generally output by itself, as one whole unit. GenDL currently does not support outputting of individual parts of a drawing, or children of a drawing. The Drawing can have as many children and other descendants as you like. Only those children which are of type base-view (see Section 5.3.1 [Introduction to Base-view], page 48) will be included when the drawing is output.

The main user-controllable :input-slots to a drawing are page-length and page-width. These are assumed to be in points, where there are 72 points per inch (about 28 points per cm) for purposes of printed output. The default page size is for US Letter paper, or 8.5 inches (612 points) for page-width, and 11 inches (792 points) for page-length. Page-height is essentially the thickness of the page, which is always zero (0).

5.3 Base View

5.3.1 Introduction to Base-view

Base-view represents a flat rectangular section of a drawing, and is a window onto a set of 3D and/or 2D geometric objects transformed and scaled in a specified way. The objects can be auto-fit to the page, or scaled and translated manually with user-specified inputs.

A base-view by itself does not have any defined behavior in GenDL. It must be contained as a child object of a base-drawing.

See the reference documentation for base-view for detailed explanations of each of the input-slots and other messages. Below is an overview of the common ones:

5.3.2 The Objects to be Displayed in a Base-view

There are three main input-slots for base-view which specify what objects are to be included in a view:

- *objects* - a list of GenDL objects. These objects will be displayed in each view by default. Note that these objects are taken directly – the children or leaves of these objects are not displayed (n.b. this is analogous to the cad-output output-function). These objects are defined in the normal 3D "world" coordinate system, but will be transformed and scaled according to the properties of the base-view.

- *object-roots* - a list of GenDL objects whose _leaves_ will be displayed in the view (n.b. this is analogous to the cad-output-tree output-format). These objects are defined in the normal 3D "world" coordinate system, but will be transformed and scaled according to the properties of the base-view.

- *annotation-objects* - a list of GenDL objects (usually 2D objects such as dimensioning or text primitives) which you want to display in the view. These objects are defined in the coordinate system of the view, and are not scaled or transformed (so, for example, their size will remain constant regardless of the scale of the base-view).

5.3.3 The Properties of a Base-view

The common user-specified properties for a base view are:

- *view-scale* - a Number which specifies a factor to convert from model space to drawing space (in points). If you do not specify this, it will be computed automatically so as to fit all objects within the base-view. NOTE that if this is left to be auto-computed, then you CANNOT normally refer to the view-scale from within any of the objects or object-roots passed into the view, as this would cause a circular reference. If you would like to override this restriction, you can include the object which refers to the view-scale in the view's list of immune-objects (documented in the reference materials).

- *view-center* - a 3D point in the model space which should become the center of the base-view. If you do not specify this, it will be computed automatically so as to center all the objects within the view.

- *projection-vector* - a 3D vector which represents the line of a camera looking onto the objects in model space.

- *left-margin* - Number which allows the left (and right) margins to be expanded.

- *front-margin* - Number which allows the front (and rear) margins to be expanded.

5.4 Drawing and Output Formats Examples

Note that the robot-assembly is contained in

(translate-logical-pathname "sys:src;demos;robot;")

and you should execute the following command before proceeding:

(cl-lite (translate-logical-pathname "sys:src;demos;robot;"))

this will insure that the robot example is loaded and compiled.

5.4.1 Example of a Base-drawing with a Contained Base-view

```
(in-package :gdl-user)
(define-object robot-drawing (base-drawing)
  :objects
  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :object-roots (list (the robot)))
   (robot :type 'robot::assembly
          :hidden? t)))
```

Set self to the robot-drawing and you will be able to output this drawing as a PDF file as follows:

```
(with-format (pdf "/tmp/robot-drawing.pdf")
  (write-the-object (make-object 'robot-drawing) cad-output))
```

and as DXF with:

```
(with-format (dxf "/tmp/robot-drawing.dxf")
  (write-the-object (make-object 'robot-drawing) cad-output))
```

Also you can test it in Tasty by instantiating robot-drawing in a Tasty session, and invoking the Add Node (AN) action on the root object. Be sure to set the Tasty view to top. Bellow is an illustration of the outputted file:
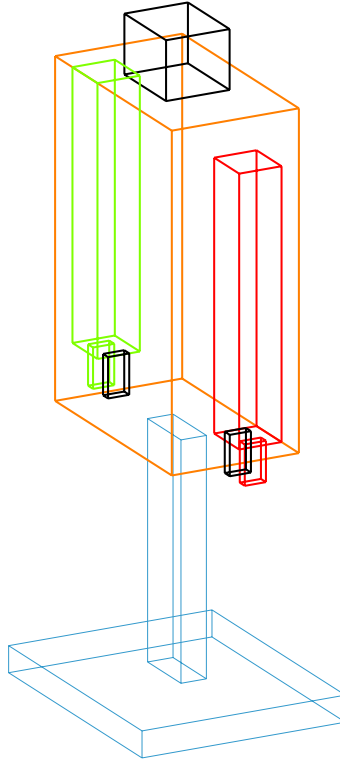
5.4.2 Example of a Base-drawing with some Dimensions

Note that this example has the main 3D geometry in a separate branch from the drawing itself:

```
(in-package :gdl-user)
(define-object box-with-drawing (base-object)

  :objects
  ((drawing :type 'dimensioned-drawing
            :objects (list (the box) (the length-dim)))

   (length-dim :type 'horizontal-dimension
               :start-point (the box (vertex :rear :top :left))
               :end-point (the box (vertex :rear :top :right)))

   (box :type 'box
        :length 10 :width 20 :height 30)))

(define-object dimensioned-drawing (base-drawing)
  :input-slots (objects)

  :objects
  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :objects (the objects))))
```

Set self to the box-with-drawing and you will be able to output this drawing as a PDF file as follows:

```
(with-format (pdf "/tmp/dimensioned-drawing.pdf")
  (write-the-object (make-object 'box-with-drawing)
                    drawing cad-output))
```

and as DXF with:

```
(with-format (dxf "/tmp/dimensioned-drawing.dxf")
  (write-the-object (make-object 'box-with-drawing)
                    drawing cad-output))
```

Also you can probe it in Tasty by instantiating box-with-drawing in a Tasty session, and invoking the Add Node (AN) action on the drawing child object. Be sure to set the Tasty view to top. Bellow is an illustration of the outputted file:
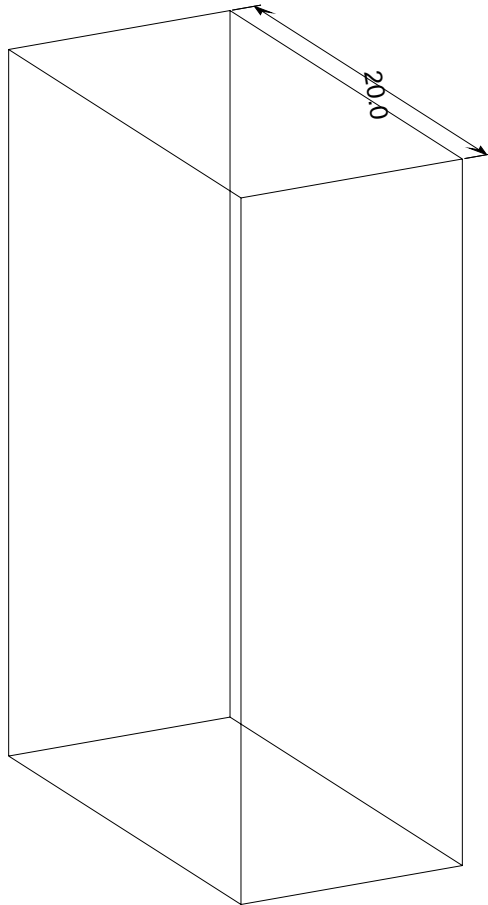
5.4.3 Example of a Base-drawing with two Views

Now we give an example of a drawing with two separate views, one trimetric and one top:

```
(in-package :gdl-user)
(define-object box-with-two-viewed-drawing (base-object)

  :objects

  ((drawing :type 'two-viewed-drawing
            :objects (list (the box) (the length-dim)))

   (length-dim :type 'horizontal-dimension
               :start-point (the box (vertex :rear :top :left))
               :end-point (the box (vertex :rear :top :right)))

   (box :type 'box
        :length 10 :width 20 :height 30)))

(define-object two-viewed-drawing (base-drawing)

  :input-slots (objects)

  :objects

  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :length (half (the length))
              :center (translate (the center)
                        :rear (half (the-child length)))
              :objects (the objects))

   (top-view :type 'base-view
             :projection-vector (getf *standard-views* :top)
             :length (half (the length))
             :center (translate (the center)
                       :front (half (the-child length)))
             :objects (the objects))))
```

Set self to the box-with-two-viewed-drawing and you will be able to output this drawing as a PDF file as follows:

```
(with-format (pdf "/tmp/two-viewed-drawing.pdf")
   (write-the-object (make-object 'box-with-two-viewed-drawing)
                     drawing cad-output))
```

and as DXF with:

```
(with-format (dxf "/tmp/two-viewed-drawing.dxf")
   (write-the-object (make-object 'box-with-two-viewed-drawing)
                     drawing cad-output))
```

Also you can probe it in Tasty by instantiating box-with-two-viewed-drawing in a Tasty session, and invoking the Add Node (AN) action on the drawing child object. Be sure to set the Tasty view to top. Bellow is an illustration of the outputted file:

20.0

54
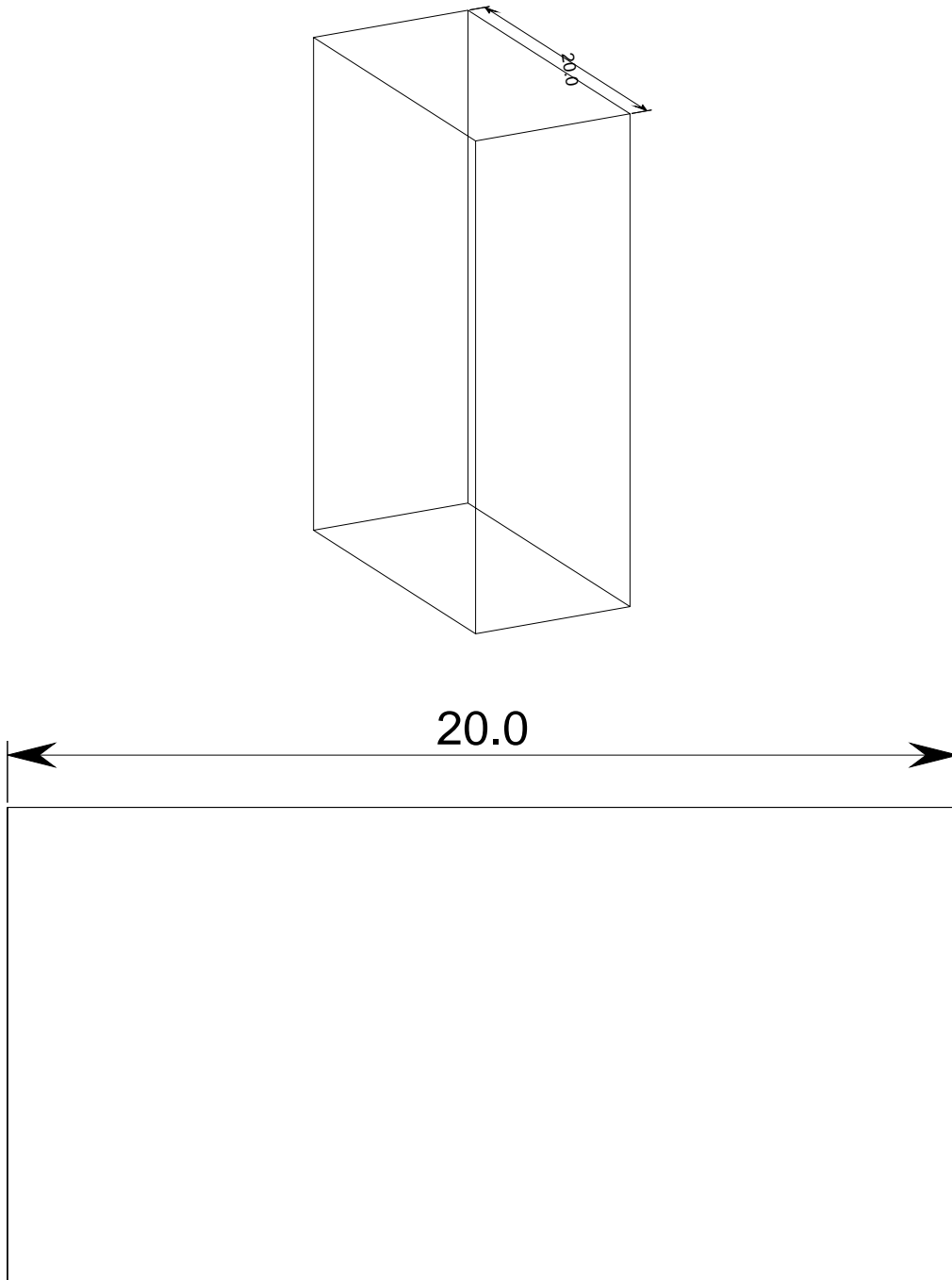
5.4.4 Example of a base-drawing with Scale-independent Annotation-object

Note that in the previous example, the character size on the dimension changes from view to view, because the view-scale is different in each view. The following example specifies the dimension as an annotation-object defined in drawing space, so that it will maintain a constant character size.

```
(in-package :gdl-user)
(define-object box-with-annotated-drawing (base-object)

  :objects

  ((drawing :type 'box-annotated-drawing
            :objects (list (the box)))

   (box :type 'box
        :length 10 :width 20 :height 30)))

(define-object box-annotated-drawing (base-drawing)

  :input-slots (objects (character-size 15)
                (witness-line-gap 10)
                (witness-line-length 15)
                (witness-line-ext 5))

  :objects

  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :length (half (the length))
              :center (translate (the center)
                                 :rear (half (the-child length)))
              :objects (the objects)
              :annotation-objects (list (the main-length-dim)))

   (main-length-dim
    :type 'vertical-dimension
    :pass-down (character-size witness-line-gap witness-line-length
                               witness-line-ext)
    :start-point (the main-view
                      (view-point (the box (vertex :rear :top :right))))
    :end-point (the main-view
                    (view-point (the box (vertex :rear :bottom :right))
                                ))
    :dim-value (3d-distance (the box (vertex :rear :top :right))
                            (the box (vertex :rear :bottom :right)))
    :text-above-leader? nil)

   (top-view :type 'base-view
             :projection-vector (getf *standard-views* :front)
             :length (half (the length))
             :center (translate (the center)
                                :front (half (the-child length)))
             :objects (the objects)
             :annotation-objects (list (the top-length-dim)))
```

```
(top-length-dim
 :type 'vertical-dimension
 :pass-down (character-size witness-line-gap witness-line-length
 :end-point (the top-view
             (view-point (the box (vertex :rear :bottom :right))
                        )))))
```

Bellow is an illustration of this example:
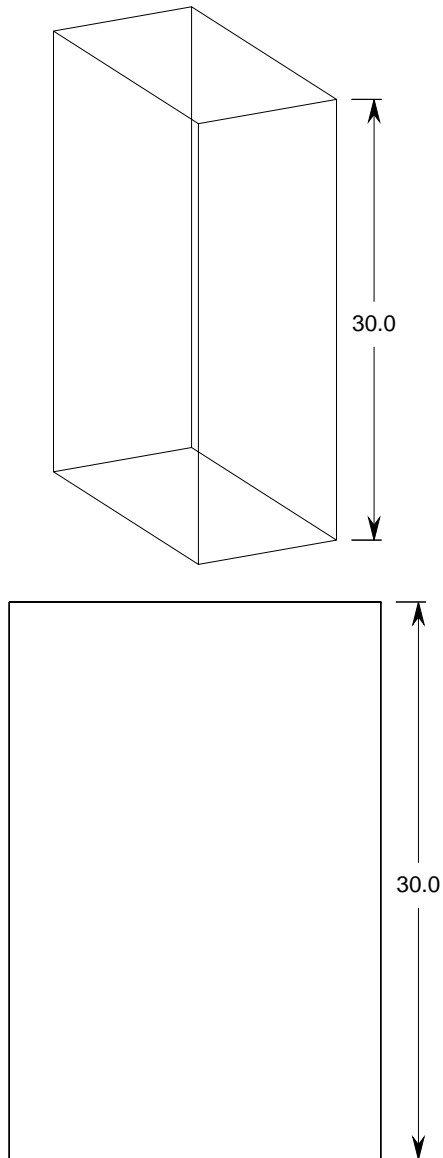
You can output this drawing as a PDF file as follows:

```
(with-format (pdf "/tmp/box-annotated-drawing.pdf")
  (write-the-object (make-object 'box-with-annotated-drawing)
                    drawing  cad-output))
```

and as DXF with:

```
(with-format (dxf "/tmp/box-annotated-drawing.dxf")
  (write-the-object (make-object 'box-with-annotated-drawing)
                    drawing  cad-output))
```

Also you can probe it in Tasty by instantiating box-with-annotated-drawing in a Tasty session, and invoking the Add Node (AN) action on the drawing child object. Be sure to set the Tasty view to top.

5.4.5 Example of a Base-drawing with Immune Annotation-object

The following is not necessary to understand, but might come in useful occasionally:

Sometimes you may want to specify a dimension in model coordinates, but have its character-size, witness-line-length, etc. be predictable in terms of drawing space, regardless of the view-scale. This can be done by defining the character-size, witness-line-length, etc, as a factor of the view's view-scale, but in order to do this, the dimension object must be included in the views list of immune-objects. Otherwise, a circular reference will result, as the base-view tries to use the dimension in order to compute the scale, but the dimension tries to use the scale in order to compute its sizing.

Here is an example:

```
(in-package :gdl-user)

(define-object box-with-immune-dimension (base-object)

  :objects
  ((drawing :type 'immune-dimension-drawing
            :objects (list (the box)))

   (box :type 'box
        :length 10
        :width 20
        :height 30)))
```

```
(in-package :gdl-user)

(define-object immune-dimension-drawing (base-drawing)

  :input-slots ((objects) (character-size 20) (witness-line-gap 10)
                (witness-line-length 15) (witness-line-ext 5))

  :objects
  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :objects (append (the objects) (list (the length-dim)))
              :immune-objects (list (the length-dim)))

   (length-dim :type 'horizontal-dimension
               :character-size (/ (the character-size)
                                  (the main-view view-scale))
               :witness-line-gap (/ (the witness-line-gap)
                                    (the main-view view-scale))
               :witness-line-length (/ (the witness-line-length)
                                       (the main-view view-scale))
               :witness-line-ext (/ (the witness-line-ext)
                                    (the main-view view-scale))
               :start-point (the box (vertex :rear :top :left))
               :end-point (the box (vertex :rear :top :right))

               )))
```

Set self to the box-with-immune-dimension and you will be able to output this drawing as a PDF file as follows:

```
(with-format (pdf "/tmp/immune-dimension.pdf")
  (write-the-object (make-object 'box-with-immune-dimension)
                    drawing  cad-output))
```

and as DXF with:

```
(with-format (dxf "/tmp/immune-dimension.dxf")
  (write-the-object (make-object 'box-with-immune-dimension)
                    drawing  cad-output))
```

and you can probe it in Tasty by instantiating box-with-immune-dimension in a Tasty session, and invoking the Add Node (AN) action on the drawing child object. Be sure to set the Tasty view to top.

5.4.6 Example of a VRML Output Format

In this example is illustrated the use of a dedicated computed-slot employed to output a VRML file which contains the geometry of a simple vase.

```
(in-package :gdl-user)

(define-object vrml-example (base-object)

  :computed-slots

  ((points-data '((1.5 0.0 0.0) (2.5 0.0 0.0)
                  (3.0 0.0 0.0) (4.0 0.0 0.0)
                  (5.0 0.5 0.0) (7.0 3.0 0.0)
                  (5.0 6.5 0.0) (3.5 8.5 0.0)
                  (4.5 10.0 0.0) (6.0 10.0 0.0)))

   (control-points (mapcar #'(lambda(list) (apply-make-point list))
                           (the points-data))))
  :objects

  ((conture :type 'b-spline-curve
            :hidden? t
            :control-points (the control-points))

   (vase :type 'revolved-surfaces
         :display-controls (list :color :periwinkle )
         :axis-point (make-point 1.5 0 0)
         :axis-vector (make-vector 0 1 0)
         :curves (list (the conture)))))
```

You can output this 3D model as a VRML file as follows:

- by setting self the vrml-example and asking at the prompt level for **(the vase-vrml).** or
- as presented before with:

```
(with-format (vrml "d:/vase.wrl")
  (write-the-object (make-object 'vrml-example)
                    cad-output-tree))
```

It is also possible to export the 3D geometry in a VRML pure BSpline format which is supported by BSContact and some other VRML viewers. To achieve this, you have to set self on the vase **surface 0** and evaluate the following expression at prompt level:

```
(with-format (vrml "d:/vase.wrl" :use-bsplines? t)
  (format t "~a" (write-the cad-output)))
```

Note: The VRML pure BSpline format is experimental at the moment.

Bellow is an illustration of the model as solid, tessellated and iso-wireframe:
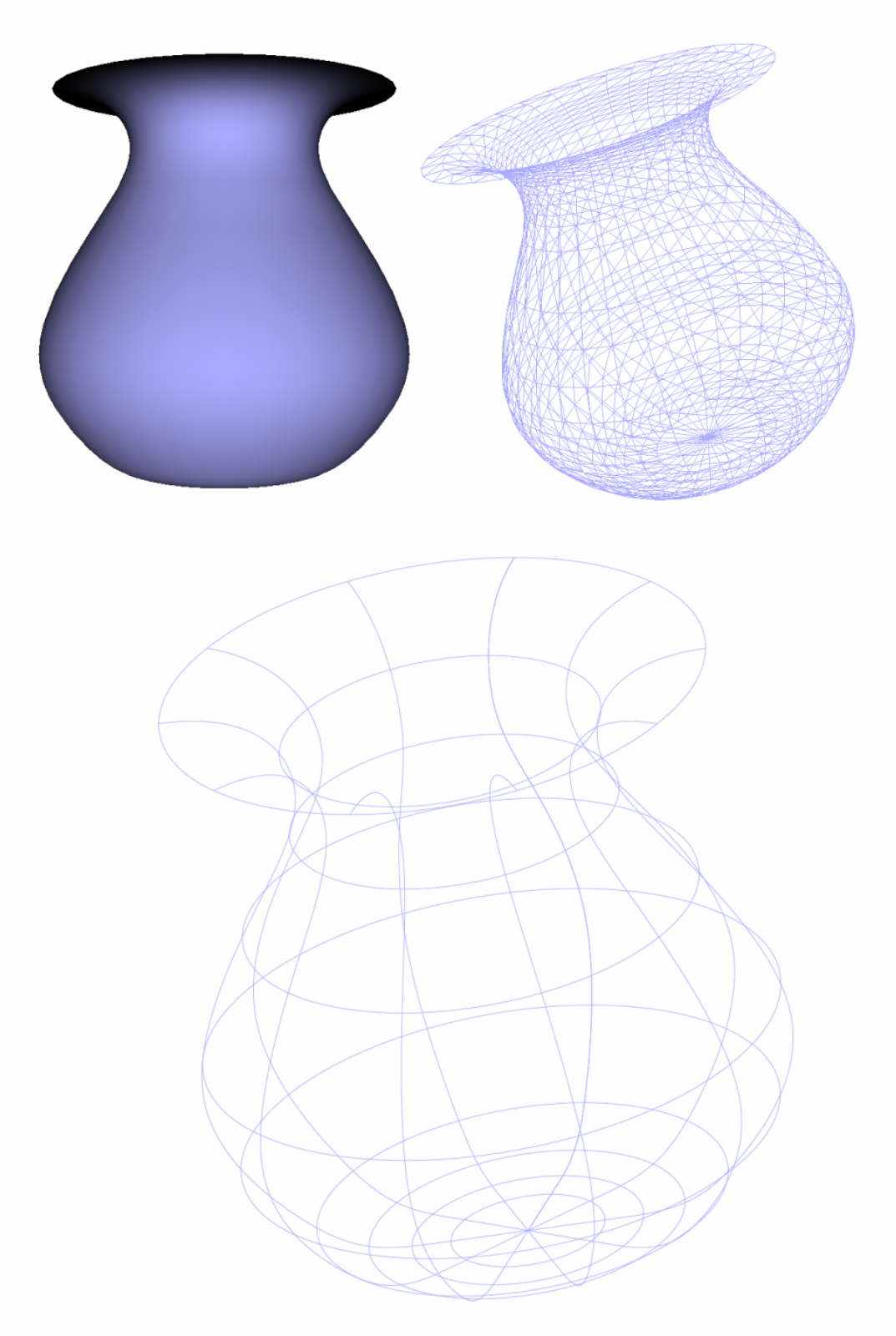
5.4.7 Example of a IGES Output Format

In order to use this example, first you have to compile the example presented in Section 5.4.6 [Example of a VRML Output Format], page 58).

```
(in-package :gdl-user)

(define-object iges-example (base-object)

  :computed-slots
  ((points (list
             (list (make-point 3.5 1 0)(make-point 9 6 0.2)(make-point 6 10 0.2))
             (list (make-point 4.5 1 0)(make-point 9 6 -0.2)(make-point 6 10 -0.2)))))
  :objects

  ((vase :type 'vrml-example
         :hidden? t)

   (handle-curve :type 'fitted-curve
                 :hidden? t
                 :sequence (:size 2)
                 :points (nth (the-child index)(the points)))

   (handle :type 'ruled-surface
           :hidden? t
           :curve-1 (first (list-elements (the handle-curve)))
           :curve-2 (second (list-elements (the handle-curve))))

   (handle-surf :type 'separated-solid
                :hidden? t
                :other-brep (the vase vase (surfaces 0) brep)
                :brep (the handle brep))

   (h-vase :type 'united-solid
           :other-brep (the vase vase (surfaces 0) brep)
           :brep (the handle-surf (breps 1)))))
```

Set self the iges-example and you will be able to output the 3D representation of the h-vase in a IGES file as follows:

```
(with-format (iges "/tmp/iges-example.igs ":units :millimeters)
  (write-the-object (make-object 'iges-example)
                    h-vase cad-output))
```

6 Custom User Interfaces in GenDL

GenDL contains a built-in web server and supports the creation of generative *web-based* user interfaces[1]. Using the same **define-object** syntax which you have already learned, you can define web pages, sections of web pages, and form control elements such as type-in fields, checkboxes, and choice lists. Using this capability does require a basic working knowledge of the HTML language[2].

Any web extensions such as custom JavaScript and JavaScript libraries can also be used, as with any standard web application.

With the primitive objects and functions in its "GWL" package, GenDL supports both the traditional "Web 1.0" interfaces (with fillout forms, page submittal, and complete page refresh) as well as so-called "Web 2.0" interaction with AJAX.

6.1 Package and Environment for Web Development

Similarly to **gdl:define-package**, you can use **gwl:define-package** in order to create a working package which has access to the symbols you will need for building a web application (in addition to all the other GenDL symbols).

The **:gwl-user** package is pre-defined and may be used for practice work. For real projects, you should define your own package using **gwl:define-package**.

The acronym "GWL" stands for Generative Web Language, which is not actually a separate language from GenDL itself, but rather a set of primitive objects and functions available with GenDL for building web applications. The YADD reference documentation for package "Generative Web Language" provides detailed specifications for all the primitive objects and functions.

6.2 Traditional Web Pages and Applications

To make a GenDL object presentable as a web page, the following two steps are needed:

1. Mix **base-html-sheet** into the object definition.
2. define a **:function** called **main-sheet** within the object definition.

The **main-sheet** function should return valid HTML for the page. The easiest way to produce HTML is with the use of an HTML generating library, such as *cl-who*[3] or *htmlGen*[4], both of which are built into GenDL.

For our examples we will use cl-who, which is currently the standard default HTML generating library used internally by Genworks. Here we will make note of the major features of cl-who while introducing the examples; for complete documentation on cl-who, please visit the page at Edi Weitz' website listed in the footnote.

---

[1] GenDL does not contain support for native desktop GUI applications. Although the host Common Lisp environment (e.g. Allegro CL or LispWorks) may contain a GUI builder and Integrated Development Environment, and you are free to use these, Genworks cannot provide support for them.

[2] We will not cover HTML in this manual, but plentiful resources are available online and in print

[3] http://weitz.de/

[4] http://allegroserve.sourceforge.net/aserve-disfdt/doc/htmlgen.html

6.2.1 A Simple Static Page Example

In the following example, the GWL convenience macro **with-cl-who** is used; this sets up a standard default environment for outputting HTML within a GWL application.

```
(in-package :gwl-user)

(define-object president (base-html-sheet)
  :input-slots
  ((name "Carter") (term 1976) (table-border 1))

  :functions
  ((write-html-sheet
    () (with-cl-who (:indent t)
         (:html (:head (:title (fmt "Info on President: ~a"
                                    (the name))))
                (:body ((:table :border (the table-border))
                        (:tr (:th "Name") (:th "Term"))
                        (:tr (:td (str (the name)))
                             (:td (str (the term)))))))))))))
;;
;; Access the above example with
;; http://localhost:9000/make?object=gwl-user::president
;;
```

Produces the following HTML output:

```
<html>
  <head>
    <title>Info on President: Carter
    </title>
  </head>
  <body>
    <table border="1">
      <tr> <th>Name</th>
           <th>Term</th>
      </tr>
      <tr> <td>Carter</td>
           <td>1976</td>
      </tr>
    </table>
  </body></html>
```

Which looks similar to the following in a web browser:
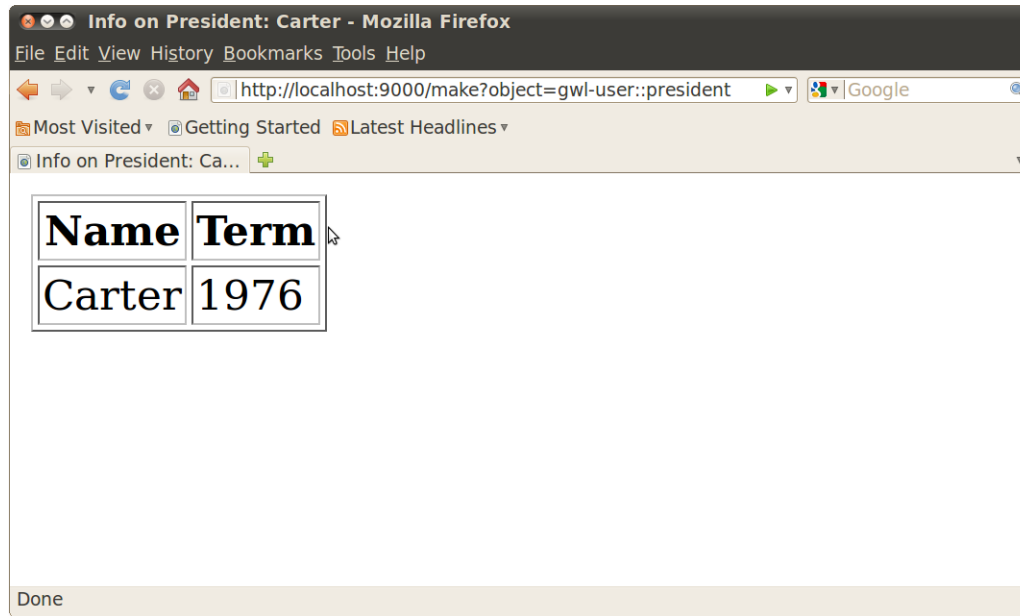
Several important concepts are packed into the above example. Note the following:

- Our convenience macro **with-cl-who** is used to wrap the native **with-html-output** macro which comes with the cl-who library.

- We use the keyword argument **:indent t** in order to pretty-print the generated HTML. This does not affect the browser display but can make the generated HTML easier to read and debug. This option should be left as nil (the default) for production deployments.

- The **fmt** symbol has special meaning within the cl-who environment and works the same as a Common Lisp **format nil** in order to evaluate a format string together with matching arguments, and produce a string at runtime.

- The **str** symbol has special meaning within the cl-who environment and works by evaluating an expression at runtime to return a string or other printable object, which is then included at that point in the HTML output.

- Expressions within the **body** of an HTML tag have to be evaluated, usually by use of the **fmt** or **str** in cl-who. There are three examples of this in the above sample: one **fmt** and two **str**.

- Expressions within a *tag attribute* are always evaluated automatically, and so do **not** require a **str** or other special symbol to force evaluation at runtime. Tag attributes in HTML (or XML) are represented as a plist spliced in after a tag name, wrapped in extra parentheses around the tag name. In the sample above, the **:border (the table-border)** is an example of a tag attribute on the **:table** tag. Notice that the expression **(the table-border)** does not need **str** to be evaluated - it gets evaluated automatically.

- In cl-who, if a tag attribute evaluates to nil, then that tag attribute will be left out of the output completely. For example if **the table-border** evaluates to nil, then the **:table** tag will be outputted without any attributes at all. This is a convenient way to conditionalize tag attributes.

- The URL **http://localhost:9000/make?object=gwl-user::president** is published auto- matically based on the package and name of the object definition. When you visit

this URL, the response is redirected to a unique URL identified by a *session ID*. This ensures that each user to your application site will see their own specific instance of the page object. The session ID is constructed from a combination of the current date and time, along with a pseudo-random number.

6.2.2 A Simple Dynamic Page which Mixes HTML and Common Lisp/GenDL

Within the cl-who environment, it is possible to include any standard Common Lisp structures such as **let**, **dolist**, **dotimes**, etc, which accept a *body* of code. The requirement is that any internal code body must be wrapped in a list beginning with the special symbol **htm**, which has meaning to cl-who.

The following example uses this technique to output an HTML table row for each "row" of data in a list of lists:

```
(in-package :gwl-user)

(define-object presidents (base-html-sheet)
  :input-slots
  ((presidents (list (list :name "Ford"
                           :term 1974)
                     (list :name "Carter"
                           :term 1976)
                     (list :name "Clinton"
                           :term 1992)
                     (list :name "Bush"
                           :term 2000)
                     (list :name "Obama"
                           :term 2008)))

   (table-border 1))

  :functions
  ((write-html-sheet
    ()
    (with-cl-who (:indent t)
      (let ((title (format nil "Info on ~a Presidents:"
                           (length (the presidents)))))
        (htm
         (:html
          (:head (:title (str title)))
          (:body
           (:p (:c (:h3 (str title))))
           ((:table :border (the table-border))
            (:tr (:th "Name") (:th "Term"))
            (dolist (president (the presidents))
              (htm
               (:tr (:td (str (getf president :name)))
                    (:td (str (getf president :term)))))))))))))))
;;
;; Access the above example with
;; http://localhost:9000/make?object=gwl-user::presidents
;;
```

The output looks similar to the following in a web browser:
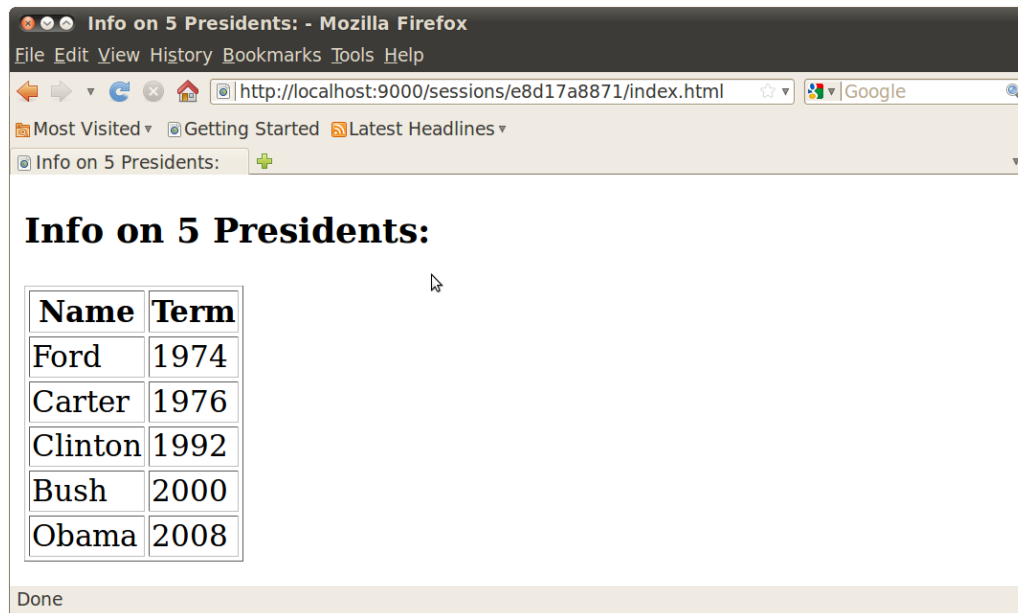
Note the following from this example:

- **title** is a **let** variable, so we use **(str title)** to evaluate it as a string. We do not use **(str (the title))** because **title** is a local variable and not a message (i.e. slot) in the object.

- Inside the **dolist**, we "drop back into" HTML mode using the **htm** symbol.

6.2.3 Linking to Multiple Pages

The base-html-sheet mixin provides a **self-link** message for the purpose of generating a hyperlink to that page. Typically you will have a "parent" page object which links to its "child" pages, but GenDL pages can link to other pages anywhere in the GenDL tree[5].

In the example below, we provide links from a parent page into a child page with detailed information on each president:

---

[5] In order for dependency-tracking to work properly, the pages must all belong to the same tree, i.e. they must share a common root object

```
(in-package :gwl-user)

(define-object presidents-with-pages (base-html-sheet)
  :input-slots
  ((presidents (list (list :name "Ford" :term 1974)
                     (list :name "Carter" :term 1976)
                     (list :name "Clinton" :term 1992)
                     (list :name "Bush" :term 2000)
                     (list :name "Obama" :term 2008)))


   (table-border 1))


  :objects
  ((president-pages :type 'president-page
                    :sequence (:size (length (the presidents)))
                    :name (getf (nth (the-child index) (the presidents))
                                :name)
                    :term (getf (nth (the-child index) (the presidents))
                                :term)))


  :functions
  ((write-html-sheet
    ()
    (with-cl-who (:indent t)
      (let ((title (format nil "Info on ~a Presidents:"
                           (length (the presidents)))))
        (htm
         (:html
          (:head (:title (str title)))
          (:body
           (:p (:c (:h3 (str title))))
           (:ol
            (dolist (page (list-elements (the president-pages)))
              (htm
               (:li
                (the-object
                 page
                 (write-self-link :display-string
                                  (the-object page name)))))))))))))))

;;
;; Access the above example with
;; http://localhost:9000/make?object=gwl-user::presidents-with-pages
;;
```

```
(in-package :gwl-user)

(define-object president-page (base-html-sheet)
  :input-slots
  (name term)

  :functions
  ((write-html-sheet
    ()
    (with-cl-who ()
      (let ((title (format nil "Term for President ~a:"
                           (the name))))
        (htm
         (:html
          (:head (:title (str title)))
          (:body
           (the (write-back-link :display-string "&lt;Back"))
           (:p (:c (:h3 (str title))))
           (:p (str (the term)))))))))))
```

The output looks similar to the following in a web browser:

Note the following from this example:

- The **write-self-link** message is a function which can take a keyword argument of **:display-string**. This string is used for the actual hyperlink text.

- There is a **write-back-link** message which similarly can take a keyword argument of **:display-string**. This generates a link back to (the return-object) which, by default in base-html-sheet, is (the parent).

6.2.4 Form Controls and Fillout-Forms

6.2.4.1 Form Controls

GenDL provides a set of primitives useful for generating the standard HTML form-controls[6] such as text, checkbox, radio, submit, menu, etc. These should be instantiated as child objects in the page, then included in the HTML for the page using **str** within an HTML **form** tag (see next section).

6.2.4.2 Fillout Forms

A traditional web application must enclose form controls inside a **form** tag and specify an **action** (a web URL) to receive and respond to the form submission. The response will cause the entire page to refresh with a new page. In GenDL, such a form can be generated by wrapping the layout of the form controls within the **with-html-form** macro.

Here is an example which allows the user to enter a year, and the application will respond with the revenue amount for that year. Additional form controls are also provided to adjust the table border and cell padding.

This example when instantiated in a web browser, might look like this:

---

[6] http://www.w3.org/TR/html401/interact/forms.html

```
(in-package :gwl-user)

(define-object revenue-lookup-old-school (base-ajax-sheet)

  :input-slots

  ((revenue-data '(2003 25000
                   2004 34000
                   2005 21000
                   2006 37000
                   2007 48000
                   2008 54000
                   2009 78000)))

  :functions

  ((write-html-sheet
    ()
    (with-cl-who ()
      (when *developing?* (str (the development-links)))
      (with-html-form (:cl-who? t)
        (:p (str (the table-border html-string)))
        (:p (str (the cell-padding html-string)))
        (:p (str (the selected-year html-string)))
        (:p ((:input :type :submit :value " OK "))))
      (:p ((:table :border (the table-border value)
                   :cellpadding (the cell-padding value))
           (:tr (:th (fmt "Revenue for Year ~a:"
                          (the selected-year value)))
                (:td (str (getf (the revenue-data)
                                (the selected-year value)))))))))))

  :objects

  ((table-border :type 'menu-form-control
                 :size 1 :choice-list '(0 1)
                 :default 0)

   (cell-padding :type 'menu-form-control
                 :size 1 :choice-list '(0 3 6 9 12)
                 :default 0)

   (selected-year :type 'menu-form-control
                  :size 1 :choice-list (plist-keys (the revenue-data))
                  :default (first (the-child choice-list)))))


(publish-gwl-app "/revenue-lookup-old-school"
                 "gwl-user::revenue-lookup-old-school")



;;
;; Access the above example with
;; http://localhost:9000/make?object=gwl-user::revenue-lookup-old-school
;;
```

6.3 Partial Page Updates with gdlAjax

AJAX stands for Asynchronous JavaScript and XML[7], and allows for more interactive web applications which respond to user events by updating only part of the web page. The "Asynchronous" in Ajax refers to the ability for a web page to continue interacting while one part of the page is being updated by a server request. Requests need not be Asynchronous, they can also be Synchronous ("SJAX"), which would cause the web browser to block execution of any other tasks while the request is being carried out. The "XML" refers to the format of the data that is typically returned from an AJAX request.

GenDL contains a simple framework refered to as *gdlAjax* which supports a uniquely convenient and generative approach to AJAX (and SJAX). With gdlAjax, you use standard GenDL object definitions and child objects in order to model the web page and the sections of the page, and the dependency tracking engine which is built in to GenDL automatically keeps track of which sections of the page need to be updated after a request.

Moreover, the state of the internal GenDL model which represents the page and the page sections is kept identical to the displayed state of the page. This means that if the user hits the "Refresh" button in the browser, the state of the page will remain unchanged. This is not true of some other Ajax frameworks.

6.3.1 Steps to create a gdlAjax application

First, it is important to understand that the fundamentals from the previous section on Standard Web Applications still apply for gdlAjax applications — that is, HTML generation, page linking, etc. These techniques will all still work in a gdlAjax application.

To produce a gdlAjax application involves three main differences from a standard web application:

1. You mix in **base-ajax-sheet** instead of **base-html-sheet**. **base-ajax-sheet** mixes in **base-html-sheet**, so it will still provide all the functionality of that mixin. In fact, you can use **base-ajax-sheet** in standard web applications and you won't notice any difference if you do everything else the same.

2. Instead of a **write-html-sheet** message, you specify a **main-sheet-body** message. The **main-sheet** can be a computed-slot or function, and unlike the **write-html-sheet** message, it should simply return a string, not send output to a stream. Also, it only fills in the body of the page — everything between the <body> and </body> tags. The head tag of the page is filled in automatically and can be customized in various ways.

3. Any sections of the page which you want to be able to change state in response to an Ajax call must be made into separate page sections, or "sheet sections," and the HTML for their **main-div** included in the main page's **main-sheet-body** by use of cl-who's **str** directive.

---

[7] http://en.wikipedia.org/wiki/Ajax_(programming)

```
(in-package :gwl-user)

(define-object revenue-lookup (base-ajax-sheet)

  :input-slots

  ((revenue-data '(2003 25000
                   2004 34000
                   2005 21000
                   2006 37000
                   2007 48000
                   2008 54000
                   2009 78000)))

  :computed-slots

  ((main-sheet-body
    (with-cl-who-string ()
      (str (the main-section main-div)))))

  :objects

  ((table-border :type 'menu-form-control
                 :size 1
                 :choice-list '(0 1)
                 :default 0
                 :ajax-submit-on-change? t)

   (cell-padding :type 'menu-form-control
                 :size 1
                 :choice-list '(0 3 6 9 12)
                 :default 0
                 :ajax-submit-on-change? t)

   (selected-year :type 'menu-form-control
                  :size 1
                  :choice-list (plist-keys (the revenue-data))
                  :default (first (the-child choice-list))
                  :ajax-submit-on-change? t)

   (main-section
    :type 'sheet-section
    :inner-html (with-cl-who-string ()
                  (:p (str (the development-links)))
                  (:p (str (the table-border html-string)))
                  (:p (str (the cell-padding html-string)))
                  (:p (str (the selected-year html-string)))
                  (:p ((:table :border (the table-border value)
                               :cellpadding (the cell-padding value))
                       (:tr (:th (fmt "Revenue for Year ~a:"
                                      (the selected-year value)))
                            (:td (str (getf (the revenue-data)
                                            (the selected-year value)))))))))))

(publish-gwl-app "/revenue-lookup"
                 "gwl-user::revenue-lookup")
```

Note the following from this example:

- We mix in **base-ajax-sheet** and specify a **main-sheet-body** slot, which uses **with-cl-who-string** to compute a string of HTML. This approach is also easier to debug, since the **main-sheet-body** string can be evaluated in the tasty inspector or at the command-line.

- We use **str** to include the string for the main page section (called **main-section** in this example) into the **main-sheet-body**.

- In the **main-section**, we also use **str** to include the html-string for each of three form-controls. We have provided a form control for the table border, the table padding, and the revenue year to look up.

- The only page section in this example is **(the main-section)**. This is defined as a child object, and has its **main-view** computed in the parent and passed in as an input. The **sheet-section** will automatically compute a **main-div** message based on the **main-view** that we are passing in. The **main-div** is simply the **main-view**, wrapped with an HTML DIV ("division") tag which contains a unique identifier for this section, derived from the root-path to the GenDL object in the in tree which represents the sheet section.

- We introduce the CL function **publish-gwl-app**, which makes available a simplified URL for visiting an instance of this object in the web browser. In this case, we can access the instance using **http://localhost:9000/revenue-lookup**

6.3.2 Including Graphics

The fundamental mixin or child type to make a graphics viewport is **base-ajax-graphics-sheet**. This object definition takes several optional input-slots, but the most essential is the **:display-list-objects** and the **:display-list-object-roots**. As indicated by their names, you specify a list of nodes to include in the graphics output with the **:display-list-objects**, and a list of nodes whose **leaves** you want to display in the graphics output with the **:display-list-object-roots**. View controls, rendering format, action to take when clicking on objects, etc, can be controlled with other optional input-slots.

The following example contains a simple box with two graphics viewports and ability to modify the length, height, and and with of the box:

```
(in-package :gwl-user)

(define-object box-with-inputs (base-ajax-sheet)

  :computed-slots
  ((use-raphael? t)

   (main-sheet-body
    (with-cl-who-string ()
      (:p (when *developing?* (str (the development-links))))
      (:p (str (the inputs-section main-div)))
      (:table
       (:tr
        (dolist (viewport (list-elements (the viewport-sections)))
          (htm (:td (:td (str (the-object viewport main-div)))))))))))

  :objects
  ((box :type 'box
        :height (the inputs-section box-height value)
        :width (the inputs-section box-width value)
        :length (the inputs-section box-length value))

   (inputs-section :type 'inputs-section)

   (viewport-sections
    :type 'base-ajax-graphics-sheet
    :sequence (:size 2)
    :view-direction-default (ecase (the-child index)
                              (0 :top) (1 :trimetric))
    :image-format-default :raphael
    :display-list-objects (list (the box))
    :length 250 :width 250)))



(define-object inputs-section (sheet-section)

  :computed-slots
  ((inner-html (with-cl-who-string ()
                 (:p (str (the box-length html-string)))
                 (:p (str (the box-width html-string)))
                 (:p (str (the box-height html-string))))))

  :objects
  ((box-length :type 'text-form-control
               :default 25
               :ajax-submit-on-change? t)
   (box-width :type 'text-form-control
              :default 35
              :ajax-submit-on-change? t)
   (box-height :type 'text-form-control
               :default 45
               :ajax-submit-on-change? t)))


(publish-gwl-app "/box-with-inputs"
                 "gwl-user::box-with-inputs")

;;
;; Access the above example with
;; http://localhost:9000/make?object=gwl-user::box-with-inputs
;;
```

This will produce a web browser output similar to the following:
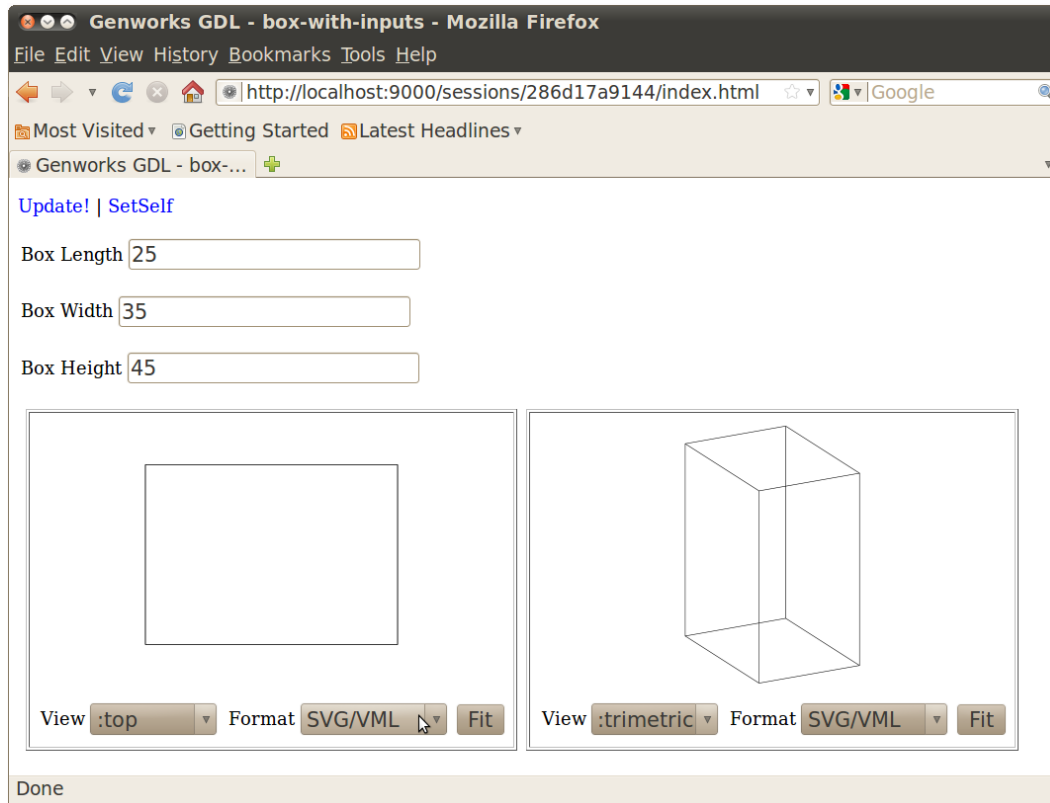
Note the following from this example:

- The **(:use-raphael? t)** enables raphael for SVG or VML output.
- The **:raphael** image-format generates SVG or VML, depending on the browser.
- We conditionally include development-links for full Update and SetSelf! actions.
- We include two viewports in the main-sheet-bodyd, elements from a sequence of size 2.
- In the inputs-section, we use the **html-string** message from each form-control to display the default decoration (prompt, etc).

7 Reference Documentation for GenDL Objects; Operators; and Parameters

GenDL contains auto-generated reference documentation for all its built-in primitive objects and operators. This documentation can be accessed from a live GenDL session through the "Documentation" tab of TaSty.

7.0.1 Gdl

7.0.1.1 Base-rule-object

Description

Encapsulates a basic computation, usually to be displayed to the user. Typically this would be used as a mixin into a more sophisticated rule-object, but the type can be used to detect objects which should be processed as "rules."

Optional-input-slots

rule-description *string*

Short description of the rule (generally one line) Defaults to NIL

rule-description-help *string*

Verbose description of the purpose of the rule

rule-result *string*

The basic return-value, or result, of evaluating the rule

rule-result-help *string*

Verbose description of how the rule result is computed

rule-title *string*

Title to be used with the rule object Defaults to NIL

strings-for-display *string*

Determines the rule's default name in various internal GDL contexts Defaults to the `rule-title`, or "Unnamed Rule" if `rule-title` is NIL

suppress-display? *boolean*

Determines whether the rule is displayed by default in reports etc

violated? *boolean*

Indicates whether this rule violates a standard condition

7.0.1.2 Null-object

Description

A part with no geometric representation and no children. Use this in a conditional `:type` expression if you want to turn off a branch of the tree conditionally.

7.0.1.3 Quantification

Description

A quantification is an aggregate created as a result of specifying `:sequence (:size ...))` or `:sequence (:indices ...))` in an `:objects` specification. Usually, the elements of a quantified set are referenced by using extra parentheses around the message in the reference chain and using the index number. But the aggregate itself also supports certain messages, documented here. One message, `number-of-elements`, is not listed in the normal messages section because it is internal. It can be used, and returns an integer representing the cardinality of the aggregate.

Computed-slots

first *gdl object*

Returns the first element of the aggregate

last *gdl object*

Returns the last element of the aggregate

7.0.1.4 Vanilla-mixin*

Description

Vanilla-Mixin is automatically inherited by every object created in GDL. It provides basic messages which are common to all GDL objects defined with the define-object macro, unless `:no-vanilla-mixin t` is specifqqied at the toplevel of the define-object form.

Optional-input-slots

hidden? *boolean*

Indicates whether the object should effectively be a hidden-object even if specified in :objects Default is nil

root *gdl instance*

The root-level node in this object's "tree" (instance hierarchy)

strings-for-display *string or list of strings*

Determines how the name of objects of this type will be printed in most places This defaults to the name-for-display (generally the part's name as specified in its parent), followed by an index number if the part is an element of a sequence

visible-children *list of gdl instances*

Additional objects to display in Tatu tree Typically this would be a subset of hidden-children Defaults to NIL

Computed-slots

aggregate *gdl instance*

In an element of a sequence, this is the container object which holds all elements

all-mixins *list of symbols*

Lists all the superclasses of the type of this object

children *list of gdl instances*

All objects from the :objects specification, including elements of sequences as flat lists

direct-mixins *list of symbols*

Lists the direct superclasses of the type of this object

first? *boolean*

For elements of sequences, T iff there is no previous element

hidden-children *list of gdl instances*

All objects from the :hidden-objects specification, including elements of sequences as flat lists

index *integer*

Sequential index number for elements of a sequence, NIL for singular objects

last? *boolean*

For elements of sequences, T iff there is no next element

leaf? *boolean*

T if this object has no children, NIL otherwise

leaves *list of gdl objects*

A Collection of the leaf nodes of the given object

name-for-display *keyword symbol*

The part's simple name, derived from its object specification in the parent or from the type name if this is the root instance

next *gdl instance*

For elements of sequences, returns the next part in the sequence

parent *gdl instance*

The parent of this object, or NIL if this is the root object

previous *gdl instance*

For elements of sequences, returns the previous part in the sequence

root-path *list of symbols or of pairs of symbol and integer*

Indicates the path through the instance hierarchy from the root to this object Can be used in conjunction with the `follow-root-path` GDL function to return the actual instance

root-path-local *list of symbols or of pairs of symbol and integer*

Indicates the path through the instance hierarchy from the local root to this object Can be used in conjunction with the `follow-root-path` GDL function to return the actual instance

root? *boolean*

T iff this part has NIL as its parent and therefore is the root node

safe-children *list of gdl instances*

All objects from the :objects specification, including elements of sequences as flat lists Any children which throw errors come back as a plist with error information

safe-hidden-children *list of gdl instances*

All objects from the :hidden-objects specification, including elements of sequences as flat lists Any children which throw errors come back as a plist with error information

type *symbol*

The GDL Type of this object

Functions

documentation *plist*

Returns the `:documentation` plist which has been specified the specific part type of this instance.

follow-root-path *gdl instance*

Using this instance as the root, follow the reference chain represented by the given path.

message-documentation *string*

This is synonymous with `slot-documentation`

message-list *list of keyword symbols*

Returns the messages (slots, objects, and functions) of this object, according to the filtering criteria as specified by the arguments.

Keyword Arguments:

category Keyword, Default Value: all, Either `:all` or the individual category of messages to be returned. This can be one of:

- :computed-slots
- :settable-computed-slots
- :required-input-slots
- :optional-input-slots
- :defaulted-input-slots
- :query-slots
- :functions
- :objects
- :quantified-objects
- :hidden-objects
- :quantified-hidden-objects

.

message-type Keyword Symbol, :local or :global, Default Value: global, Indicates whether to return messages only from the local specific part type, or from all superclasses (mixins) as well.

return-category? Boolean, Default Value: nil, Indicates whether or not the category of each message should be returned before each message in the returned list.

base-part-type Symbol naming a GDL Part Type, Default Value: nil, Indicates a "base" part from which no messages should be returned, nor should messages be returned from superclasses (mixins) of this base part. If NIL (the default), messages are considered from all superclasses.

sort-order Keyword Symbol, Default Value: unsorted, One of: `:unsorted`, `:by-category`, or `:by-name`.

filter Function Object of two arguments or `:normal`, Default Value: normal, If a function object, applies this function to each returned category and message keyword, and filters out all pairs for which the function returns NIL. If `:normal` (the default), then no filtering is done.

mixins *list of symbols*

Returns the names of the immediate superclasses of this object.

Keyword Arguments:

local? Boolean, Default Value: t, Indicates whether to give only direct mixins or all mixins from the entire inheritance hierarchy.

restore-all-defaults! *void*

Restores all settable-slots in this instance to their default values.

restore-slot-default! *nil*

Restores the value of the given slot to its default, thus "undoing" any forcibly set value in the slot. Any dependent slots in the tree will respond accordingly when they are next demanded. Note that the slot must be specified as a keyword symbol (i.e. prepended with a

colon (":")), otherwise it will be evaluated as a variable according to normal Lisp functional evaluation rules.

restore-slot-defaults! *nil*

Restores the value of the given slots to their defaults, thus "undoing" any forcibly set values in the slots. Any dependent slots in the tree will respond accordingly when they are next demanded. Note that the slots must be specified as keyword symbols (i.e. prepended with colons (":")), otherwise they will be evaluated as variables according to normal Lisp functional evaluation rules.

Keyword Arguments:

force? Boolean, Default Value: *force-restore-slot-default?*, Indicates whether the slot values should be unbound, regardless of whether it had actually been bashed previously.

restore-tree! *void*

Restores all settable-slots in this instance, and recursively in all descendant instances, to their default values.

set-slot! *nil*

Forcibly sets the value of the given slot to the given value. The slot must be defined as `:settable` for this to work properly. Any dependent slots in the tree will respond accordingly when they are next demanded. Note that the slot must be specified as a keyword symbol (i.e. prepended with a colon (":")), otherwise it will be evaluated as a variable according to normal Lisp functional evaluation rules.

Keyword Arguments:

remember? Boolean, Default Value: t, Determines whether to save in current version-tree.

set-slots! *nil*

Forcibly sets the value of the given slots to the given values. The slots must be defined as `:settable` for this to work properly. Any dependent slots in the tree will respond accordingly when they are next demanded. Note that the slots must be specified as a keyword symbols (i.e. prepended with a colon (":")), otherwise they will be evaluated as variables according to normal Lisp functional evaluation rules.

slot-documentation *plist of symbols and strings*

Returns the part types and slot documentation which has been specified for the given slot, from most specific to least specific in the CLOS inheritance order. Note that the slot must be specified as a keyword symbol (i.e. prepended with a colon (":")), otherwise it will be evaluated as a variable according to normal Lisp functional evaluation rules.

slot-source *body of gdl code, in list form*

.

slot-status *keyword symbol*

Describes the current status of the requested slot:

1. `:unbound`: it has not yet been demanded (this could mean either it has never been demanded, or something it depends on has been modified since the last time it was demanded and eager setting is not enabled).

2. `:evaluated`: it has been demanded and it is currently bound to the default value based on the code.

3. `:set`: (for :settable slots only, which includes all required :input-slots) it has been modified and is currently bound to the value to which it was explicitly set.

4. `:toplevel`: (for root-level object only) its value was passed into the root-level object as a toplevel input at the time of object instantiation.

update! *void*

Uncaches all cached data in slots and objects througout the instance tree from this node, forcing all code to run again the next time values are demanded. This is useful for updating an existing model or part of an existing model after making changes and recompiling/reloading the code of the underlying definitions. Any set (modified) slot values will, however, be preserved by the update.

write-snapshot *void*

Writes a file containing the toplevel inputs and modified settable-slots starting from the root of the current instance. Typically this file can be read back into the system using the `read-snapshot` function.

Keyword Arguments:

filename String or pathname, Default Value: /tmp/snap.gdl, The target file to be written.

7.0.2 Gwl

7.0.2.1 Application-mixin

Description

This mixin generates a default GWL user interface, similar to `node-mixin`, but you should use `application-mixin` if this is a leaf-level application (i.e. has no children of type `node-mixin` or `application-mixin`

7.0.2.2 Base-ajax-graphics-sheet

Description

This mixes together base-ajax-sheet with base-html-graphics-sheet, and adds html-format output-functions for several of the new formats such as ajax-enabled png/jpeg and Raphael vector graphics.

Optional-input-slots

background-color *array of three numbers between 0 and 1*

RGB Color in decimal format Color to be used for the background of the viewport Defaults to the `:background` from the global `*colors-default*` parameter

display-list-object-roots *list of gdl objects*

The leaves of each of these objects will be included in the geometry display Defaults to nil

display-list-objects *list of gdl objects containing geometry*

These are the actual objects themselves, not nodes which have children or other descendants that you want to display If you want to display the leaves of certain nodes, include the objects for those nodes in the display-list-object-roots, not here Defaults to nil

field-of-view-default *number in angular degrees*

The maximum angle of the view frustrum for perspective views Defaults to 45 which is natural human eye field of view

image-format *keyword symbol*

Determines the default image format Defaults to the currently selected value of the image-format-selector, which itself defaults to :raphael

image-format-default *keyword symbol, one of the keys from (the image-format-plist)*

Default for the image-format-selector Defaults to :png

image-format-plist *plist of keys and strings*

The default formats for graphics display Defaults to:

```
(list :png "PNG image"
      :jpeg "jpeg image"
      :raphael "SVG/VML")
```

immune-objects *list of gdl objects*

These objects are not used in computing the scale or centering for the display list Defaults to nil

include-view-controls? *boolean*

Indicates whether standard view-controls panel should be included with the graphics

projection-vector *3d vector*

This is the normal vector of the view plane onto which to project the 3D objects Defaults to (getf *standard-views* (the view-selector value)), and (the view-selector value) defaults to :top

use-raphael-graf? *boolean*

Include raphael graphing library in the page header? Default nil

use-raphael? *boolean*

Include raphael javascript library in the page header? Default nil

view-direction-default *default view initially in the view-selector which is automatically included in the view-controls*

viewport-border-default *number*

Thickness of default border around graphics viewport Default is 1

Settable-computed-slots

js-to-eval *string of valid javascript*

This Javascript will be send with the Ajax response, and evaluated after the innerHTML for this section has been replaced

Computed-slots

graphics *string of valid html*

This can be used to include the geometry, in the format currently selected by the image-format-selector If the include-view-controls? is non-nil, the view-controls will be appended at the bottom of the graphics inside a table

inner-html *string*

This can be used with (str) [in cl-who] or (:princ) [in htmlGen] to output this section of the page, without the wrapping :div tag [so if you use this, your code would be responsible for wrapping the :div tag with :id (the dom-id)]

raster-graphics *string of valid html*

This can be used to include the PNG or JPG raster-graphics of the geometry

vector-graphics *string of valid html*

This can be used to include the SVG or VML vector-graphics of the geometry

view-controls *string of valid html*

This includes the image-format-selector, the reset-zoom-button, and the view-selector, in a simple table layout You can override this to make the view-controls appear any way you want and include different and/or additional form-controls

web3d-graphics *string of valid html*

This can be used to include the VRML or X3D graphics of the geometry

Hidden-objects

image-format-selector *object of type menu-form-control*

Its value slot can be used to determine the format of image displayed

Functions

write-embedded-x3dom-world *void*

Writes an embedded X3D tag with content for the `view-object` child of this object. The `view-object` child should exist and be of type `web-drawing`.

Examples

FLAG – Fill in!!!

7.0.2.3 Base-ajax-sheet

Description

(Note: this documentation will be moved to the specific docs for the html-format/base-ajax-sheet lens, when we have lens documentation working properly)

Produces a standard main-sheet for html-format which includes the standard GDL Javascript to enable code produced with gdl-ajax-call to work, and optionally to include the standard JQuery library.

If you want to define your own main-sheet, then there is no use for base-ajax-sheet, you can just use base-html-sheet. But then you have to include any needed Javascript yourself, e.g. for gdl-ajax-call support or jQuery.

The html-format lens for base-ajax-sheet also defines a user hook function, main-sheet-body, which produces a "No Body has been defined" message by default, but which you can fill in your own specific lens to do something useful for the body.

Optional-input-slots

body-class *string or nil*

Names the value of class attribute for the body tag Default is nil

doctype-string *string or nil*

Contains the string for the doctype at the top of the document Default is:
""

main-sheet-body *string of html*

The main body of the page This can be specified as input or overridden in subclass, otherwise it defaults to the content produced by the :output-function of the same name in the applicable lens for html-format

respondent *gdl object*

Object to respond to the form submission Defaults to self

title *string*

The title of the web page Defaults to "Genworks GDL -" followed by the strings-for-display

Settable-optional-input-slots

additional-header-content *string of valid html*

Additional tag content to go into the page header, if you use the default main-sheet message and just fill in your own main-sheet-body, as is the intended use of the base-ajax-sheet primitive

additional-header-js-content *valid javascript*

This javascript is added to the head of the page -just before- the body When jquery is loaded (by setting the input-slot (use-jquery t)), the javascript can use the '$([selector])' shortcuts to do the magic The javascript is automagically wrapped in the appropriate html tags to ensure a good execution by the javascript engine When the use-jquery slot is true (t) than the javascript is wrapped in a '$(document) ready' function as a $GoodPractice^{TM}$:example-1 when use-jquery is set to true (t)

```
$('ul#menu') superfish(delay:1,speed:'fast') ;
```

results in the following piece of HTML added just before the body

```
&lt;script type='text/javascript'&gt;
  $(document) ready(function ()
    $('ul#menu') superfish(delay:1,speed:'fast') ;
  );
&lt;/script&gt;
```

:example-2 when use-jquery is set to nil (false, not loaded)

```
function initMenu(
    // do javascript magic here
  );
```

results in the following piece of HTML added just before the body

```
&lt;script type='text/javascript' language='Javascript'&gt;
  function initMenu(
    // do javascript magic here
  );
&lt;/script&gt;
```

ui-specific-layout-js *absolute uri in the browser to application user interface (ui) specific jquery layout javascript*

This is additional JavaScript that needs to be loaded in order to initiate the layout of a user interface GDL uses the jQuery Layout plugin (`http://layout jquery-dev net/documentation html`) to construct a layout The HTML part of the user interface is normally defined in a the main-sheet-body, a lens of the main assembly This input slot defaults to /static/gwl/js/tasty-initlayout-3 js You can check this file as a reference for your own application UI specific jQuery Layout JavaScript

use-jquery? *boolean*

Include jquery javascript libraries in the page header? Default nil

Examples

```
(in-package :gdl-user)

(gwl:define-package :ajax-test (:export #:assembly))

(in-package :ajax-test)

(define-object assembly (base-ajax-sheet)

  :objects
  ((inputs-section :type 'inputs-section)

   (outputs-section :type 'outputs-section
                    :box (the viewport box)
                    :color (the inputs-section color))

   (viewport :type 'viewport
             :box-color (the inputs-section color))))

(define-lens (html-format assembly)()
  :output-functions
  ((main-sheet-body
    ()
    (with-cl-who ()
      (:table
       (:tr
        (:td (str (the inputs-section main-div)))
        (:td (str (the outputs-section main-div)))
        (:td (str (the viewport main-div)))))))))

(define-object inputs-section (sheet-section)

  :computed-slots ((color (the menu-control value)))

  :objects
  ((menu-control :type 'menu-form-control
                 :choice-list (list :red :green :blue)
                 :default :red
                 :onchange (the (gdl-ajax-call
                                 :form-controls (list (the-child)))))

   (little-grid :type 'grid-form-control
                :form-control-types '(text-form-control
                                      text-form-control
                                      button-form-control)
                :form-control-attributes '((:ajax-submit-on-change? t)
                                           (:ajax-submit-on-change? t))
                :form-control-inputs
                (mapcar #'(lambda(row)
                            (list nil nil
                                  (list :onclick
                                        (the (gdl-ajax-call
                                              :function-key :do-something!
                                              :arguments
                                              (list (the-object row index)))))))))
```

```
                              (list-elements (the-child rows)))
                   :default '((:color :number :press-me)
                              (:red 42 "OK")
                              (:blue 50 "OK")))))

   :computed-slots
   ((inner-html (with-cl-who-string ()
                  (str (the little-grid form-control-string))
                  (str (the menu-control html-string)))))

   :functions
   ((do-something! (index)
      (format t "Processing row ~a...~%" index))))



(define-object outputs-section (sheet-section)

  :input-slots (color box)

  :computed-slots
  ((inner-html (with-cl-who-string ()
                 (:p "The box volume is: " (fmt "~a" (the box volume)))
                 (:p "The box color is: "
                     ((:span :style (format nil "color: ~a" (the color)))
                      (str (the color)))))))))



(define-object viewport (base-ajax-graphics-sheet)

  :input-slots (box-color)

  :computed-slots ((length 300)
                   (width 300)
                   (display-list-objects (list (the box)))
                   (projection-vector (getf *standard-views*
                                            (the view-selector value)))
                   (inner-html
                    (with-cl-who-string ()
                      (str (the view-selector html-string))
                      (str (the reset-zoom-button form-control-string))
                      (str (the raster-graphics)))))

  :objects ((box :type 'box
                 :length 20 :width 25 :height 30
                 :display-controls (list :color (the box-color)))))

(publish-gwl-app "/ajax-test" "ajax-test:assembly")
```

7.0.2.4 Base-form-control

Description

This object can be used to represent a single HTML form control. It captures the initial default value, some display information such as the label, and all the standard HTML tag attributes for the tag e.g. INPUT, SELECT, TEXTAREA. GWL will process the data types according to specific rules, and validate the typed value according to other default rules. A custom validation-function can also be provided by user code.

Sequences of these objects (with :size, :indices, :matrix, and :radial) are supported.

This facility and its documentation is expected to undergo significant and frequent upgrades in the remainder of GDL 1573 and upcoming 1575.

Current to-do list:

1. Currently this works with normal HTTP form submission and full page reloading. We intend to make it work with AJAX and surgical page update as well.
2. We intend to provide inputs for all the standard tag attributes for the accompanying LABEL tag for the form control.
3. Additional form control elements to be included, to cover all types of form elements specified in current HTML standard from

   http://www.w3.org/TR/html401/interact/forms.html#h-17.2.1

   - button-form-control: submit buttons, reset buttons, push buttons.
   - checkbox-form-control: checkboxes, radio buttons (multiple of these must be able to have same name)
   - menu-form-control: select, along with optgroup and option.
   - text-form-control: single-line text input (including masked passwords) and multi-line (TEXTAREA) text input.
   - file-form-control: file select for submittal with a form.
   - hidden-form-control: input of type hidden.
   - object-form-control: (not sure how this is supposed to work yet).

Also, we have to study and clarify the issue of under what conditions values can possibly take on nil values, and what constitutes a required field as opposed to a non-validated field, and whether a blank string on a text input should be represented as a nil value or as an empty string.

Note that checkbox-form-control and menu-form-control currently get automatically included in the possible-nils.

Optional-input-slots

accept *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

accesskey *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

align *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

allow-invalid-type? *boolean*

If non-nil, then values which fail the type test will still be allowed to be the value Default is nil

allow-invalid? *boolean*

If non-nil, then values which fail the type or validation test will still be allowed to be the value Default is t

allow-nil? *boolean*

Regardless of :domain, if this is non-nil, nil values will be accepted Defaults to t if (the default) is nil, otherwise defaults to nil

alt *string or nil*

Maps to HTML form control attribute of the same name Default is nil

append-error-string? *boolean*

Determines whether a default error string is appended to string ouput-function for html-format (and therefore html-string computed-slot as well) Defaults to t

default *lisp value of a type compatible with (the domain)*

This is the initial default value for the control This must be specified by user code, or an error will result

disabled? *boolean*

Maps to HTML form control attribute of the same name Default is nil

domain *keyword symbol, one of :number, :keyword, :list-of-strings, :list-of-anything, or :string*

This specifies the expected and acceptable type for the submitted form value If possible, the submitted value will be coerced into the specified type The default is based upon the Lisp type of (the default) provided as input to this object If the default is nil, the domain will default to :string

ismap? *boolean*

Maps to HTML form control attribute of the same name Default is nil

label-position *keyword symbol or nil*

Specifies where the label tag goes, if any Can be :table-td (label goes in a td before the form control), :table-td-append (label goes in a td after the form control), :prepend (label tag wraps around form control and label text comes before form control), :append (label tag wraps around form control and label text comes after form control), :table-with-class (like :table-td, but adds a class "form-control" to the table), or :as-div (puts label and control inside a div of class "form-control")

Default is :table-td

lang *string or nil*

Maps to HTML form control attribute of the same name Default is nil

maxlength *number or nil*

   Maps to HTML form control attribute of the same name Default is nil

nullify-empty-string? *boolean*

   Regardless of :domain, if this is non-nil, empty strings will convert to nil Defaults to (the allow-nil?)

onblur *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onchange *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onclick *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

ondblclick *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onfocus *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onkeydown *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onkeypress *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onkeyup *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onmousedown *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onmousemove *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onmouseout *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onmouseover *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onmouseup *string or nil*

   Maps to HTML form control attribute of the same name Default is nil

onselect *string or nil*

Maps to HTML form control attribute of the same name Default is nil

preset? *boolean*

This switch determines whether this form-control should be preset before the final setting, in order to allow any interdependencies to be detected for validation or detecting changed values Default is nil

prompt *string*

The prompt used in the label

readonly? *boolean*

Maps to HTML form control attribute of the same name Default is nil

size *number or nil*

Maps to HTML form control attribute of the same name Default is nil

src *string or nil*

Maps to HTML form control attribute of the same name Default is nil

style *string or nil*

Maps to HTML form control attribute of the same name Default is nil

tabindex *integer or nil*

Maps to HTML form control attribute of the same name Default is nil

title *string or nil*

Maps to HTML form control attribute of the same name Default is nil

usemap *string or nil*

Maps to HTML form control attribute of the same name Default is nil

validation-function *function of one argument*

The argument will be the submitted form value converted to the proper type The return value from this function can be nil, any non-nil value, or a plist with keys :validated-value and :error The following behavior applies:

- If the function returns nil, error is set to :unspecified-validation-fail
- If the function returns a plist with keys :validated-value and :error, and if :error is non-nil, it means the value is not acceptable, the form-controls error message is set to this error (usually a keyword symbol), and the error string will be appended to the html-string by default
- If the function returns any other value, then the properly typed submitted form value is considered valid and is used

In the case of an error, the form-control's failed-value message is set to the properly typed submitted form value If allow-invalid? is non-nil, then the form-control's value message is also set to this value (i e the invalid value is still accepted, even though a non-nil error is present)

Default is (list :validated-value value :error nil)

Settable-computed-slots

error *string or error object*

This will be set to a validation error if any, and cleared when the error is gone

failed-value *lisp value*

The value which was attempted to be set but failed validation

value *lisp value*

The current value of this form control

Functions

restore-defaults! *void*

Restores the default for the value, the failed-value, and the error.

Examples

```
(in-package :gwl-user)

(define-object test-form (base-html-sheet)

  :objects
  ((username :type 'text-form-control
             :size 35
             :maxlength 30
             :allow-nil? t
             :default "Ron Paul")

   (age :type 'text-form-control
        :size 5
        :validation-function #'(lambda(input) (or (null input) (> 80 input 70)))
        :domain :number
        ;;:default 72
        :default nil )

   (bio :type 'text-form-control
        :rows 8
        :size 120
        :default "
Congressman Ron Paul is the leading advocate for freedom in our nation's capital.
As a member of the U.S. House of Representatives, Dr. Paul tirelessly works for
limited constitutional government, low taxes, free markets, and a return to sound
monetary policies. He is known among his congressional colleagues and his constituents
for his consistent voting record. Dr. Paul never votes for legislation unless the
proposed measure is expressly authorized by the Constitution. In the words of former
Treasury Secretary William Simon, Dr. Paul is the one exception to the Gang of 535 on
Capitol Hill.")

   (issues :type 'menu-form-control
```

```
                       :choice-list (list "Taxes" "Health Care" "Foreign Policy")
                       :default "Taxes"
                       :multiple? t)

         (color :type 'menu-form-control
                :size 7
                :choice-plist (list :red "red"
                                    :green "green"
                                    :blue "blue"
                                    :magenta "magenta"
                                    :cyan "cyan"
                                    :yellow "yellow"
                                    :orange "orange")
                :validation-function #'(lambda(color)
                                         (if (intersection (ensure-list color)
                                                           (list :yellow :magenta))
                                             (list :error :disallowed-color-choice)
                                           t))
                ;;:append-error-string? nil
                :multiple? t
                :default :red
                ;;:onchange "alert('hey now');"
                )

         (early-riser? :type 'checkbox-form-control
                       :default nil)

         (favorite-links :type 'text-form-control
                         :sequence (:size 3)
                         :size 70
                         :default "http://")))

    (define-lens (html-format test-form)()
      :output-functions
      ((main-sheet
        ()
        (with-html-output (*html-stream* nil :indent t)
          (:html (:head (:title "Test Form"))
                 (:body (:h2 (:center "Test Form"))
                        (the write-development-links)
                        (with-html-form (:cl-who? t)
                          (:p (str (the username html-string)))
                          (:p "(internal value is: " (fmt "~s" (the username value)) ")")
                          (:p (str (the age html-string)))
                          (:p "(internal value is: " (fmt "~s" (the age value)) ")")
                          (:p (str (the bio html-string)))
                          (:p (:table
                               (:tr (:td (str (the issues html-string))))
                               (:tr (:td (str (the color html-string))))))
                          (:p (str (the early-riser? html-string)))

                          (dolist (link (list-elements (the favorite-links)))
                            (htm (str (the-object link html-string))))

                          (:p ((:input :type :submit :value " OK "))))))))))

    (publish :path "/fe"
             :function #'(lambda(req ent)
```

```
(gwl-make-object req ent "gwl-user::test-form")))
```

7.0.2.5 Base-html-graphics-sheet

Description

This mixin allows a part to be displayed as a web page in GWL, and to contain one graphics area. It requires the geom-base module to be loaded. This will probably be extended to allow more than one graphics area. This mixin inherits from base-html-sheet, so just like with `base-html-sheet` you can prepare the output with the `write-html-sheet` function in a the object which mixes this in, or in a `main-sheet` output-function in an html-format view of the object.

Optional-input-slots

standard-views *plist of keywords and 3d vectors*

Indicates the views to show in the graphics controls

use-bsplines? *boolean*

Determines whether to use native bspline data in the vrml

Settable-optional-input-slots

digitation-mode *keyword symbol, one of <tt>:zoom-and-center</tt>, <tt>:report-point</tt>, or <tt>:measure-distance</tt>*

- If `:zoom-and-center`, sets the user-center and user-scale accordingly when graphics area is clicked
- If `:report-point`, the slot `digitized-point` is set with the x y value
- If `measure-distance`, the slot `:digitized-distance` is set with the resultant distance

  Default is `:zoom-and-center`

image-format *keyword symbol*

Determines the default image format Defaults to :png

view *keyword symbol*

Determines the default view from the `standard-views` Defaults to :trimetric

zoom-factor *number*

The factor used for zooming in or out

zoom-mode *keyword symbol, one of :in, :out, or :none, or nil*

If :in, then clicks in the graphics area will increase the zoom factor by (the zoom-factor) If :out, then clicks will decrease the factor by that amount If :none or nil, then clicks will have no effect

Functions

background-color *keyword symbol, string, list, or vector*

Default background for the graphics viewport. Can be specified as a name (keyword or string) in *color-table*, an html-style hex string (starting with #), or a decimal RGB triplet in a list or vector. The default comes from the :background entry in `*colors-default*`.

foreground-color *keyword symbol, string, list, or vector*

Default foreground for the graphics viewport. Can be specified as a name (keyword or string) in *color-table*, an html-style hex string (starting with #), or a decimal RGB triplet in a list or vector. The default comes from the :foreground entry in `*colors-default*`.

report-point *void*

Process the points selected by digitizing in the graphics. You can override this function to do your own processing. By default, it prints the information to the console.

write-embedded-vrml-world *void*

Writes an EMBED tag and publishes a VRML world for the `view-object` child of this object. The `view-object` child should exist and be of type `web-drawing`.

write-embedded-x3d-world *void*

Writes an OBJECT tag and publishes an X3D world for the `view-object` child of this object. The `view-object` child should exist and be of type `web-drawing`.

write-geometry *void*

Writes an image tag and publishes an image for the `view-object` child of this object. The `view-object` child should exist and be of type `web-drawing`. For objects of type `gwl:application-mixin` or `gwl:node-mixin`, this is done automatically. For the time being, we recommend that you use `gwl:application-mixin` or `gwl:node-mixin` if you want to display geometric parts in a GWL application.

  Keyword Arguments:

   include-view-controls? Boolean, Default Value: t, Determines whether the standard view controls are displayed below the image.

7.0.2.6 Base-html-sheet

Description

This mixin allows a part to be displayed as a web page in GWL. The main output can be specified either in a `write-html-sheet` function in the object which mixes this in, or in a `main-sheet` output-function in an html-format view of the object.

Optional-input-slots

check-sanity? *boolean*

Determines whether a a sanity check is done (with the `check-sanity` function) before presenting the response page if this page is a respondent Default is NIL

return-object *gdl object*

Default object to which control will return with the write-back-link method

target *string*

Name of a browser frame or window to display this page Default of NIL indicates to use the same window

transitory-slots *list of keyword symbols*

Messages corresponding to form fields which should not be retained against Updates to the model (e g calls to the update! function or hitting the Update button or link in the browser in development mode) Defaults to NIL (the empty list)

Settable-computed-slots

query-plist *plist*

Contains submitted form field names and values for which no corresponding settable computed-slots exist Where corresponding settable computed-slots exist, their values are set from the submitted form fields automatically

Computed-slots

header-plist *plist*

Extra http headers to be published with the URI for this page

url *string*

The web address in the current session which points at this page Published on demand

Functions

after-present! *void*

This is an empty function by default, but can be overridden in the respondent of a form, to do some processing after the respondent's `write-html-sheet` function runs to present the object.

after-set! *void*

This is an empty function by default, but can be overridden in the requestor of a form, to do some processing after the requestor's form values are set into the specified bashee.

before-present! *void*

This is an empty function by default, but can be overridden in the respondent of a form, to do some processing before the respondent's `write-html-sheet` function runs to present the object. This can be useful especially for objects which are subclasses of higher-level mixins such as `application-mixin` and `node-mixin`, where you do not have direct access to the `write-html-sheet` function and typically only define the `model-inputs` function.

It is not always reliable to do processing in the `model-inputs` function, since some slots which depend on your intended modifications may already have been evaluated by the time the `model-inputs` function runs.

before-response! *void*

This is an empty function by default, but can be overridden in a user specialization of base-html-sheet, to do some processing before the header-plist is evaluated and before the HTTP response is actually initiated.

before-set! *void*

This is an empty function by default, but can be overridden in the requestor of a form, to do some processing before the requestor's form values are set into the specified bashee.

check-sanity *nil or error object*

This function checks the "sanity" of this object. By default, it checks that following the object's root-path from the root resolves to this object. If the act of following the root-path throws an error, this error will be returned. Otherwise, if the result of following the root-path does not match the identity of this object, an error is thrown indicating this. Otherwise, NIL is returned and no error is thrown. You can override this function to do what you wish. It should return NIL if the object is found to be "sane" and an throw an error otherwise. If check-sanity? is set to T in this object, this function will be invoked automatically within an ignore-errors by the function handling the GWL "/answer" form action URI when this object is a respondent, before the main-sheet is presented.

process-cookies! *void*

This is an empty function by default, but can be overridden in a user specialization of base-html-sheet, to do some processing before the header-plist is evaluated and before the HTTP response is actually initiated, but after the cookies-received have been set.

restore-form-controls! *void*

Calls restore-defaults! on all the form-controls in this sheet.

sanity-error *void*

Emits a page explaining the sanity error. This will be invoked instead of the write-main-sheet if check-sanity? is set to T and the check-sanity throws an error. You may override this function to do what you wish. By default a minimal error message is displayed and a link to the root object is presented.

select-choices *void*

Writes an HTML Select field with Options.

Keyword Arguments:

size Integer, Default Value: 1, determines size of selection list. Default of 1 is a pulldown menu.

name Keyword symbol or string, Default Value: `nil`, Determines the name of the field, which should probably match a settable computed-slot.

keys List of strings, numbers, or symbols, Default Value: `nil`, Values, the selected one of which will be returned as the value of the field.

values List of strings, Default Value: keys, Keys to display in the selection-list.

tabindex Integer, Default Value: `nil`, If given, this will generate the tabindex tag for this HTML input field.

98

write-child-links *void*

Creates a default unordered list with links to each child part of self. The text of the links will come from each child's strings-for-display.

write-development-links *void*

Writes links for access to the standard developer views of the object, currently consisting of an update (Refresh!) link, a Break link, and a ta2 link.

write-html-sheet *void*

This GDL function should be redefined to generate the HTML page corresponding to this object. It can be specified here, or as the `main-sheet` output-function in an html-format lens for this object's type. This `write-html-sheet` function, if defined, will override any `main-sheet` function defined in the lens. Typically a `write-html-sheet` function would look as follows:

write-self-link *void*

Emits a hyperlink pointing to self. Note that if you need extra customization on the display-string (e.g. to include an image tag or other arbitrary markup), use with-output-to-string in conjunction with the html-stream macro.

Keyword Arguments:

display-string String, Default Value: (the strings-for-display), String to be displayed.

display-color Keyword symbol or HTML color string, Default Value: nil, Determines the color of the displayed link text. Default of NIL indicates web browser default (usually blue).

target String, Default Value: (the target), Names a frame or window to open the link when clicked.

class String, Default Value: nil, Names a stylesheet class.

id String, Default Value: nil, Names a stylesheet id.

on-mouse-over String, Default Value: nil, Javascript code to run on mouse over.

on-mouse-out String, Default Value: nil, Javascript code to run on mouse out.

write-standard-footer *void*

Writes some standard footer information. Defaults to writing Genworks and Franz copyright and product links. Note that VAR agreements often require that you include a "powered by" link to the vendor on public web pages.

7.0.2.7 Checkbox-form-control

Description

This represents a INPUT of TYPE CHECKBOX

Optional-input-slots

domain *keyword symbol*

The domain defaults to :boolean for the checkbox-form-control However, this can be overridden in user code if the checkbox is supposed to return a meaningful value other than nil or t (e g for a group of checkboxes with the same name, where each can return a different value)

possible-nil? *boolean*

Indicates whether this should be included in possible-nils Defaults to t

Examples

Please see base-form-control for all the examples.

7.0.2.8 Grid-form-control

Description

Beginnings of spread-sheet-like grid control.

To do: Add row button, sort by column values, save & restore snapshot. Easy way for user to customize layout and markup.

Allow for all types of form-control for each column.

Optional-input-slots

default *list of lists*

These values become the default row and column values for the grid

form-control-attributes *list of plists*

Each plist contains the desired form-control inputs for the respective column in the table

form-control-inputs *list of lists plists*

Each list corresponds to one row and contains plists desired form-control inputs for the respective column in the table

form-control-types *list of symbols naming gdl object types*

This must be the same length as a row of the table The corresponding form-element in the grid will be of the specified type Default is nil, which means all the form-controls will be of type 'text-form-control

include-delete-buttons? *boolean*

Should each row have a delete button? Default is nil

row-labels *list of strings*

One for each row

Computed-slots

100

form-controls *list of gdl objects*

All the children or hidden-children of type base-form-control

7.0.2.9 Gwl-rule-object

Description

Used to display a rule as a GWL web page. Mixes together `base-html-sheet` and `base-rule-object`.

7.0.2.10 Menu-form-control

Description

This represents a SELECT form control tag wrapping some OPTION tags. OPTION-GROUP is not yet implemented, but will be.

Optional-input-slots

choice-list *list*

Display values, also used as return values, for selection list Specify this or choice-plist, not both

choice-plist *plist*

Keywords and display values for the selection list Specify this or choice-list, not both

choice-styles *plist*

Keywords and CSS style for display of each choice The keys should correspond to the keys in choice-plist, or the items in choice-list if no choice-plist is given

disabled-keys *list of keyword symbols*

Each of these should match a key in the choice-plist, and where there is a match, that key will be disabled in the rendering

multiple? *boolean*

Are multiple selections allowed? Default is nil

possible-nil? *boolean*

Indicates whether this should be included in possible-nils Defaults to (the multiple?)

size *number*

How many choices to display

test *predicate function of two arguments*

Defaults based on type of first in choice-plist: eql for keywords, string-equal for strings, and equalp otherwise

Examples

```
...

:objects
((menu-1 :type 'menu-form-control
         :choice-plist (list 1 "one" 2 "two")))

...
```

Please see base-form-control for a broader example which uses more form-control primitives together.

7.0.2.11 Node-mixin

Description

Generates a default GWL user interface with a model-inputs area, user-navigable tree with child applications, graphics view with controls, and rule display.

Child objects should be of type `node-mixin` or `application-mixin`. Child hidden-objects may be of any type.

The `ui-display-list-objects` is appended up automatically from those of the children.

Optional-input-slots

default-tree-depth *integer*

Determines how many descendant levels to show in the tree initially Default is 1

node-ui-display-list-objects *gdl object list*

Appends additional objects to the automatically-appended `ui-display-list-objects` from the children

Computed-slots

ui-display-list-leaves *list of gdl objects*

This should be overridden with a list of objects of your choice These objects (not their leaves, but these actual nodes) will be scaled to fit and displayed in the graphics area Defaults to NIL

ui-display-list-objects *list of gdl object roots*

The leaves of these objects will be displayed in the graphics Defaults to the appended result of children's `ui-display-list-objects`

7.0.2.12 Radio-form-control

Description

!!! Not applicable for this object !!!

Optional-input-slots

description-position *keyword symbol or nil*

Specifies where the description for each radio goes, if any Can be:

:paragraph-prepend (or :p-prepend or :p)

Description goes in a paragraph tag before the input tag

:paragraph-append (or :p-append)

Description goes in a paragraph tag after the input tag

:table-row-prepend (or :table-tr or :table-tr-prepend)

Description goes in a table cell wrapped in a table row before the input tag table cell

:table-row-append (or :table-tr-append)

Description goes in a table cell wrapped in a table row after the input tag table cell

nil (or any other value)

No description, only the bare input tag for the radio

Default is :paragraph-append

table-class *string*

Allows you to specify a class for the table surrounding the radio input elements Defaults to empty string

Computed-slots

multiple? *boolean*

Are multiple selections allowed? Default is nil

7.0.2.13 Session-control-mixin

Description

Mixin to the root object of the part which you wish to have session control over

Optional-input-slots

org-type *type of original object, useful when viewing session report log*

recovery-expires-at *expiration time of the recovery object*

After the recovery object has replaced the orginal instance at what time should the recovery instance expire?

recovery-url *url to which a user will be redirected if requesting a session that has been cleared*

session-duration *length of time a session should last without activity in minutes*

use-recovery-object? *boolean*

Determines whether expired sessions are replaced by recovery object Default is nil

Settable-optional-input-slots

expires-at *universal time after which the session should expire*

Functions

This is the function called to check for and handle session control

Keyword Arguments:

  debug? Boolean, Default Value: nil, Prints debug statement if needed.

clear-now? *boolean*

Test to run to see if this session has expired and needs to be cleared now.

session-clean-up *gets called right before the instance is going to get cleared*

Is intended to be used to stop any instance states that may not be elequently handled by the garbage collector. ie database connections, multiprocessing locks, open streams etc.

Method which will set the expires-at slot to the current time + the session-duration

7.0.2.14 Sheet-section

Description

Basic mixin to support an object representing a section of an HTML sheet (i.e. web page). Currently this simply mixes in skeleton-ui-element, and the functionality is not extended. Sheet-section is also mixed into base-html-sheet, so it and any of its subclasses will be considered as sheet-sections if they are the child of a base-ajax-sheet.

Examples

```
FLAG -- fill in!!!
```

7.0.2.15 Skeleton-ui-element

Description

104

Basic mixin to support constructing a gdl ajax call relative to this node. Note that in order for a node to represent a section of a web page, you should use sheet-section (which mixes this in), rather than this raw primitive.

This is a mixin into base-html-sheet, and some of the previous base-html-sheet functionality has been factored out into this mixin.

Of special note in this object is thefunction `gdl-ajax-call` which generates Javascript appropriate for attaching with a UI event, e.g. onclick, onchange, onblur, etc. In this Javascript you can specify a GDL function (on this object, self) to be run, and/or specify a list of form-control objects which are rendered on the current page, whose values should be submitted and processed ("bashed") into the server.

Optional-input-slots

bashee *gdl object*

Object to have its settable computed-slots and/or query-plist set from the fields on the form upon submission Defaults to self

dom-id *string*

This is the auto-computed dom-id which should be used for rendering this section If you use the main-div HTML string for rendering this object as a page section, then you do not have to generate the :div tag yourself - the main-div will be a string of HTML which is wrapped in the correct :div tag already

force-validation-for *list of gdl objects of type form-control*

The validation-function will be forced on these objects when a form is submitted, even if the object's html form-control does not happen to be included in the values submitted with the form Defaults to nil

inner-html *string*

This can be used with (str) [in cl-who] or (:princ) [in htmlGen] to output this section of the page, without the wrapping :div tag [so if you use this, your code would be responsible for wrapping the :div tag with :id (the dom-id)]

js-to-eval *string of valid javascript*

This Javascript will be send with the Ajax response, and evaluated after the innerHTML for this section has been replaced

Defaulted-input-slots

respondent *gdl object*

Object to respond to the form submission Defaults to self

Computed-slots

failed-form-controls *list of gdl objects*

All the form-controls which do not pass validation

form-controls *list of gdl objects*

All the children or hidden-children of type base-form-control

html-sections *list of html sections to be scanned and possibly replaced in response to gdl ajax calls*

Override this slot at your own risk The default is all sections who are most recently laid out on the respondent sheet, and this is set programmatically every time the sheet section's main-div is demanded

main-div% *string*

This should be used with (str) [in cl-who] or (:princ) [in htmlGen] to output this section of the page, including the wrapping :div tag

ordered-form-controls *list of gdl objects, which should be of type 'base-form-control*

[Note – this slot is not really necessary for protecting out-of-bounds sequence references anymore, the form-control processor protects against this by itself now]

These objects are validated and bashed first, in the order given If the cardinality of one form-control depends on another as in the example below, then you should list those dependent objects first Default is nil

:examples

```
:computed-slots ((number-of-nozzles (the number-of-nozzles-form value))
  (ordered-form-controls
   (append (list-elements (the inner-flange-form))
   (list (the number-of-nozzles-form)))))

:objects
((inner-flange-form
  :type 'menu-form-control
  :choice-plist (list :hey "hey" :now "now")
  :default :hey
  :sequence (:size (the number-of-nozzles)))

 (number-of-nozzles-form
  :type 'text-form-control
  :prompt "Number of Shell Nozzles Required: "
  :domain :number
  :default 0)
```

possible-nils *list of keyword symbols*

Messages corresponding to form fields which could be missing from form submission (e g checkbox fields) Defaults to the names of any children or hidden-children of type menu-form-control or checkbox-form-control

preset-all? *boolean*

This switch determines whether all form-controls should be preset before the final setting, in order to allow any interdependencies to be detected for validation or detecting changed values If this is specified as a non-nil value, then any nil values of (the preset?) on individual form controls will be ignored If this is specified as nil, then (the preset?) of individual form-controls (default of these is also nil) will be respected Default is nil

Functions

gdl-ajax-call *string*

. This function returns a string of Javascript, appropriate to use for events such as :onclick, :onchange, etc, which will invoke an Ajax request to the server, which will respond by replacing the innerHTML of affected :div's, and running the Javascript interpreter to evaluate (the js-to-eval), if any.

Keyword Arguments:

bashee GDL Object, Default Value: (the bashee), This object will have the function-key called on it, if any.

respondent GDL Object, Default Value: (the respondent), This must be the object which represents the actual web page being used.

function-key Keyword symbol, Default Value: nil, This keyword symbol must name a GDL function or method which is to be invoked with the Ajax call.

arguments List of values, Default Value: nil, This is the argument list on which the function named by function-key will be applied.

form-controls List of GDL objects of type base-form-control, Default Value: nil, Each of the objects in this list will have its current value (as entered by the user) scraped from the web page and its value in the model .

bashed to reflect what has been entered on the page, Default Value: `nil`,

Examples

```
FLAG -- Fill in!!!
```

7.0.2.16 Text-form-control

Description

This represents a INPUT TYPE=TEXT or TEXTAREA form control tag.

Optional-input-slots

cols *integer*

The number of columns for a TEXTAREA (if rows is > 1) Defaults to (the size)

number? *boolean*

Specifies whether this should be a number form control with support for numerical input Defaults to nil Use number-form-control to get a default of t

password? *boolean*

Specifies whether this should be a password form control with obscured screen text Note that this does not automatically give encrypted transmission to the server - you need SSL for that Defaults to nil Use password-form-control to get a default of t

rows *integer*

The number of rows If more than 1, this will be a TEXTAREA Defaults to 1

Examples

Please see base-form-control for all the examples.

7.0.2.17 Web-drawing

Description

Container object for displaying a view of geometric or text-based entities in a web application. This is supposed to be the type of the view-object hidden-child of base-html-graphics-sheet. Also, in a GWL application using application-mixin, you can include one object of this type in the ui-display-list-leaves.

Optional-input-slots

immune-objects *list of gdl objects*

These objects are not used in computing the scale or centering for the display list Defaults to nil

object-roots *list of gdl objects*

The leaves of each of these objects will be included in the geometry display Defaults to nil

objects *list of gdl objects*

These nodes (not their leaves but the actual objects) will be included in the geometry display Defaults to nil

projection-vector *3d vector*

This is the normal vector of the view plane onto which to project the 3D objects Defaults to (getf *standard-views* :top)

raphael-canvas-id *string*

Unique ID on the page for the raphael canvas div By default this is passed in from the base-ajax-graphics-sheet and based on its root-path, but can be specified manually if you are making a web-drawing on your own Defaults (in the standalone case) to "RaphaelCanvas"

Computed-slots

center *3d point*

Indicates in global coordinates where the center of the reference box of this object should be located

image-file *pathname or string*

Points to a pre-existing image file to be displayed instead of actual geometry for this object Defaults to nil

Objects

main-view *gdl object of type geom-base:base-view*

This is the actual drawing view which is used to present the geometry Defaults to an internally-computed object, this should not be overridden in user code

Examples

```
(in-package :gwl-user)

(define-object test-html-graphics-sheet (base-html-graphics-sheet)

  :objects

  ((b-splines :type 'test-b-spline-curves)

   (boxed-spline :type 'surf:boxed-curve
                 :curve-in (the b-splines (curves 0))
                 :orientation (alignment :top (the (face-normal-vector :rear)))
                 :show-box? t)

   (view-object :type 'web-drawing
                :page-length (the graphics-height value)
                :page-width (the graphics-width value)
                :projection-vector (getf *standard-views* (the view))
                :object-roots (the ui-display-roots))

   (graphics-height :type 'text-form-control
                    :default 350)

   (graphics-width :type 'text-form-control
                   :default 500)

   (bg-color :type 'text-form-control
             :default :black)

   (fg-color :type 'text-form-control
```

```
                       :default :white))

           :computed-slots
           ((background-color (lookup-color (the :bg-color value) :format :decimal))
            (foreground-color (lookup-color (the :fg-color value) :format :decimal))

            (view :trimetric :settable)

            ("list of gdl objects. Objects to be displayed in the graphics window."
             ui-display-roots (list (the b-splines) (the boxed-spline)))))

         (define-lens (html-format test-html-graphics-sheet)()

           :output-functions

           ((main-sheet
             ()
             (with-html-output (*html-stream* nil :indent t)
               (:html (:head (:title "Test HTML Graphics Sheet"))
                      (:body (when gwl:*developing?* (the write-development-links))
                             (:h2 (:center "Test HTML Graphics Sheet"))
                             (with-html-form (:cl-who? t)
                               (:table (:tr (:td (:ul
                                                     (:li (str (the graphics-height html-string)))
                                                     (:li (str (the graphics-width html-string)))
                                                        (:li (str (the bg-color html-string)))
                                                        (:li (str (the fg-color html-string))))
                                                   (:p (:input :type :submit :value " OK ")))
                                            (:td (write-the geometry)))))))))))

         (publish :path "/t-h-g-s"
                  :function #'(lambda(req ent)
                                (gwl-make-object req ent "gwl-user::test-html-graphics-sheet")))

         (define-object test-b-spline-curves (base-object)

           :input-slots

           ((control-points (list (make-point 0 0 0)
                                  (make-point 2 3.0 0.0)
                                  (make-point 4 2.0 0.0)
                                  (make-point 5 0.0 0.0)
                                  (make-point 4 -2.0 0.0)
                                  (make-point 2 -3.0 0.0)
                                  (make-point 0 0 0))))

           :objects

           ((curves :type 'surf:b-spline-curve
                    :sequence (:size 6)
                    :control-points (the control-points)
                    :degree (1+ (the-child :index))
                    :display-controls (list :line-thickness (* 0.3 (the-child index))
                                            :color (ecase (the-child index)
                                                     (0 :red) (1 :orange) (2 :yellow) (3 :green)
                                                     (4 :blue) (5 :red-violet))))

            (points :type 'point
```

```
:sequence (:size (length (rest (the control-points))))
:center (nth (the-child index) (rest (the control-points)))
:display-controls (list :color :green))))
```

7.0.3 Geom-base

7.0.3.1 Angular-dimension

Description

This dimensional object produces a clear and concise arc dimensional annotation.

Required-input-slots

arc-object *gdl object*

The arc being measured

Optional-input-slots

center-point *3d point*

The center of the arc being measured

dim-text-start *3d point*

Determines where the text will start Defaults to halfway along the arc, just beyond the radius

end-point *3d point*

The end point of the arc being measured

leader-radius *number*

The radius for the leader-arc

start-point *3d point*

The start point of the arc being measured

text-along-leader-padding-factor *number*

Amount of padding above leader for text-along-leader? t This is multiplied by the character-size to get the actual padding amount Defaults to 1/3

witness-1-to-center? *boolean*

Determines whether a witness line extends all the way from the start-point to the center Defaults to nil

witness-2-to-center? *boolean*

Determines whether a witness line extends all the way from the end-point to the center Defaults to nil

Computed-slots

dim-value *number*

    2D distance relative to the base-plane-normal Can be over-ridden in the subclass

Examples

```
(in-package :gdl-user)

(define-object angular-dimension-test (base-object)

  :objects
  ((arc :type 'arc
        :display-controls (list :color :green )
        :radius 30
        :end-angle (degrees-to-radians 90))


   (dimension :type 'angular-dimension
              :display-controls (list :color :blue )
              :leader-radius (+ (* 0.1 (the arc radius))(the arc radius))
              :arc-object (the arc))

   (explicit-dimension :type 'angular-dimension
                       :center-point (the arc center)
                       :start-point (the arc (point-on-arc (degrees-to-radians 10)))
                       :end-point (the arc (point-on-arc (degrees-to-radians 60))))))
(generate-sample-drawing
 :objects (list
           (the-object (make-object 'angular-dimension-test) arc)
           (the-object (make-object 'angular-dimension-test) dimension)
           (the-object (make-object 'angular-dimension-test) explicit-dimension))
 :projection-direction (getf *standard-views* :top))
```

112
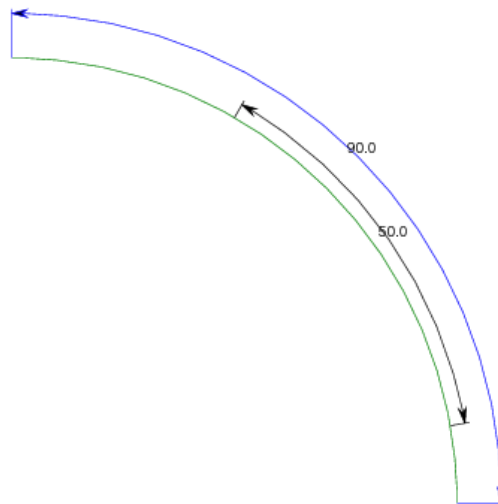
7.0.3.2 Arc

Description

A segment of a circle. The start point is at the 3 o'clock position, and positive angles are measured anti-clockwise.

Required-input-slots

radius *number*
    Distance from center to any point on the arc

Optional-input-slots

end-angle *angle in radians*
    End angle of the arc Defaults to twice pi

start-angle *angle in radians*
    Start angle of the arc Defaults to zero

Computed-slots

end *3d point*
    The end point of the arc

height *number*
    Z-axis dimension of the reference box Defaults to zero

length *number*
    Y-axis dimension of the reference box Defaults to zero

start *3d point*
    The start point of the arc

width *number*
    X-axis dimension of the reference box Defaults to zero

Functions

equi-spaced-points *list of points*
    Returns a list of points equally spaced around the arc, including the start and end point of the arc.
    Optional Arguments:

> **number-of-points** Number, Default Value: 4, How many points to return.

point-on-arc *3d point*

   The point on the arc at a certain angle from the start.

tangent *3d vector*

   Returns the tangent to the arc at the given point (which should be on the arc).

Examples

```
(in-package :gdl-user)

(define-object arc-sample (arc)
  :computed-slots ((radius 30) (end-angle (half pi/2)))))

(generate-sample-drawing :objects (make-object 'arc-sample))
```
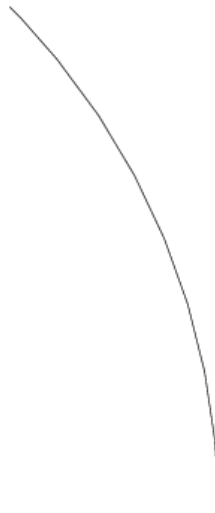
7.0.3.3 Base-drawing

Description

   Generic container object for displaying one or more scaled transformed views of geometric or text-based entities. The contained views are generally of type `base-view`. In a GWL application-mixin, you can include one object of this type in the ui-display-list-leaves.

   For the PDF output-format, you can also use the cad-output output-function to write the drawing as a PDF document.

   Since base-drawing is inherently a 2D object, only the top view (getf *standard-views* :top) makes sense for viewing it.

Optional-input-slots

height *number*

114

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

page-length *number in pdf points*

Front-to-back (or top-to-bottom) length of the paper being represented by this drawing The default is (* 11 72) points, or 11 inches, corresponding to US standard letter-size paper

page-width *number in pdf points*

Left-to-right width of the paper being represented by this drawing The default is (* 8 5 72) points, or 8 5 inches, corresponding to US standard letter-size paper

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object base-drawing-sample (base-drawing)

  :objects
  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :object-roots (list (the surf)))

   (surf :type 'cylinder-sample
         :hidden? t)))

(generate-sample-drawing :objects (make-object 'base-drawing-sample))
```
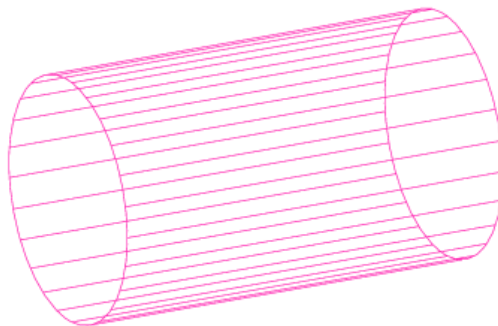
7.0.3.4 Base-object

Description

Base-Object is a superclass of most of GDL's geometric primitives. It provides an imaginary geometric reference box with a length, width, height, center, and orientation.

Optional-input-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

image-file *pathname or string*

Points to a pre-existing image file to be displayed instead of actual geometry for this object Defaults to nil

local-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding this geometric object

obliqueness *3x3 orthonormal matrix of double-float numbers*

This is synonymous with the `orientation`

Defaulted-input-slots

center *3d point*

Indicates in global coordinates where the center of the reference box of this object should be located

display-controls *plist*

May contain keywords and values indicating display characteristics for this object The following keywords are recognized currently:

:color

color keyword from the *color-table* parameter, or an HTML-style hexidecimal RGB string value, e g "#FFFFFF" for pure white Defaults to :black

:line-thickness

an integer, defaulting to 1, indicating relative line thickness for wireframe representations of this object

:dash-pattern

(currently PDF/PNG/JPEG only) This is a list of two or three numbers which indicate the length, in pixels, of the dashes and blank spaces in a dashed line The optional third number indicates how far into the line or curve to start the dash pattern

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

orientation *3x3 matrix of double-float numbers*

Indicates the absolute Rotation Matrix used to create the coordinate system of this object This matrix is given in absolute terms (i e with respect to the root's orientation), and is generally created with the alignment function It should be an *orthonormal* matrix, meaning each row is a vector with a magnitude of one (1 0)

width *number*

X-axis dimension of the reference box Defaults to zero

Computed-slots

color-decimal *vector of three real numbers*

The RBG color of this object specified in :display-controls Defaults to the foreground color specified in `*colors-default*` This message should not normally be overridden in user application code

local-center *3d point*

The center of this object, from the perspective of the parent Starting from the parent's center and using the parent's orientation, this is the relative center of this object

local-orientation *3x3 matrix of double-float numbers*

Indicates the local Rotation Matrix used to create the coordinate system of this object This is the "local" orientation with respect to the parent Multiplying the parent's orientation with this matrix will always result in the absolute orientation for this part

:note An orientation of NIL indicates the 3x3 identity matrix

Hidden-objects

bounding-bbox *gdl object of type box*

A box representing the bounding-box

local-bbox *gdl object of type box*

A box representing the local-box

Functions

axis-vector *3d vector*

Returns the vector pointing in the positive direction of the specified axis of this object's reference box.

edge-center *3d point*

Returns the center of the requested edge of this object's reference box.

face-center *3d point*

Returns the center of the requested face of this object's reference box.

face-normal-vector *3d vector*

Returns the vector pointing from this object's reference box center to its requested face-center.

face-vertices *list of four 3d points*

Returns the vertices of the indicated face.

global-to-local *3d-point*

This function returns the point given in global coordinates, into relative local coordinates, based on the orientation and center of the object to which the global-to-local message is sent.

in-face? *boolean*

Returns non-nil if the given point is in halfspace defined by the plane given a point and direction.

line-intersection-points *list of 3d points*

Returns the points of intersection between given line and the reference box of this object.

local-to-global *3d-point*

This function returns the point given in relative local coordinates, converted into global coordinates, based on the orientation and center of the object to which the local-to-global message is sent.

vertex *3d point*

Returns the center of the requested vertex (corner) of this object's reference box.

Examples

```
(in-package :gdl-user)

(define-object tower (base-object)

 :input-slots
  ((number-of-blocks 50) (twist-per-block 1)
   (block-height 1) (block-width 5) (block-length 7))

 :objects
  ((blocks :type 'box
           :sequence (:size (the number-of-blocks))
           :center (translate (the center)
                              :up (* (the-child index)
                                     (the-child height)))
           :width (the block-width)
           :height (the block-height)
           :length (the block-length)
           :orientation (alignment
```

```
                              :rear (if (the-child first?)
                                        (rotate-vector-d (the (face-normal-vector  :rear))
                                                         (the twist-per-block)
                                                         (the (face-normal-vector :top)))
                                        (rotate-vector-d (the-child previous
                                                                    (face-normal-vector :rear))
                                                         (the twist-per-block)
                                                         (the (face-normal-vector :top))))
                              :top (the (face-normal-vector :top))))))
;;
;;Test run
;;
#|
gdl-user(46): (setq self (make-object 'tower))
#&lt;tower (at) #x750666f2&gt;
gdl-user(47): (setq test-center (the (blocks 10) center))
#(0.0 0.0 10.0)
gdl-user(48): (the (blocks 10) (global-to-local test-center))
#(0.0 0.0 0.0)
gdl-user(49): (the (blocks 10) (local-to-global (the (blocks 10)
                                                (global-to-local test-center))))
#(0.0 0.0 10.0)
gdl-user(50):
gdl-user(50): (setq test-vertex (the (blocks 10) (vertex :top :right :rear)))
#(1.7862364748012536 3.9127176305081863 10.5)
gdl-user(51): (the (blocks 10) (global-to-local test-vertex))
#(2.500000000000001 3.500000000000001 0.5)
gdl-user(52): (the (blocks 10) (local-to-global (the (blocks 10)
                                                (global-to-local test-vertex))))
#(1.786236474801254 3.9127176305081877 10.5)
gdl-user(53):
|#
;;
;;
;;
```
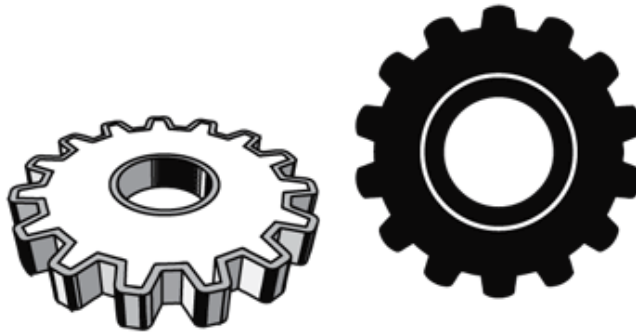
Image not applicable for this GDL object

7.0.3.5 Base-view

Description

Generic container object for displaying a scaled transformed view of geometric or text-based objects. `Base-view` can be used by itself or as a child of a `base-drawing`

In a GWL application-mixin, you can include an object of this type in the ui-display-list-leaves.

For the PDF output-format, you can also use the cad-output output-function to write the view as a PDF document.

Since base-view is inherently a 2D object, only the top view (getf *standard-views* :top) makes sense for viewing it.

Optional-input-slots

center *3d-point*

Center of the view box Specify this or corner, not both NOTE that the center is no longer defaulting (so that it can self-compute properly when corner is specified), so it is necessary to explicitly give either start or center for base-view

corner *3d-point*

Top left (i e rear left from top view) of the view box Specify this or center, not both

immune-objects *list of gdl objects*

These objects are immune from view scaling and transform computations and so can freely refer to the view-scale, view-center, and other view information for self-scaling views Defaults to NIL

snap-to *3d vector*

For a top view, this vector specifies the direction that the rear of the box should be facing Defaults to `*nominal-y-vector*`

Defaulted-input-slots

annotation-objects *list of gdl objects*

These objects will be displayed in each view by default, with no scaling or transform (i e they are in Drawing space

border-box? *boolean*

Determines whether a rectangular border box is drawn around the view, with the view's length and width Defaults to nil

front-margin *number in drawing scale (e*

g points) Amount of margin on front and rear of page when `view-scale` is to be computed automatically Defaults to 25

left-margin *number in drawing scale (e*

g points) Amount of margin on left and right of page when `view-scale` is to be computed automatically Defaults to 25

object-roots *list of gdl objects*

The leaves from each of these objects will be displayed in each view by default

objects *list of gdl objects*

These objects will be displayed in each view by default

projection-vector *3d unitized vector*

Direction of camera pointing to model (the object-roots and/or the objects) to create this view The view is automatically "twisted" about this vector to result in "up" being as close as possible to the Z vector, unless this vector is parallel to the Z vector in which case "up" is taken to be the Y (rear) vector This vector is normally taken from the `*standard-views*` built-in GDL parameter Defaults to `(getf *standard-views* :top)`, which is the vector [0, 0, 1]

view-center *3d point in model space*

Point relative to each object's center to use as center of the view

view-scale *number*

Ratio of drawing scale (in points) to model scale for this view Defaults to being auto-computed

Functions

model-point *3d point*

Takes point in view coordinates and returns corresponding point in model coordinates.

view-point *3d point*

Takes point in model coordinates and returns corresponding point in view coordinates.

Examples

```
(in-package :gdl-user)

(define-object box-with-two-viewed-drawing (base-object)

  :objects
  ((drawing :type 'two-viewed-drawing
            :objects (list (the box) (the length-dim)))

   (length-dim :type 'horizontal-dimension
               :hidden? t
               :start-point (the box (vertex :rear :top :left))
               :end-point (the box (vertex :rear :top :right)))
```

```
    (box :type 'box
         :hidden? t
         :length 5 :width 10 :height 15)))

(define-object two-viewed-drawing (base-drawing)

  :input-slots (objects)

  :objects

  ((main-view :type 'base-view
              :projection-vector (getf *standard-views* :trimetric)
              :length (half (the length))
              :center (translate (the center)
                                 :rear (half (the-child length)))
              :objects (the objects))

   (top-view :type 'base-view
             :projection-vector (getf *standard-views* :top)
             :length (* 0.30 (the length))
             :objects (the objects))))

  (generate-sample-drawing :objects
   (the-object (make-object 'box-with-two-viewed-drawing) drawing top-view))
```
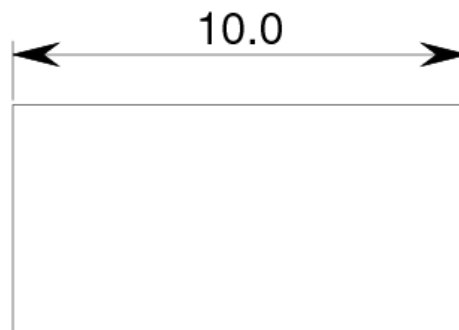

7.0.3.6 Bezier-curve

Description

GDL currently supports third-degree Bezier curves, which are defined using four 3D *control-points* . The Bezier curve always passes through the first and last control points and lies within the convex hull of the control points. At the start point (i.e. the first control point), the curve is tangent to the vector pointing from the start point to the

second control point. At the end point (i.e. the last control point), the curve is tangent to the vector pointing from the end point to the third control point.

Required-input-slots

control-points *list of 4 3d points*

Specifies the control points for the Bezier curve

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

Functions

circle-intersection-2D *list of 3d points*

Returns points of intersection in the Z plane between this Bezier curve and the circle in the Z plane with center `center` and radius `radius`.

Keyword Arguments:

accuracy Number, Default Value: (* 10.0 double-float-epsilon), Target accuracy.

maximum-number-of-iterations Integer, Default Value: 30, Maximum iterations of polynomial solver.

line-intersection-2D *list of 3d points*

Returns points of intersection in the Z plane between this Bezier curve and the infinite line containing point `point` and direction `vector`. Use the between? function if you wish to establish whether the point is contained in a particular line segment.

Keyword Arguments:

accuracy Number, Default Value: (* 10.0 double-float-epsilon), Target accuracy.

maximum-number-of-iterations Integer, Default Value: 30, Maximum iterations of polynomial solver.

point *3d point*

Returns the point on this Bezier curve corresponding to the given `parameter`, which should be between 0 and 1.

Examples

```
(in-package :gdl-user)

(define-object bezier-sample (bezier-curve)
 :computed-slots
 ((control-points (list (make-point 0 0 0)
                        (make-point 1 1 0)
```

```
                            (make-point 2 1 0)
                            (make-point 3 0 0)))) 
 :objects
 ((points-display :type 'points-display
                  :points (the control-points)))) 

(generate-sample-drawing :objects (let ((self (make-object 'bezier-sample)))
                                     (list self (the points-display))))
```

7.0.3.7 Box

Description

This represents a "visible" base-object – a six-sided box with all the same messages as base-object, which knows how to output itself in various formats.

Computed-slots

volume *number*

Total volume of the box

Examples

```
(in-package :gdl-user)

(define-object box-sample (box)
  :computed-slots ((display-controls (list :color :blue-neon))
                   (length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+))))

(generate-sample-drawing :objects (make-object 'box-sample)
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.3.8 C-cylinder

Description

Provides a simple way to create a cylinder, by specifying a start point and an end point.

Required-input-slots

end *3d point*

Center of the end cap

start *3d point*

Center of the start cap

Computed-slots

center *3d point*

Center point of the center-line

center-line *list of two 3d points*

Represents line segment connecting center of end cap to center of start cap

length *number*

Distance between cap centers

orientation *3x3 orthonormal rotation matrix*

Resultant orientation given the specified start and end points

Examples

```
(in-package :gdl-user)

(define-object c-cylinder-sample (c-cylinder)
  :computed-slots
  ((display-controls (list :color :plum :transparency 0.2))
   (start (make-point 0 0 0))
   (end (make-point 0 0 10))
   (number-of-sections 7)
   (radius 3)))

(generate-sample-drawing :objects (make-object 'c-cylinder-sample)
                         :projection-direction (getf *standard-views* :trimetric))
```


7.0.3.9 Center-line

Description

Creates a dashed single centerline or crosshair centerline on a circle.

Required-input-slots

size *number*

The length of the centerline

Optional-input-slots

circle? *boolean*

Determines whether this will be a circle crosshair Defaults to nil

Defaulted-input-slots

gap-length *number*

Distance between dashed line segments Defaults to 0 1

long-segment-length *number*

Length of longer dashed line segments Defaults to 1 0

short-segment-length *number*

Length of shorter dashed line segments Defaults to 0 25

Computed-slots

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object center-line-test (base-object)

 :objects
 ((circle-sample :type 'circle
                 :display-controls (list :color :green)
                 :center (make-point 10 10 10 )
                 :radius 10)

  (center-line-sample :type 'center-line
                      :circle? t
                      :center (the circle-sample center)
                      :size (* 2.1 (the circle-sample radius)))))

(generate-sample-drawing
:objects (list
           (the-object (make-object 'center-line-test)
                       circle-sample)
           (the-object (make-object 'center-line-test)
                       center-line-sample))
:projection-direction (getf *standard-views* :top))
```
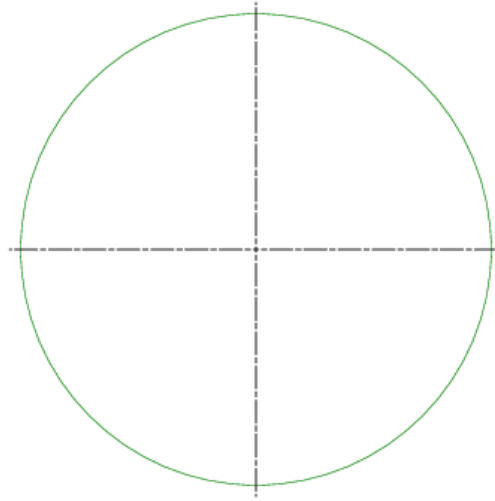
7.0.3.10 Circle

Description

The set of points equidistant from a given point. The distance from the center is called the radius, and the point is called the center. The start point of the circle is at the 3 o'clock position, and positive angles are measured anti-clockwise.

Computed-slots

area *number*

The area enclosed by the circle

circumference *number*

The perimeter of the circle

end-angle *angle in radians*

End angle of the arc Defaults to twice pi

start-angle *angle in radians*

Start angle of the arc Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object circle-sample (circle)
  :computed-slots
  ((radius 10)))

(generate-sample-drawing :objects (make-object 'circle-sample))
```
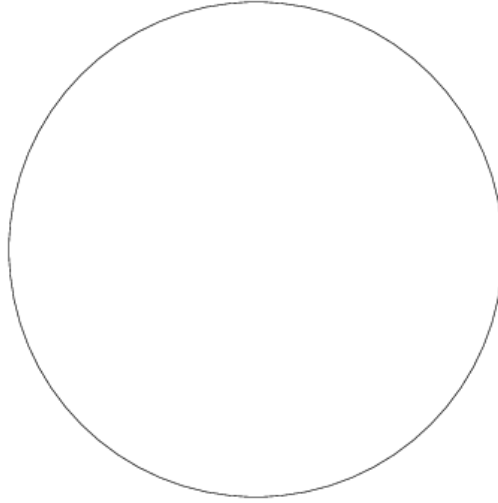
7.0.3.11 Cone

Description

A pyramid with a circular cross section, with its vertex above the center of its base. Partial cones and hollow cones are supported.

Optional-input-slots

inner-radius-1 *number*

The radius of the inner hollow part at the top end for a hollow cone

inner-radius-2 *number*

The radius of the inner hollow part at the bottom end for a hollow cone

radius-1 *number*

The radius of the top end of the cone

radius-2 *number*

The radius of the bottom end of the cone

Computed-slots

height *number*

Z-axis dimension of the reference box Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object cone-sample (cone)
  :computed-slots
  ((display-controls (list :color :blue-neon
                           :transparency 0.5
                           :shininess 0.8
                           :specular-color :white))
   (length 10) (radius-1 2)(inner-radius-1 1)
   (radius-2 5) (number-of-sections 5)
   (inner-radius-2 3)))

(generate-sample-drawing :objects (make-object 'cone-sample)
                         :projection-direction :trimetric)
```
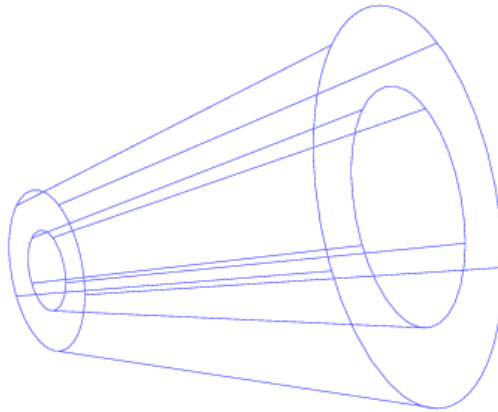
7.0.3.12 Cylinder

Description

An extrusion of circular cross section in which the centers of the circles all lie on a single line (i.e., a right circular cylinder). Partial cylinders and hollow cylinders are supported.

Required-input-slots

length *number*

Distance from center of start cap to center of end cap

radius *number*

Radius of the circular cross section of the cylinder

Optional-input-slots

bottom-cap? *boolean*

Determines whether to include bottom cap in shaded renderings Defaults to T

closed? *boolean*

Indicates that a partial cylinder (or cone) should have a closed gap

height *number*

Z-axis dimension of the reference box Defaults to zero

inner-radius *number*

Radius of the hollow inner portion for a hollow cylinder

number-of-sections *integer*

Number of vertical sections to be drawn in wireframe rendering mode

top-cap? *boolean*

Determines whether to include bottom cap in shaded renderings Defaults to T

width *number*

X-axis dimension of the reference box Defaults to zero

Computed-slots

direction-vector *3d vector*

Points from the start to the end

end *3d point*

The center of the end cap

hollow? *boolean*

Indicates whether there is an inner-radius and thus the cylinder is hollow

start *3d point*

The center of the start cap

Examples

```
(in-package :gdl-user)

(define-object cylinder-sample (cylinder)
  :computed-slots
  ((display-controls (list :color :pink-spicy))
   (length 10)
   (radius 3)
   (number-of-sections 25)))


(generate-sample-drawing :objects (make-object 'cylinder-sample)
                         :projection-direction (getf *standard-views* :trimetric))
```
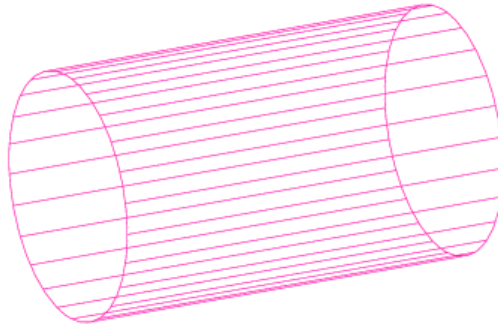
7.0.3.13 Ellipse

Description

A curve which is the locus of all points in the plane the sum of whose distances from two fixed points (the foci) is a given positive constant. This is a simplified 3D ellipse which will snap to the nearest quarter if you make it a partial ellipse. For a full ellipse, do not specify start-angle or end-angle.

Required-input-slots

major-axis-length *number*

Length of (generally) the longer ellipse axis

minor-axis-length *number*

Length of (generally) the shorter ellipse axis

Optional-input-slots

end-angle *angle in radians*

End angle of the ellipse Defaults to 2pi for full ellipse

start-angle *angle in radians*

Start angle of the ellipse Defaults to 0 for full ellipse

Computed-slots

height *number*

    Z-axis dimension of the reference box Defaults to zero

length *number*

    Y-axis dimension of the reference box Defaults to zero

width *number*

    X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object ellipse-sample (ellipse)
  :computed-slots
  ((minor-axis-length 10)
   (major-axis-length (* (the minor-axis-length) +phi+))
   (start-angle 0)
   (end-angle pi)))

(generate-sample-drawing :objects (make-object 'ellipse-sample))
```

7.0.3.14 General-note

Description

    Creates a text note in the graphical view port and in a PDF DXF output file.

Optional-input-slots

center *3d-point*

Center of the text Specify this or start, not both NOTE that the center is no longer defaulting (so that it can self-compute properly when start is specified), so it is necessary to explicitly give either start or center for general-note

character-size *number*

Specifies the character size in drawing units

dxf-font *string*

This names the DXF font for this general-note Defaults to (`the font`)

dxf-offset *number*

The start of text will be offset by this amount for DXF output Default is 0

dxf-size-ratio *number*

The scale factor for DXF character size vs PDF character size Default is 0 8

dxf-text-x-scale *number in percentage*

Adjusts the character width for DXF output Defaults to the text-x-scale

font *string*

The font for PDF Possibilities for built-in PDF fonts are:

- courier
- courier-bold
- courier-boldoblique
- courier-oblique
- helvetica
- helvetica-bold
- helvetica-boldoblique
- helvetica-oblique
- symbol
- times-roman
- times-bold
- times-bolditalic
- times-italic
- zapfdingbats

Defaults to "Courier"

height *number*

Z-axis dimension of the reference box Defaults to zero

justification *keyword symbol, :left, :right, or :center*

Justifies text with its box Default is :left

leading *number*

Space between lines of text Default is 1 2 times the character size

length *number*

Y-axis dimension of the reference box Defaults to zero

outline-shape-type *keyword symbol*

Currently can be :bubble, :rectangle, or :none Default is :none

start *3d-point*

Start of the text Specify this or center, not both

strings *list of strings*

The text to be displayed in the note

text-x-scale *number in percentage*

Adjusts the character width for PDF output Defaults to 100

underline? *boolean*

Determines whether text is underlined

width *number*

Determines the width of the containing box Default is the maximum-text-width

Computed-slots

maximum-text-width *number*

Convienence computation giving the maximum input width required to keep one line per string

Examples

```
(define-object general-note-test (base-object)

 :computed-slots

 ((blocks-note
  (list
   "David Brown" "Created by" "ABC 2"
   "Jane Smith" "Approved by" "CCD 2"))
  (blocks-center
   (list '(-15  5 0) '(-40  5 0) '(-55  5 0)
         '(-15 15 0) '(-40 15 0) '(-55 15 0)))
  (blocks-width (list 30 20 10 30 20 10)))

 :objects

 ((title-block :type 'box
               :sequence (:size (length (the blocks-center)))
               :display-controls (list :color :red)
               :center (apply-make-point
                         (nth (the-child index )
```

```
                              (the blocks-center)))
                   :length 10
                   :width (nth (the-child index )
                               (the blocks-width))
                   :height 0)

      (general-note-sample :type 'general-note
                           :sequence (:size (length (the blocks-note)))
                           :center (the (title-block
                                          (the-child index)) center)
                           :character-size 2.5
                           :strings (nth (the-child index)
                                          (the blocks-note)))))
      (generate-sample-drawing
      :objects (list-elements (make-object 'general-note-test))
      :projection-direction (getf *standard-views* :top))
```

CCD 2	Approved by	Jane Smith
ABC 2	Created by	David Brown

7.0.3.15 Global-filleted-polygon-projection

Description

Similar to a global-polygon-projection, but the polygon is filleted as with global-filleted-polygon.

Optional-input-slots

default-radius *number*

Specifies a radius to use for all vertices Radius-list will take precedence over this

radius-list *list of numbers*

Specifies the radius for each vertex ("corner") of the filleted-polyline

Examples

```
(in-package :gdl-user)

(define-object global-filleted-polygon-projection-sample
    (global-filleted-polygon-projection)
    :computed-slots
    ((display-controls (list :color :blue-steel
                             :transparency 0.3
                             :shininess 0.7
                             :spectral-color :white))
     (default-radius 5)
     (projection-depth 5)
     (vertex-list (list (make-point 0 0 0)
                        (make-point 10 10 0)
                        (make-point 30 10 0)
                        (make-point 40 0 0)
                        (make-point 30 -10 0)
                        (make-point 10 -10 0)
                        (make-point 0 0 0)))))

(generate-sample-drawing :objects
                         (make-object 'global-filleted-polygon-projection-sample)
                         :projection-direction :trimetric)
```
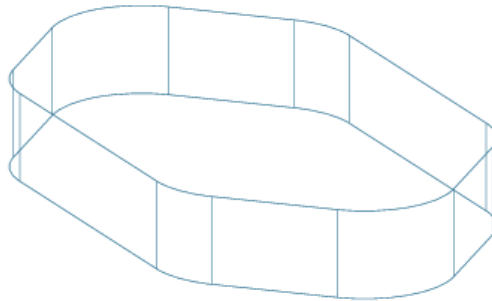

7.0.3.16 Global-filleted-polyline

Description

A sequence of points connected by straight line segments, whose corners are filleted according to specified radii. Please see global-filleted-polyline-mixin for documentation on the messages.

Examples

```
(in-package :gdl-user)

(define-object global-filleted-polyline-sample (global-filleted-polyline)
  :computed-slots
  ((default-radius 5)
    (vertex-list (list (make-point 0 0 0)
                       (make-point 10 10 0)
                       (make-point 30 10 0)
                       (make-point 40 0 0)
                       (make-point 30 -10 0)
                       (make-point 10 -10 0)
                       (make-point 0 0 0)))))

(generate-sample-drawing :objects (make-object 'global-filleted-polyline-sample))
```
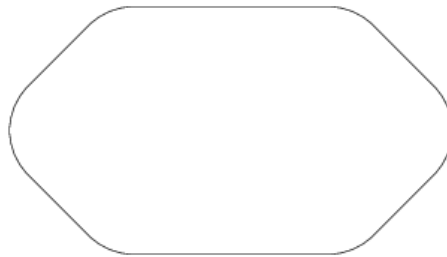
7.0.3.17 Global-polygon-projection

Description

A polygon "extruded" for a given distance along a single vector. For planar polygons, the projection vector must not be orthogonal to the normal of the plane of the polygon. The vertices and projection-vector are given in the global coordinate system, so the local center and orientation do not affect the positioning or orientation of this part.

Required-input-slots

projection-depth *number*

The resultant distance from the two end faces of the extrusion

vertex-list *list of 3d points*

The vertex list making up the polyline, same as the input for global-polyline

Optional-input-slots

offset *keyword symbol*

The direction of extrusion with respect to the vertices in vertex-list and the projection-vector:

- `:up` Indicates to start from current location of vertices and move in the direction of the projection-vector

- `:down` Indicates to start from current location of vertices and move in the direction opposite the projection-vector

- `:center` Indicates to start from current location of vertices and move in the direction of the projection-vector *and* opposite the projection-vector, going half the projection-depth in each direction

projection-vector *3d vector*

Indicates the straight path along which the extrusion should occur

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

Examples

```
(in-package :gdl-user)

(define-object global-polygon-projection-sample (global-polygon-projection)
  :computed-slots
  ((display-controls (list :color :gold-old :transparency 0.3))
   (projection-depth 5)
   (vertex-list (list (make-point 0 0 0)
                      (make-point 10 10 0)
                      (make-point 30 10 0)
                      (make-point 40 0 0)
                      (make-point 30 -10 0)
                      (make-point 10 -10 0)
                      (make-point 0 0 0)))))

(generate-sample-drawing :objects (make-object 'global-polygon-projection-sample)
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.3.18 Global-polyline

Description

A sequence of points connected by straight line segments. Please see global-polyline-mixin for documentation on the messages.

Examples

```
(in-package :gdl-user)

(define-object global-polyline-sample (global-polyline)
  :computed-slots
  ((vertex-list (list (make-point 0 0 0)
                      (make-point 10 10 0)
                      (make-point 30 10 0)
                      (make-point 40 0 0)
                      (make-point 30 -10 0)
                      (make-point 10 -10 0)
                      (make-point 0 0 0)))))

(generate-sample-drawing :objects (make-object 'global-polyline-sample))
```
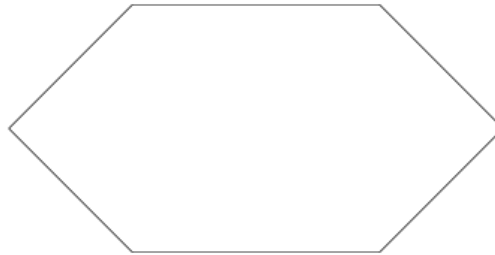
7.0.3.19 Horizontal-dimension

Description

Creates a dimension annotation along the horizontal axis.

Optional-input-slots

dim-text-start *3d point*

Determines where the text will start Defaults to reasonable location for horizontal-dimension

Computed-slots

base-plane-normal *must be specified in the subclass except for angular*

leader-direction-1-vector *must be specified in the subclass except for angular*

leader-direction-2-vector *must be specified in the subclass except for angular*

witness-direction-vector *must be specified in the subclass except for angular*

Examples

```
(in-package :gdl-user)

(define-object  box-view (base-object)

  :objects
  ((box :type 'box
        :length 10 :width (* (the-child length) +phi+)
        :height (* (the-child :width) +phi+))

   (width-dimension :type 'horizontal-dimension
```

```
                              :character-size (/ (the box length) 20)
                              :arrowhead-width (/ (the-child character-size) 3)
                              :start-point (the box (vertex :top :left :rear))
                              :end-point (the box (vertex :top :right :rear)))))

      (generate-sample-drawing :object-roots (make-object 'box-view))
```
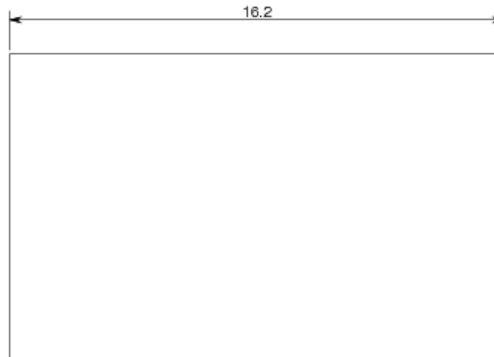
7.0.3.20 Label

Description

!!! Not applicable for this object !!!

Required-input-slots

leader-path *list of 3d points*

List making up leader line, starting from where the arrowhead normally is

Optional-input-slots

arrowhead-length *length (from tip to tail) of arrowhead glyph*

Defaults to twice the `arrowhead-width`

arrowhead-style *keyword symbol*

Style for arrowhead at start of `leader-path` Currently supported values are `:none`, `:wedge` (the Default), and `:double-wedge`

arrowhead-style-2 *keyword symbol*

Style for arrowhead on end of `leader-path` Currently supported values are `:none` (the Default), `:wedge`, and `:double-wedge`

arrowhead-width *width of arrowhead glyph*

Defaults to five times the line thickness (2 5)

character-size *number*

Size (glyph height) of the label text, in model units Defaults to 10

dxf-font *string*

This names the DXF font for this general-note Defaults to `(the font)`

dxf-offset *number*

The start of text will be offset by this amount for DXF output Default is 2

dxf-size-ratio *number*

The scale factor for DXF character size vs PDF character size Default is 0 8

dxf-text-x-scale *number in percentage*

Adjusts the character width for DXF output Defaults to the text-x-scale

font *string naming a standard pdf font*

Font for the label text Defaults to "Helvetica"

outline-shape-type *keyword symbol*

Indicates shape of outline enclosing the text Currently `:none`, `:bubble`, `:rectangle`, and `nil` are supported The default is nil

strings *list of strings*

Text lines to be displayed as the label Specify this or text, not both

text *string*

Text to be displayed as the label

text-gap *number*

Amount of space between last point in leader-path and beginning of the label text Defaults to the width of the letter "A" in the specified `font` and `character-size`

text-side *keyword symbol, either <tt>:left</tt> or <tt>:right</tt>*

Determines whether the label text sits to the right or the left of the last point in the `leader-path` The default is computed based on the direction of the last segment of the leader-path

Defaulted-input-slots

view-reference-object *gdl object or nil*

View object which will use this dimension Defaults to NIL

Computed-slots

orientation *3x3 matrix of double-float numbers*

Indicates the absolute Rotation Matrix used to create the coordinate system of this object This matrix is given in absolute terms (i e with respect to the root's orientation), and is generally created with the alignment function It should be an *orthonormal* matrix, meaning each row is a vector with a magnitude of one (1 0)

Examples

```
(in-package :gdl-user)

(define-object  label-sample (base-object)

  :objects
  ((box :type 'box
        :length 10 :width (* (the-child length) +phi+)
        :height (* (the-child :width) +phi+))

   (corner-label :type 'label
                 :leader-path (let ((start (the box (vertex :top :right :rear))))
                                (list start
                                      (translate start :right (/ (the box width) 10)
                                                       :rear (/ (the box width) 10))
                                      (translate start :right (/ (the box width) 7)
                                                       :rear (/ (the box width) 10))))
                 :text "The Corner"
                 :character-size (/ (the box width) 15))))


(generate-sample-drawing :object-roots (make-object 'label-sample))
```
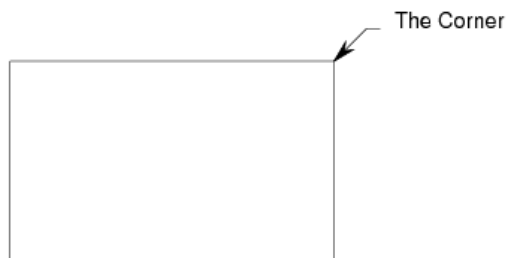

7.0.3.21 Leader-line

Description

Creates a leader line with arrows on zero, one, or both ends

Required-input-slots

path-points *list of 3d points*

Leader-line is rendered as a polyline going through these points

Optional-input-slots

arrowhead-length *number*

The length of the arrows Defaults to (* (the arrowhead-width) 2)

arrowhead-style *keyword*

Controls the style of first arrowhead Currently only :wedge is supported Default is :wedge

arrowhead-style-2 *keyword*

Controls the style and presence of second arrowhead Currently only :wedge is supported Default is :none

arrowhead-width *number*

The width of the arrows Defaults to (* (the line-thickness) 5)

break-points *list of two points or nil*

The start and end of the break in the leader line to accomodate the dimension-text, in cases where there is overlap

7.0.3.22 Line

Description

Provides a simple way to create a line, by specifying a start point and an end point.

Required-input-slots

end *3d point*

The end point of the line, in global coordinates

start *3d point*

The start point of the line, in global coordinates

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

center *3d point*

    The center of the line

direction-vector *3d vector*

    Points from start to end of the line

length *number*

    The distance from start to end of the line

Examples

```
(in-package :gdl-user)

(define-object line-sample (line)
  :computed-slots
  ((start (make-point -10 -10 0))
   (end (make-point 10 10 0))))

(generate-sample-drawing :objects (make-object 'line-sample))
```
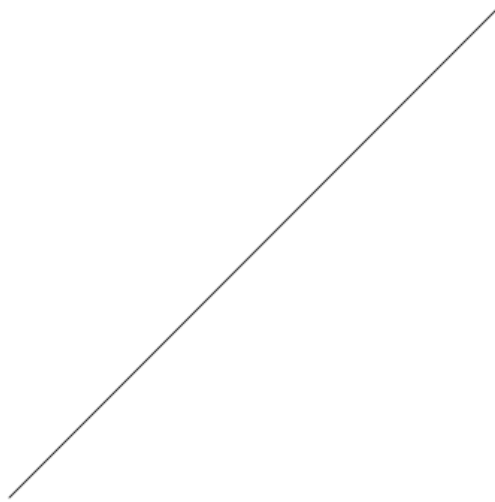

7.0.3.23 Linear-dimension

Description

    Creates a dimension along either the horizontal, vertical, or an arbitray axis. Use `horizontal-dimension`, `vertical-dimension`, or `parallel-dimension`, respectively, to achieve these.

Required-input-slots

base-plane-normal *must be specified in the subclass except for angular*

end-point *3d point*

 Actual point where the dimension will stop measuring

leader-direction-1-vector *must be specified in the subclass except for angular*

leader-direction-2-vector *must be specified in the subclass except for angular*

start-point *3d point*

 Actual point where the dimension will start measuring

witness-direction-vector *must be specified in the subclass except for angular*

Optional-input-slots

arrowhead-length *length (from tip to tail) of arrowhead glyph*

 Defaults to twice the `arrowhead-width`

arrowhead-style *keyword symbol*

 Style for arrowhead on end of `leader-line` Currently supported values are `:none`, `:wedge` (the Default), and `:double-wedge`

arrowhead-style-2 *keyword symbol*

 Style for arrowhead on end of `leader-line` Currently supported values are `:none` (the Default), `:wedge`, and `:double-wedge`

arrowhead-width *width of arrowhead glyph*

 Defaults to half the character-size

character-size *number*

 Size (glyph height) of the label text, in model units Defaults to 1

dim-text *string*

 Determines the text which shows up as the dimension label Defaults to the dim-value, which is computed specially in each specific dimension type

dim-text-bias *keyword symbol, :start, :end, or :center*

 Indicates where to position the text in the case when **outside-leaders?** is non-nil Defaults to :center

dim-text-start *3d point*

 Determines where the text will start Defaults to halfway between start-point and end-point

dim-text-start-offset *3d vector (normally only 2d are used)*

 The dim-text-start is offset by this vector, in model space Defaults to #(0 0 0 0 0 0)

dim-value *number*

2D distance relative to the base-plane-normal Can be over-ridden in the subclass

dxf-font *string*

This names the DXF font for this general-note Defaults to (`the font`)

dxf-offset *number*

The start of text will be offset by this amount for DXF output Default is 2

dxf-size-ratio *number*

The scale factor for DXF character size vs PDF character size Default is 0 8

dxf-text-x-scale *number in percentage*

Adjusts the character width for DXF output Defaults to the text-x-scale

flip-leaders? *boolean*

Indicates which direction the witness lines should take from the start and end points The Default is NIL, which indicates :rear (i e "up") for `horizontal-dimensions` and :right for `vertical-dimensions`

font *string naming a standard pdf font*

Font for the label text Defaults to "Helvetica"

full-leader-line-length *number*

Indicates the length of the full leader when `outside-leaders?` is nil This defaults to nil, which indicates that the full-leader's length should be auto-computed based on the given start-point and end-point

justification *keyword symbol, :left, :right, or :center*

For multi-line dim-text, this justification is applied

leader-1? *boolean*

Indicates whether the first (or only) leader line should be displayed The Default is T

leader-2? *boolean*

Indicates whether the second leader line should be displayed The Default is T

leader-line-length *number*

Indicates the length of the first leader for the case when `outside-leaders?` is non-NIL

leader-line-length-2 *number*

Indicates the length of the second leader for the case when `outside-leaders?` is non-NIL

leader-text-gap *number*

Amount of gap between leader lines and dimension text, when the dimension text is within the leader Defaults to half the character-size

orientation *3x3 matrix of double-float numbers*

Indicates the absolute Rotation Matrix used to create the coordinate system of this object This matrix is given in absolute terms (i e with respect to the root's orientation), and is generally created with the alignment function It should be an *orthonormal* matrix, meaning each row is a vector with a magnitude of one (1 0)

outline-shape-type *keyword symbol*

Currently can be :bubble, :rectangle, or :none Default is :none

outside-leaders-length-factor *number*

Indicates the default length of the outside-leaders as a multiple of arrowhead-length Defaults to 3

outside-leaders? *boolean*

Indicates whether the leader line(s) should be inside or outside the interval between the start and end points The default is NIL, which indicates that the leader line(s) should be inside the interval

text-above-leader? *boolean*

Indicates whether the text is to the right or above the leader line, rather than in-line with it Default is T

text-along-axis? *boolean*

Where applicable, determines whether text direction follows leader-line direction

text-x-scale *number in percentage*

Adjusts the character width for the dimension-text and currently only applies only to PDF output

underline? *gdl*

witness-line-2? *boolean*

Indicates whether to display a witness line coming off the `end-point` Default is T

witness-line-ext *number*

Distance the witness line(s) extend beyond the leader line Default is 0 3

witness-line-gap *number*

Distance from the `start-point` and `end-point` to the start of each witness-line Default is 0 1

witness-line-length *number*

Length of the witness lines (or of the shorter witness line in case they are different lengths)

witness-line? *boolean*

Indicates whether to display a witness line coming off the `start-point` Default is T

Defaulted-input-slots

view-reference-object *gdl object or nil*

> View object which will use this dimension Defaults to NIL

7.0.3.24 Parallel-dimension

Description

> Creates a dimension annotation along an axis from a start point to an end point.

Computed-slots

base-plane-normal *must be specified in the subclass except for angular*

dim-text-start *3d point*

> Determines where the text will start Defaults to reasonable location for horizontal-dimension

leader-direction-1-vector *must be specified in the subclass except for angular*

leader-direction-2-vector *must be specified in the subclass except for angular*

witness-direction-vector *must be specified in the subclass except for angular*

Examples

```
(in-package :gdl-user)

(define-object  parallel-dimension-sample (base-object)


  :objects
  ((box :type 'box
        :length 10 :width (* (the-child length) +phi+)
        :height (* (the-child :width) +phi+))

   (length-dimension :type 'parallel-dimension
                     :character-size (/ (the box length) 20)
                     :start-point (the box (vertex :top :left :front))
                     :end-point (the box (vertex :top :right :rear)))))

  (generate-sample-drawing :object-roots (make-object 'parallel-dimension-sample))
```
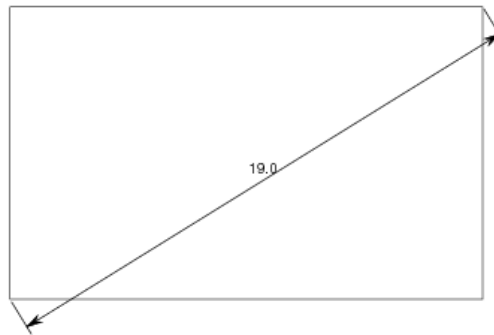
7.0.3.25 Pie-chart

Description

Generates a standard Pie Chart with colored filled pie sections.

This object was inspired by the pie-chart in Marc Battyani's (marc.battyani(at)fractalconcept.com) █ cl-pdf, with contributions from Carlos Ungil (Carlos.Ungil(at)cern.ch).

Optional-input-slots

data *list of numbers*

The relative size for each pie piece These will be normalized to percentages Defaults to NIL, must be specified as non-NIL to get a result

include-legend? *boolean*

Determines whether the Legend is included in standard output formats Defaults to `t`

labels&colors *list of lists, each containing a string and a keyword symbol*

This list should be the same length as `data` These colors and labels will be assigned to each pie piece and to the legend Defaults to NIL, must be specified as non-NIL to get a result

line-color *keyword symbol naming color from <tt>*color-table*</tt>*

Color of the outline of the pie Defaults to :black

radius *number*

The radius of the pie Defaults to 0 35 times the `width`

title *string*

Title for the chart Defaults to the empty string

title-color *keyword symbol naming color from <tt>*color-table*</tt>*

Color of title text Defaults to :black

title-font *string*

Currently this must be a PDF font name Defaults to "Helvetica "

title-font-size *number*

Size in points of title font Defaults to 12

Examples

```
(in-package :gdl-user)

(define-object pie-sample (pie-chart)
  :computed-slots
  ((data (list 30 70))

   (labels&colors '(("Expenses" :red) ("Revenue" :green)))

   (width 200)

   (title "Cash Flow")))

(generate-sample-drawing :objects (make-object 'pie-sample))
```
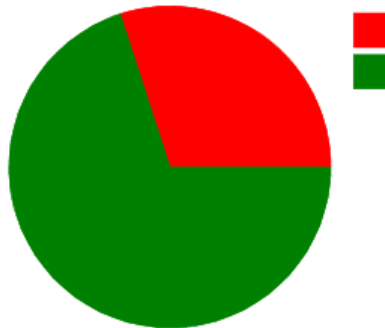

7.0.3.26 Point

Description

Visual representation of a point as a small view-independent crosshair. This means the crosshair will always appear in a "top" view regardless of the current view transform. The crosshair will not scale along with any zoom state unless the `scale?` optional input-

slot is non-NIL. The default color for the crosshairs is a light grey (:grey-light-very in the *color-table*).

Optional-input-slots

crosshair-length *number*

Distance from center to end of crosshairs used to show the point Default value is 3

radius *number*

Distance from center to any point on the sphere

scaled? *boolean*

Indicates whether the crosshairs drawn to represent the point are scaled along with any zoom factor applied to the display, or are fixed with respect to drawing space The default is NIL, meaning the crosshairs will remain the same size regardless of zoom state

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

Examples

```
(in-package :gdl-user)

(define-object point-sample (base-object)

  :objects
  ((bezier :type 'bezier-curve
           :control-points (list (make-point 0 0 0)
                                 (make-point 1 1 0)
                                 (make-point 2 1 0)
                                 (make-point 3 0 0)))
   (points-to-show :type 'point
                   :sequence (:size (length (the bezier control-points)))
                   :center (nth (the-child :index)
                                (the bezier control-points))
                   :radius 0.08
                   :display-controls (list :color :blue))))

(generate-sample-drawing :object-roots (make-object 'point-sample))
```

7.0.3.27 Route-pipe

Description
   Defines an alternating set of cylinders and torus sections for the elbows

Required-input-slots

outer-pipe-radius *number*
   Radius to the outer surface of the piping

vertex-list *list of 3d points*
   Same as for global-filleted-polyline (which is mixed in to this part)

Optional-input-slots

inner-pipe-radius *number*
   Radius of the inner hollow part of the piping NIL for a solid pipe

Computed-slots

bounding-box *list of two 3d points*
   The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

height *number*
   Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

orientation *3x3 matrix of double-float numbers*

Indicates the absolute Rotation Matrix used to create the coordinate system of this object This matrix is given in absolute terms (i e with respect to the root's orientation), and is generally created with the alignment function It should be an *orthonormal* matrix, meaning each row is a vector with a magnitude of one (1 0)

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object route-pipe-sample (base-object)

 :objects

 ((pipe :type 'route-pipe
        :vertex-list (list #(410.36 436.12 664.68)
                           #(404.21 436.12 734.97)
                           #(402.22 397.48 757.72)
                           #(407.24 397.48 801.12)
                           #(407.24 448.0 837.0)
                           #(346.76 448.0 837.0))
        :default-radius 19
        :outer-pipe-radius 7
        :inner-pipe-radius nil
        :display-controls (list :color :blue-steel
                                :transparency 0.0
                                :shininess 0.7
                                :spectral-color :white))))


   (generate-sample-drawing :objects (the-object (make-object 'route-pipe-sample) pipe)
                            :projection-direction (getf *standard-views* :trimetric))
```
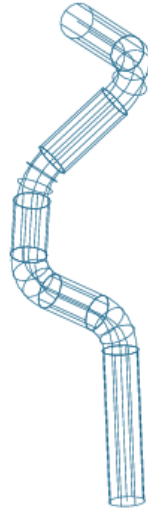
7.0.3.28 Sample-drawing

Description

Defines a simple drawing with a single view for displaying objects or object-roots.

Optional-input-slots

page-length *number in pdf points*

Front-to-back (or top-to-bottom) length of the paper being represented by this drawing
The default is (* 11 72) points, or 11 inches, corresponding to US standard letter-size paper

page-width *number in pdf points*

Left-to-right width of the paper being represented by this drawing The default is (* 8 5
72) points, or 8 5 inches, corresponding to US standard letter-size paper

7.0.3.29 Sphere

Description

The set of points equidistant from a given center point.

Required-input-slots

radius *number*

Distance from center to any point on the sphere

Optional-input-slots

number-of-horizontal-sections *number*

How many lines of latitude to show on the sphere in some renderings Default value is 4

number-of-vertical-sections *number*

How many lines of longitude to show on the sphere in some renderings Default value is 4

Defaulted-input-slots

end-horizontal-arc *angle in radians*

Ending horizontal angle for a partial sphere Default is twice pi

end-vertical-arc *angle in radians*

Ending vertical angle for a partial sphere Default is pi/2

inner-radius *number*

Radius of inner hollow for a hollow sphere Default is NIL, for a non-hollow sphere

start-horizontal-arc *angle in radians*

Starting horizontal angle for a partial sphere Default is 0

start-vertical-arc *angle in radians*

Starting vertical angle for a partial sphere Default is -pi/2

Computed-slots

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object sphere-sample (sphere)

 :computed-slots

  ((radius 150)
   (number-of-vertical-sections 10)
   (number-of-horizontal-sections 10)
   (display-controls (list :color :green-forest-medium))))
```

```
(generate-sample-drawing :objects (make-object 'sphere-sample)
                         :projection-direction :trimetric)
```
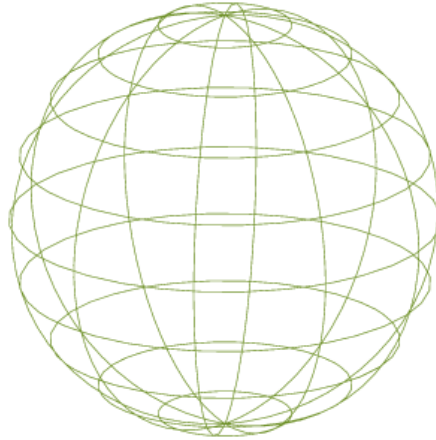
7.0.3.30 Spherical-cap

Description

The region of a sphere which lies above (or below) a given plane. Although this could be created with a partial sphere using the sphere primitive, the spherical cap allows for more convenient construction and positioning since the actual center of the spherical cap is the center of its reference box.

Required-input-slots

axis-length *number*

The distance from the center of the base to the center of the dome

base-radius *number*

Radius of the base

Optional-input-slots

number-of-horizontal-sections *integer*

How many lines of latitude to show on the spherical-cap in some renderings Default value is 2

number-of-vertical-sections *integer*

How many lines of longitude to show on the spherical-cap in some renderings Default value is 2

Defaulted-input-slots

cap-thickness *number*

Thickness of the shell for a hollow spherical-cap Specify this or inner-base-radius, not both

inner-base-radius *number*

Radius of base of inner for a hollow spherical-cap Specify this or cap-thickness, not both

Computed-slots

end-angle *angle in radians*

End angle of the arc Defaults to twice pi

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

sphere-center *3d point*

Center of the sphere containing the spherical-cap

sphere-radius *number*

Radius of the sphere containing the spherical-cap

start-angle *angle in radians*

Start angle of the arc Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object spherical-cap-sample (spherical-cap)

 :computed-slots
```

```
((base-radius 150)
 (cap-thickness 7)
 (axis-length (* (the base-radius) +phi+))
 (number-of-vertical-sections 10)
 (number-of-horizontal-sections 10)
 (display-controls (list :color :orchid-medium :transparency 0.5))))


(generate-sample-drawing :objects (make-object 'spherical-cap-sample)
                         :projection-direction :trimetric)
```
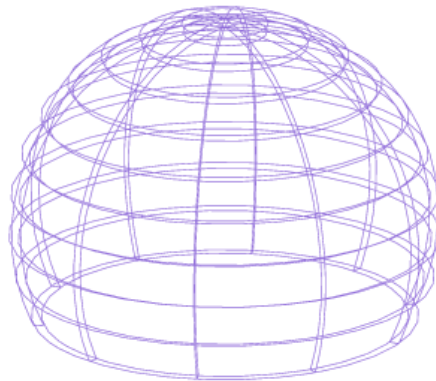
7.0.3.31 Text-line

Description

!!! Not applicable for this object !!!

Optional-input-slots

center *3d-point*

Center of the text Specify this or start, not both

start *3d-point*

Start of the text Specify this or center, not both

width *number*

X-axis dimension of the reference box Defaults to zero

Computed-slots

length *number*

    Y-axis dimension of the reference box Defaults to zero

7.0.3.32 Torus

Description

    A single-holed "ring" torus, also known as an "anchor ring." This is basically a circular cylinder "bent" into a donut shape. Partial donuts ("elbows") are supported. Partial "bent" cylinders are not currently supported.

Required-input-slots

major-radius *number*

    Distance from center of donut hole to centerline of the torus

minor-radius *number*

    Radius of the bent cylinder making up the torus

Optional-input-slots

draw-centerline-arc? *boolean*

    Indicates whether the bent cylinder's centerline arc should be rendered in some renderings

end-caps? *boolean*

    Indicates whether to include end caps for a partial torus in some renderings Defaults to T

number-of-longitudinal-sections *integer*

    Indicates the number of arcs to be drawn on along "surface" of the torus in some wireframe renderings

number-of-transverse-sections *integer*

    Indicates the number of circular cross-sections of the bent cylinder to show in some wireframe renderings

Defaulted-input-slots

arc *angle in radians*

    Indicates the end angle for the donut Defaults to twice pi for a full-circle donut

inner-minor-radius *number*

Radius of the inner hollow part of the bent cylinder for a hollow torus Defaults to NIL for a solid cylinder

Computed-slots

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(in-package :gdl-user)

(define-object torus-sample (torus)
  :computed-slots
  ((major-radius 150)
   (minor-radius 42)
   (draw-centerline-arc? t)
   (number-of-longitudinal-sections 10)
   (number-of-transverse-sections 10)
   (display-controls (list :color :green-forest-medium)))

  :hidden-objects ((view :type 'base-view
                         :projection-vector (getf *standard-views* :trimetric)
                         :page-width (* 5 72) :page-length (* 5 72)
                         :objects (list self))))


(generate-sample-drawing :objects (make-object 'torus-sample)
                         :projection-direction :trimetric)
```
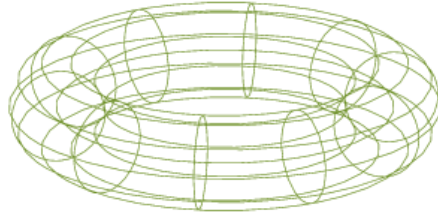
7.0.3.33 Typeset-block

Description

Block of text typeset using cl-typesetting. This object wraps the typeset block as a standard GDL object, so it can be placed in a view and positioned according to normal GDL positioning.

You can specify the width, and by default this object will compute its length automatically from the typeset content, to fit all the lines of text into the box. Because of this computed behavior of the length, the center of the box will not, in general, be in a known location compared to the start of the text. Because of this it is recommended to use :corner, rather than :center, for positioning a base-view which contains a typeset block.

In the normal case, if you want a single block in a view on a drawing, you should make the base-view object have the same width and length as the typeset-block. The base-view should also probably have :left-margin 0 and :front-margin 0.

Optional-input-slots

center *3d-point*

Center of the text Specify this or start, not both NOTE that the center is no longer defaulting (so that it can self-compute properly when start is specified), so it is necessary to explicitly give either start or center for general-note

length *number*

The length of the box to contain the compiled content Defaults is (the length-default), which will exactly fit the compiled content into the specified width If you override it to be less than this default, the content will be cropped

start *3d-point*

Start of the text Specify this or center, not both

start-line-index *number*

    The line number to start

Computed-slots

length-default *number*

    The computed length which will exactly fit the content based on (the width)

lines *list of typeset line objects*

    The list of lines in the nominal block

7.0.3.34 Vertical-dimension

Description

    Creates a dimension annotation along the vertical axis.

Optional-input-slots

dim-text-start *3d point*

    Determines where the text will start Defaults to reasonable location for horizontal-dimension

Computed-slots

base-plane-normal *must be specified in the subclass except for angular*

leader-direction-1-vector *must be specified in the subclass except for angular*

leader-direction-2-vector *must be specified in the subclass except for angular*

witness-direction-vector *must be specified in the subclass except for angular*

Examples

```
(in-package :gdl-user)

(define-object  vertical-dimension-sample (base-object)


  :objects
  ((box :type 'box
        :length 10 :width (* (the-child length) +phi+)
        :height (* (the-child :width) +phi+))

   (length-dimension :type 'vertical-dimension
```

```
                              :character-size (/ (the box length) 20)
                              :flip-leaders? t
                              :start-point (the box (vertex :top :left :front))
                              :end-point (the box (vertex :top :left :rear)))))

  (generate-sample-drawing :object-roots (make-object 'vertical-dimension-sample))
```
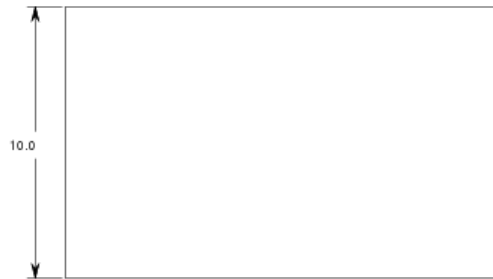

7.0.4 Surf

7.0.4.1 Approximated-curve

Description

This primitive accepts a NURBS curve and computes a new NURBS curve with presumably fewer control points, and claims to hold it to within a certain tolerance of the original curve.

The point at the start parameter and the end parameter in the result curve will be fixed to be identical to the original curve.

You can use the pinned-parameters input-slot to specify additional parameter values where the new curve will be pinned to be identical with the original curve.

Required-input-slots

curve-in *gdl curve object*

The curve to be approximated with this curve

Optional-input-slots

match-parameterization? *boolean*

Indicates whether the new-curve should contain matching parameterization between breakpoints If nil, the parameterization can slip a little, but not very much Allowing the parameterization between breakpoints to slip results in somewhat fewer control points Use non-nil only if a parameter-to-parameter match is important to your application The default is nil

pinned-parameters *list of numbers*

These are parameter values from the original curve where the approximated curve will be forced to be identical to the original curve Defaults to nil

tolerance *number*

The maximum distance deviation from the curve-in to this curve Defaults to 1 0e-5 times the diagonal of the bounding box of the input curve

Computed-slots

achieved-tolerance *number*

This should reflect the actual tolerance achieved with the approximation In the case of smlib, it is not clear exactly what this value is supposed to mean – from examples we have seen so far, the value appears to range between 0 0 and 1 0, and it appears that values close to 1 0 indicate a close approximation, and values close to 0 0 indicate a loose approximation

Examples

```
(define-object approximated-curve-test (base-object)

 :input-slots
 ((sample-b-spline-data
  '((#(-7.773502691896258 10.0 0.0)
#(-7.76304131662674 10.0 0.0035356993309923)
#(-7.746775287947699 10.0 0.0067001904580447)
#(-7.7253289934578415 10.0 0.0091817670732663)
#(-7.709372706886673 10.0 0.0109178383762297)
#(-7.693497618275636 10.0 0.0129741578942581)
#(-7.676981089448407 10.0 0.0152725944452429)
#(-7.660626985913543 10.0 0.0176076438762828)
#(-7.644400043044812 10.0 0.0198301741704749)
#(-7.628572854579257 10.0 0.0218811393098875)
#(-7.612456095337886 10.0 0.023892087537815)
#(-7.596123157036033 10.0 0.025876731740358)
#(-7.5797886014351645 10.0 0.0278392394943197)
#(-7.563598247718005 10.0 0.0297977372116626)
#(-7.54734474235553 10.0 0.0317827355165938)
#(-7.530817428296935 10.0 0.0338094455691551)
#(-7.514499984995836 10.0 0.0358111584598995)
#(-7.498530032892328 10.0 0.037762615650263)
#(-7.482560694806967 10.0 0.0396996233035483)
#(-7.466938587271525 10.0 0.0415717500361984)
#(-7.45613073401155 10.0 0.0428492028474684)
#(-7.445702016561547 10.0 0.04407409984323)
#(-7.439862498458236 10.0 0.0447555129227675)
#(-7.429535001467346 10.0 0.0459531655261334)
```

```
#(-7.423946015929491 10.0 0.0465975377728551)
#(-7.413554192167366 10.0 0.0477892161812883)
#(-7.407595596706495 10.0 0.0484686195293404)
#(-7.396934499325897 10.0 0.049677834187285)
#(-7.391025013196135 10.0 0.0503446277462212)
#(-7.380668397266659 10.0 0.0515077909624895)
#(-7.374959869886305 10.0 0.0521461185088044)
#(-7.36464849713204 10.0 0.05329441651439)
#(-7.3587350147730115 10.0 0.0539501551169936)
#(-7.348489965069932 10.0 0.0550810605426388)
#(-7.342793963594458 10.0 0.0557070781305422)
#(-7.332490622079339 10.0 0.0568343892437064)
#(-7.3264770440672065 10.0 0.0574891649794912)
#(-7.316219293859572 10.0 0.0586004349229853)
#(-7.310517864890045 10.0 0.0592151399809975)
#(-7.300540086253307 10.0 0.0602855343476251)
#(-7.294748391590457 10.0 0.0609037603868834)
#(-7.284555892216601 10.0 0.061985955452512)
#(-7.278577998313509 10.0 0.062617198511794)
#(-7.268114585011737 10.0 0.0637157114288265)
#(-7.261986919319522 10.0 0.0643552477004073)
#(-7.2514547970306005 10.0 0.0654480650996109)
#(-7.245363578736502 10.0 0.066076354340049)
#(-7.235168395459871 10.0 0.0671217900147562)
#(-7.2293285386923625 10.0 0.0677172614089348)
#(-7.21945679793601 10.0 0.0687181850892115)
#(-7.21363748676987 10.0 0.0693049508093313)
#(-7.203721331045569 10.0 0.0702991235190865)
#(-7.197774951476638 10.0 0.0708918760373581)
#(-7.18753366922008 10.0 0.071906820780931)
#(-7.181328324859931 10.0 0.0725180648535886)
#(-7.170908624453838 10.0 0.0735382495023682)
#(-7.164741045570452 10.0 0.0741384240225961)
#(-7.154470299979475 10.0 0.0751317627718068)
#(-7.1483716586933825 10.0 0.0757179834474372)
#(-7.138406833936611 10.0 0.0766699111117332)
#(-7.132501722803377 10.0 0.0772306457809111)
#(-7.12244815794467 10.0 0.0781794021319588)
#(-7.116194798062766 10.0 0.0787657100109435)
#(-7.105880729941813 10.0 0.0797262397246034)
#(-7.099656569353997 10.0 0.0803020171513102)
#(-7.089330480447966 10.0 0.0812507479304588)
#(-7.083006810743995 10.0 0.0818277014224611)
#(-7.0730076945596645 10.0 0.0827335903150743)
#(-7.067069212658176 10.0 0.0832680013237252)
#(-7.057382524393892 10.0 0.0841338221164179)
#(-7.0513228782074515 10.0 0.0846717275930971)
#(-7.041364101581027 10.0 0.0855495109968372)
#(-7.035106540724323 10.0 0.0860970019877134)
#(-7.024857742651612 10.0 0.0869868797241497)
#(-7.018438815809276 10.0 0.087539916228507)
#(-7.008168401811841 10.0 0.0884178150543803)
#(-7.00184087536049 10.0 0.0889544270356334)
#(-6.9919619641004545 10.0 0.0897854973995385)
#(-6.985881827579846 10.0 0.0902930555610231)
#(-6.976281029598491 10.0 0.0910882693796495)
#(-6.970184675056803 10.0 0.091589260852436)
#(-6.960320705968006 10.0 0.0923932980242485)
```

```
#(-6.953917746465384 10.0 0.0929107533519837)
#(-6.943757762558306 10.0 0.0937245181023455)
#(-6.937303992646001 10.0 0.0942367919470313)
#(-6.9271520790382235 10.0 0.0950351703452132)
#(-6.920707811824021 10.0 0.0955372165209965)
#(-6.910885266876965 10.0 0.0962951353561948)
#(-6.904716963255798 10.0 0.096766697374172)
#(-6.895200295978004 10.0 0.0974873900684733)
#(-6.889018398304428 10.0 0.0979511004533426)
#(-6.8792396786986085 10.0 0.0986773910779738)
#(-6.8727507213 10.0 0.0991543261574633)
#(-6.86267690754424 10.0 0.0998867125969024)
#(-6.856136640740504 10.0 0.100356993136921)
#(-6.846050345447107 10.0 0.1010740145138523)
#(-6.839489198598122 10.0 0.1015350665777411)
#(-6.829441747557248 10.0 0.1022327061061873)
#(-6.82289308331905 10.0 0.1026819075455008)
#(-6.813152437915218 10.0 0.1033417487088309)
#(-6.806847948189532 10.0 0.103763701703458)
#(-6.7971454958195565 10.0 0.1044050485135838)
#(-6.790595307231813 10.0 0.10483231339683)
#(-6.780922064334754 10.0 0.1054546472054939)
#(-6.77458575320461 10.0 0.1058569326392236)
#(-6.7649029807976975 10.0 0.1064632879605449)
#(-6.758284636701557 10.0 0.1068717205935851)
#(-6.748340665523174 10.0 0.1074760046701189)
#(-6.741671570163981 10.0 0.1078750063466806)
#(-6.731717886166939 10.0 0.1084609002957398)
#(-6.7250260196879 10.0 0.1088483306291875)
#(-6.715132250830737 10.0 0.1094113701746257)
#(-6.708479289674014 10.0 0.1097833660497434)
#(-6.698863733855831 10.0 0.1103113585573882)
#(-6.692407555733223 10.0 0.1106596095578191)
#(-6.683112518678094 10.0 0.1111518410008553)
#(-6.67673768062156 10.0 0.1114832653763077)
#(-6.667240402771452 10.0 0.1119675047371582)
#(-6.660514775321519 10.0 0.1123033129775543)
#(-6.650681029972996 10.0 0.1127835983503758)
#(-6.643918945899163 10.0 0.1131064056804846)
#(-6.63405272844892 10.0 0.1135663439198465)
#(-6.627254408957563 10.0 0.1138754999668116)
#(-6.617707456082231 10.0 0.1142985673023449)
#(-6.611225484562257 10.0 0.1145786891314138)
#(-6.60204805594244 10.0 0.1149649311892426)
#(-6.595561879720703 10.0 0.1152306714612809)
#(-6.586146787784009 10.0 0.1156055153436167)
#(-6.579359503332465 10.0 0.1158674765410608)
#(-6.569607126249581 10.0 0.1162316741844863)
#(-6.562716518300558 10.0 0.1164802471836087)
#(-6.552941578381875 10.0 0.1168201535727847)
#(-6.546090216293524 10.0 0.1170494074887217)
#(-6.536674084104508 10.0 0.117351835565929)
#(-6.530101252192292 10.0 0.1175545076386553)
#(-6.521003931182706 10.0 0.1178230920419217)
#(-6.514431925615438 10.0 0.1180084927017307)
#(-6.505237782335364 10.0 0.1182554027074824)
#(-6.498513444650254 10.0 0.1184266239748961)
#(-6.488983008492481 10.0 0.1186555956582973)
```

```
#(-6.48200268637703 10.0 0.1188128917257091)
#(-6.472326149681036 10.0 0.119016053562057)
#(-6.465389207391072 10.0 0.1191511200864016)
#(-6.455601236135456 10.0 0.1193262289159263)
#(-6.448443729690395 10.0 0.1194424780868816)
#(-6.438958198675441 10.0 0.1195804860871952)
#(-6.432273317606395 10.0 0.1196673012040853)
#(-6.422976238615972 10.0 0.1197731501514096)
#(-6.4159621089377605 10.0 0.1198409999468651)
#(-6.4067002169151355 10.0 0.1199142428159179)
#(-6.399997171744239 10.0 0.119556098103827)
#(-6.390775586474153 10.0 0.1199979664750528)
#(-6.383766014520722 10.0 0.1200169608096984)
#(-6.374398451275458 10.0 0.1200248790095977)
#(-6.367508232676765 10.0 0.1200180617465295)
#(-6.358204641636406 10.0 0.1199913666687163)
#(-6.351212499117087 10.0 0.1199579262946732)
#(-6.34179811653411 10.0 0.1198940830107079)
#(-6.334722008984675 10.0 0.1198318598583616)
#(-6.325243506895607 10.0 0.1197286664953724)
#(-6.3181038668924225 10.0 0.1196359024164523)
#(-6.3086233644061815 10.0 0.1194922029896446)
#(-6.301482992162023 10.0 0.1193683670372235)
#(-6.2919924776063425 10.0 0.1191825584163038)
#(-6.284793044978927 10.0 0.1190252258627906)
#(-6.2753314970647045 10.0 0.1187965753264672)
#(-6.26821135846841 10.0 0.1186078391724031)
#(-6.259162952769228 10.0 0.1183465300397597)
#(-6.252357920075809 10.0 0.1181346604145395)
#(-6.243732250120057 10.0 0.1178464349390531)
#(-6.237024615970206 10.0 0.117607149463944)
#(-6.22847504378218 10.0 0.1172819358381489)
#(-6.221661738722191 10.0 0.1170066698163893)
#(-6.212734929396275 10.0 0.1166237921552943)
#(-6.205560407960056 10.0 0.1162969301785835)
#(-6.196251975857688 10.0 0.1158468244886666)
#(-6.188964185271005 10.0 0.1154735323164374)
#(-6.179555494737541 10.0 0.1149639153426461)
#(-6.172229381962318 10.0 0.1145450494936365)
#(-6.162856146944508 10.0 0.113980051461802)
#(-6.155565515353094 10.0 0.1135177665315744)
#(-6.146209923750142 10.0 0.1128942970314755)
#(-6.138855137820279 10.0 0.112380030061237)
#(-6.129655628386858 10.0 0.1117056172410459)
#(-6.122486612998066 10.0 0.111156182233171)
#(-6.11363265472315 10.0 0.110447376638818)
#(-6.10657159410418 10.0 0.1098584591558712)
#(-6.097764739903218 10.0 0.1090932742430056)
#(-6.0906136679603495 10.0 0.1084460244904774)
#(-6.081792237411942 10.0 0.1076133101575569)
#(-6.074566622637572 10.0 0.106911620109585)
#(-6.065558573551578 10.0 0.1059791445416934)
#(-6.058094902257685 10.0 0.105180297224725)
#(-6.04902211746673 10.0 0.1041654080838163)
#(-6.041837834442202 10.0 0.1033285781165379)
#(-6.032806538284214 10.0 0.102232676388748)
#(-6.025326943718607 10.0 0.1012858590168515)
#(-6.016324551464302 10.0 0.1000962619743535)
```

```
                 #(-6.009099370537937 10.0 0.0991026392362128)
                 #(-6.000144802087108 10.0 0.0978213410260762)
                 #(-5.992672802847265 10.0 0.0967077356358815)
                 #(-5.983682684099936 10.0 0.095311303764513)
                 #(-5.976390894901862 10.0 0.0941327969909632)
                 #(-5.967597701948485 10.0 0.092651171023629)
                 #(-5.960300625393035 10.0 0.091369004729541)
                 #(-5.95181904289547510.0 0.0898129037222759)
                 #(-5.944827122197059 10.0 0.0884764825079818)
                 #(-5.9364937900430546 10.0 0.086815333082949)
                 #(-5.929316123058646 10.0 0.0853215712223041)
                 #(-5.920853977349434 10.0 0.0834809963775036)
                 #(-5.913697488126982 10.0 0.081855290446912)
                 #(-5.905045875020715 10.0 0.0797946852900259)
                 #(-5.897633195865675 10.0 0.077937627548088)
                 #(-5.888869222952547 10.0 0.0756214894583634)
                 #(-5.881576422780188 10.0 0.0735895003825018)
                 #(-5.8730488606276685 10.0 0.071078664456186)
                 #(-5.865886603820401 10.0 0.0688505661471062)
                 #(-5.857659076163453 10.0 0.0661282191187038)
                 #(-5.8506956581380045 10.0 0.0636758008622684)
                 #(-5.8427993520796395 10.0 0.0607060179365281)
                 #(-5.836006841543422 10.0 0.0579805075953104)
                 #(-5.828141782273885 10.0 0.0545732138995477)
                 #(-5.821264105869612 10.0 0.0513305536611062)
                 #(-5.813396489418997 10.0 0.0472425696093425)
                 #(-5.806736741014998 10.0 0.0434068855227238)
                 #(-5.799296800365852 10.0 0.0385347465920627)
                 #(-5.793149838865298 10.0 0.0339017049050897)
                 #(-5.786639267262249 10.0 0.02799360148602)
                 #(-5.781555952133565 10.0 0.0222871988704581)
                 #(-5.77690049447455 10.0 0.0149171833984971)
                 #(-5.774143850809992 10.0 0.0076254582681704)
                 #(-5.773504871989952 10.0 0.0021812762824132)
                 #(-5.77350372489622 10.0 -5.432357e-10))
 (1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
  1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)
 (0.0 0.0 0.0 0.0 0.0160453781542143 0.0241470160269717
  0.0317935799490915 0.0398053040025856 0.0478086834801164
  0.0560582201696031 0.0639202701209087 0.0718503545073776
  0.0797433850046513 0.087981750276748 0.0960087041824917
  0.1039831218391125 0.112029495827222 0.1201752268087684
  0.1282605498138822 0.136005192477863 0.1438242190390603
  0.1519870644337011 0.1593306835295498 0.1598487740853562
  0.1674658650196929 0.1680104087509454 0.175157710986505
  0.1757272662987668 0.1833705173786548 0.183966778279312
```

```
0.1915177150798282   0.1921404682290536   0.1993060375256694
0.1999541159685396   0.2074081735084615   0.2080825976726782
0.2151502427534925   0.2158498418097879   0.2234099696672682
0.2241364268834526   0.2312069092585329   0.2319587197887401
0.2390267110021689   0.239803949187276    0.2470314540979864
0.2478347213119199   0.2551668057971311   0.2559965267399797
0.2633214412146924   0.2641776785913368   0.2711080693997827
0.271989626560843    0.2788004837617649   0.2797070543520855
0.2865857660092436   0.2875176520073938   0.294691617040477
0.2956498608401993   0.3028107414290341   0.303795386191314
0.3108677223812281   0.3118785660342093   0.3186289517157898
0.3196650325637266   0.3267875123448071   0.3278501223901527
0.3348376542335923   0.3359264409310518   0.343017473844114
0.3441328588668275   0.3507020846059745   0.3518424576829762
0.3585158400916831   0.3596816211629636   0.366600003006972
0.3677920713565625   0.3748151183901361   0.3760338998362696
0.3829150808161275   0.3841602009158513   0.3906373023990221
0.3919075328521676   0.3983652317385465   0.3996605911050301
0.406480133489536    0.4078018800878236   0.4145992363495962
0.4159473838404371   0.4227235238201337   0.4240980890623294
0.4304686564547328   0.4318684065502333   0.4382164439562757
0.4396413875379139   0.4463489290384909   0.4478003170276637
0.4544823491094193   0.4559601845464303   0.4626166004905004
0.4641208860785213   0.470751784887656    0.4722825236605759
0.4785036764886068   0.4800596220929673   0.486633503837215
0.4882158852069559   0.4943709597308301   0.4959785009914117
0.502479159396033    0.5041130660952415   0.5105726294674962
0.512232853709101    0.5186483892971976   0.5203348734927932
0.5267087790327916   0.5284214732034318   0.5344904142923997
0.5362284120119565   0.5421342182310729   0.5438970713133418
0.5502033260482438   0.5519924174541494   0.5583120642198542
0.5601275228154484   0.5665206241752909   0.5683627745511391
0.5742736086012711   0.5761409693620999   0.5819524448520371
0.5838447748901722   0.5899961210140513   0.591914606679692
0.5981367389758905   0.6000816954947563   0.6062612328823314
0.6082326078238199   0.6139891743550618   0.6159856782493418
0.6217020223331746   0.623723606100719    0.6295913193586837
0.6316385567589856   0.6377653030384736   0.6398391197873293
0.6458190701143912   0.6479190753033844   0.6541566170946882
0.6562837334911363   0.6618008400206912   0.6639528131422728
0.6699223065860094   0.6721006882862063   0.6775449581602477
0.6797481264419983   0.6856437214610128   0.6878732244971574
0.6935814940177247   0.695836808314546    0.7016366241440788
0.7039181313132697   0.709705856689687    0.7120136025880942
0.7177645246694223   0.7200984749442396   0.7258142875643973
0.7281744132589539   0.7339554983783031   0.7363420968538624
0.7420859180388898   0.7444989542058508   0.7497996761741695
0.7522377951739535   0.7574084721926538   0.7598713327198711
0.7650117024519579   0.7674992864084252   0.7730174533892418
0.7755310696517345   0.7811075108508807   0.7836474335589837
0.7892724115280283   0.7918388840494792   0.7974138106069117
0.8000067565215466   0.8056169748708625   0.8082365950205777
0.8135658669535445   0.816211334521307    0.8213255105466555
0.8239962101530169   0.829242574214828    0.8319390177427586
0.8370829503324179   0.8398048884169548   0.8454152884202761
0.8481643207745085   0.8532523057058028   0.856026821694725
0.8614897782714848   0.8642910800558162   0.8693003110837616
0.8721270103832023   0.8774885682570619   0.8803418933181856
```

```
     0.8854532218703798 0.8883324456013311 0.8934528695387036
     0.8963581057296763 0.9010534532900417 0.9039834043046125
     0.9089319065807004 0.9118874759674483 0.9167609857678787
     0.9197420129769963 0.9249624084248843 0.9279701042058975
     0.9330316439657489 0.9360655784857183 0.9410404196288085
     0.9441003962903061 0.9488799412961575 0.9519654097358451
     0.9565785237508843 0.9596890256766235 0.9645868244983244
     0.9677233670213087 0.9726644785371455 0.9758272871735215
     0.9807483303509975 0.9839374252540747 0.9887471809818733
     0.9919622857532446 0.996778435740851 1.0 1.0 1.0 1.0)
    3)))

      :objects ((profile :type 'b-spline-curve
                          :control-points (first (the sample-b-spline-data))
                          :weights (second (the sample-b-spline-data))
                          :knot-vector (third (the sample-b-spline-data))
                          :degree (fourth (the sample-b-spline-data)))


               (approximate :type 'approximated-curve
                            :display-controls (list :color :red)
                            :tolerance 0.01
                            :pinned-parameters (list 0.1 0.5 0.7 0.9)
                            :match-parameterization? t
                            :curve-in (the profile))))

    (generate-sample-drawing :object-roots (make-object 'approximated-curve-test)
                             :projection-direction (getf *standard-views* :front)
                             :page-width 1000)
```

7.0.4.2 Arc-curve

Description

An arc represented exactly as a quadratic NURBS curve. Inputs are the same as for arc. Messages are the union of those for arc and those for curve.

Examples

```
(in-package :surf)

(define-object test-arc-curve (arc-curve)
  :computed-slots
  ((center (make-point 0 0 0)) (radius 10) (start-angle 0) (end-angle 2pi)))

(generate-sample-drawing :objects (make-object 'test-arc-curve))


(define-object test-arc-curve2 (arc-curve)
  :computed-slots
  ((center (make-point 0 0 0)) (radius 5) (start-angle (* 0.25 pi)) (end-angle pi)))

(generate-sample-drawing :objects (make-object 'test-arc-curve))
```

7.0.4.3 B-spline-curve

Description

A general NURBS (potentially non-Uniform, potentially Rational, b-spline) curve specified with control points, weights, knots, and degree.

If the knot-vector is different from the default, it is non-Uniform.

If any of the weights are different from 1.0, it is Rational.

Required-input-slots

control-points *list of 3d points*

The control points

Optional-input-slots

degree *integer*

Degree of the curve Defaults to 3 (cubic)

knot-vector *list of numbers*

Knots of the curve Default is NIL, which indicates a uniform knot vector

weights *list of numbers*

A weight to match each control point Should be same length as control-points Default is a value of 1 0 for each weight, resulting in a nonrational curve

Examples

```
(in-package :surf)

(define-object test-b-spline-curves (base-object)

 :input-slots
 ((control-points (list (make-point 0 0 0)
                        (make-point 2 3.0 0.0)
                        (make-point 4 2.0 0.0)
                        (make-point 5 0.0 0.0)
                        (make-point 4 -2.0 0.0)
                        (make-point 2 -3.0 0.0)
                        (make-point 0 0 0))))

 :objects
 ((curves :type 'b-spline-curve
          :sequence (:size 6)
          :control-points (the control-points)
          :degree (1+ (the-child :index))
          :display-controls (list :line-thickness (* 0.3 (the-child index))
                                  :color (ecase (the-child index)
                                           (0 :red) (1 :orange)
```

```
                                          (2 :yellow) (3 :green)
                                          (4 :blue) (5 :red-violet))))

   (points :type 'point
           :sequence (:size (length (rest (the control-points))))
           :center (nth (the-child index) (rest (the control-points)))
           :display-controls (list :color :green))))


(generate-sample-drawing :object-roots (make-object 'test-b-spline-curves))


;; Here is another example which shows the difference between a
;; simple bezier-curve from the :geom-base package, and a NURBS.
;;
(define-object bezier-and-nurbs (base-object)

  :input-slots ((control-points (list (make-point 0 0 0)
                                       (make-point 1 1 0)
                                       (make-point 2 1 0)
                                       (make-point 3 0 0))))


  :objects ((points :type 'points-display
                    :points (the control-points))


            (bezier :type 'bezier-curve
                    ;; This will be a geom-base:bezier-curve
                    :display-controls (list :color :green)
                    :control-points (the control-points))

            (b-spline :type 'b-spline-curve
                      ;; This will be an equivalent surf:b-spline-curve.
                      :display-controls (list :color :red :bezier-points t)
                      :control-points (the bezier control-points))

            ;;
            ;; The b-spline-curve is a full NURBS curve and so has
            ;; more inputs than a simple bezier-curve: degree,
            ;; weights, and knot-vector, so we can do things like:
            ;;
            (b-spline-weighted :type 'b-spline-curve
                               :display-controls (list :color :purple)
                               :control-points (the bezier control-points)
                               :weights (list 1.0 1.2 1.2 1.0))

            (b-spline-degree-2 :type 'b-spline-curve
                               :display-controls (list :color :orange)
                               :control-points (the bezier control-points)
                               :degree 2)))
```

7.0.4.4 B-spline-surface

Description

A general b-spline surface specified with control points, weights, knots, and degree.

Required-input-slots

control-points *list of lists of 3d points*

The control net

Optional-input-slots

u-degree *integer*

Degree of surface in U direction Defaults to 3

u-knot-vector *list of numbers*

Knots in U direction Default is NIL, which indicates a uniform knot vector in U direction

v-degree *integer*

Degree of surface in V direction Defaults to 3

v-knot-vector *list of numbers*

Knots in V direction Default is NIL, which indicates a uniform knot vector in V direction

weights *list of lists of numbers*

A weight to match each control point Should be congruent with control-points (i e same number of rows and columns) Default is a value of 1 0 for each weight, resulting in a nonrational surface

Examples

```
(in-package :surf)

(define-object test-b-spline-surface (b-spline-surface)

  :computed-slots
  ((points-data '(((0 0 0)(4 1 0)(8 1 0)(10 0 0)(8 -1 0)(4 -1 0)(0 0 0))
                  ((0 0 2) (4 2 2) (8 2 2) (10 0 2) (8 -2 2) (4 -2 2) (0 0 2))
                  ((0 0 4) (4 2 4) (8 2 4) (10 0 4) (8 -2 4) (4 -2 4) (0 0 4))
                  ((0 0 7) (4 1 7) (8 1 7) (10 0 7) (8 -1 7) (4 -1 7) (0 0 7))))
   (control-points (mapcar #'(lambda(list)
                                       (mapcar #'apply-make-point list))
                             (the points-data)))

   (display-controls (list :color :periwinkle :line-thickness 2))))

(generate-sample-drawing :object-roots (make-object 'test-b-spline-surface)
                         :projection-direction :trimetric)
```

7.0.4.5 Basic-surface

Description

   This routine constructs a 3D surface by interpolating a set of four boundary 3D curves.

Required-input-slots

curve-bottom *gdl curve object*

   The curve corresponding to the surface bottom boundary

curve-left *gdl curve object*

   The curve corresponding to the surface left boundary

curve-right *gdl curve object*

   The curve corresponding to the surface right boundary

curve-top *gdl curve object*

   The curve corresponding to the surface top boundary

Examples

```
(in-package :gdl-user)

(define-object test-basic-surface (base-object)

 :computed-slots
 ((control-points-u-min (list (make-point 0 0 0 )
                               (make-point 1 0 0 )
                               (make-point 2 0 1 )
                               (make-point 3 0 0 )
                               (make-point 4 0 0 )))

  (control-points-u-max (list (make-point 0 4 0 )
                               (make-point 1 4 0 )
                               (make-point 2 4 1 )
                               (make-point 3 4 0 )
                               (make-point 4 4 0 )))

  (control-points-v-min (list (make-point 0 0 0 )
                               (make-point 0 1 0 )
                               (make-point 0 2 1 )
                               (make-point 0 3 0 )
                               (make-point 0 4 0 )))

  (control-points-v-max (list (make-point 4 0 0 )
                               (make-point 4 1 0 )
                               (make-point 4 2 1 )
                               (make-point 4 3 0 )
                               (make-point 4 4 0 ))))

  :objects
```

```
((surf-curve-u-min :type 'fitted-curve
                   :display-controls (list :color :green :line-thickness 2)
                   :points (the control-points-u-min))

 (surf-curve-u-max :type 'fitted-curve
                   :display-controls (list :color :green :line-thickness 2)
                   :points (the control-points-u-max))

 (surf-curve-v-min :type 'fitted-curve
                   :display-controls (list :color :blue :line-thickness 2)
                   :points (the control-points-v-min))

 (surf-curve-v-max :type 'fitted-curve
                   :display-controls (list :color :blue :line-thickness 2)
                   :points (the control-points-v-max))

 (surface :type 'basic-surface
          :display-controls (list :color :red :line-thickness 0.5)
          :curve-bottom (the surf-curve-u-min)
          :curve-top (the surf-curve-u-max)
          :curve-left (the surf-curve-v-min)
          :curve-right (the surf-curve-v-max))

 (arc-1 :type 'arc-curve
        :display-controls (list :color :green :line-thickness 2)
        :orientation (alignment :top (the (face-normal-vector :rear)))
        :center (make-point 1 0 0)
        :radius 1
        :start-angle 0
        :end-angle pi)

 (arc-2 :type 'arc-curve
        :display-controls (list :color :green :line-thickness 2)
        :orientation (alignment :top (the (face-normal-vector :rear)))
        :center (make-point 1 -2 0)
        :radius 1
        :start-angle 0
        :end-angle pi)

 (arc-3 :type 'arc-curve
        :orientation (alignment :rear (the (face-normal-vector :right)))
        :display-controls (list :color :blue :line-thickness 2)
        :center (make-point 0 -1 0)
        :radius 1
        :start-angle 0
        :end-angle pi)

 (arc-4 :type 'arc-curve
        :orientation (alignment :rear (the (face-normal-vector :left)))
        :display-controls (list :color :blue :line-thickness 2)
        :center (make-point 2 -1 0)
        :radius 1
        :start-angle 0
        :end-angle pi)

 (surf-arc :type 'basic-surface
           :display-controls (list :color :red :line-thickness 0.5)
           :curve-bottom (the arc-1 )
```

```
                    :curve-top (the arc-2 )
                    :curve-left (the arc-3 reverse )
                    :curve-right (the arc-4))))

      (generate-sample-drawing
          :objects (the-object (make-object 'test-basic-surface) surface)
          :projection-direction (getf *standard-views* :tri-r-r))
```

7.0.4.6 Blended-solid

Description

This primitive attempts to fillet one or more edges of a brep solid.

Required-input-slots

brep *gdl brep object*

This is the original solid, whose edges you want to be filleted

default-radius *number*

This will be used as the fillet radius

Optional-input-slots

specs *plist with key :edges*

This specifies which edges are to be filleted The default (nil) means that all edges should
be filleted

Examples

```
      (in-package :gdl-user)

      (define-object blend-sample (base-object)

        :objects
        ((box :type 'box-solid
              :length 10 :width 20 :height 15)

         (blend :type 'blended-solid
                :display-controls (list :color :blue)
                :default-radius 3
                :brep (the box))))

      (generate-sample-drawing :objects (the-object (make-object 'blend-sample) blend)
                               :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.7 Box-solid

Description

A rectangular box represented as a brep solid. Contains the union of messages (e.g. input-slots, computed-slots, etc) from brep and box.

Examples

```
(in-package :surf)

(define-object test-box-solid (box-solid)
   :computed-slots ((length 10) (width 20) (height 30)))

(generate-sample-drawing :objects (the-object (make-object 'test-box-solid) )
                            :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.8 Boxed-curve

Description

This object behaves as a hybrid of a curve and a normal box. You pass in a curve-in, and it essentially traps the curve in a box, which will respond to normal GDL :center and :orientation. You can also pass a scale, or scale-x, or scale-y, or scale-z as with a transformed-curve.

Required-input-slots

curve-in *gdl curve object*

This can be any type of curve, e g b-spline-curve, fitted-curve, or an edge from a solid brep Note that the reference-box of this curve (i e its center and orientation) will have an effect on the resulting boxed-curve If you want to consider the incoming curve-in as being in global space, then make sure its center is (0 0 0) and its orientation is nil or equivalent to geom-base::+identity-3x3+

Optional-input-slots

center *3d point in global space*

You can pass in a new center for the curve's reference box, which will move the whole box including the curve This defaults to the orientation-center (if given), otherwise to the (the curve-in center)

from-center *3d point in global space*

The center with respect to which this object should be positioned Normally this should not be specified by user code, unless you know what you are doing [e g to override the center of a curve-in which is meaningless and force it to be interpreted as a curve in global space,

you could specify this as (the center) when passing it in from the parent] Default is (the curve-in center)

from-object *gdl object which mixes in base-object*

The current boxed-curve will be positioned and oriented with respect to the center and orientation of this object The default is (the curve-in)

from-orientation *3x3 transformation matrix*

The orientation with respect to which this object should be oriented Normally this should not be specified by user code, unless you know what you are doing [e g to override the orientation of a curve-in which is meaningless and force it to be interpreted as a curve in the parent's coordinate system, you could specify this as (the orientation) when passing it in from the parent] Default is (the curve-in orientation)

orientation *3x3 transformation matrix*

This will be the new orientation for the box and the contained curve Default is (the curve-in orientation) – i e identical orientation with the provided curve-in

orientation-center *3d point in global space*

If you provide this, the curve's reference box will be moved to have its center at this point, before any orientation is applied This will become the new center of the resulting boxed-curve, unless you explicitly pass in a different center Default is nil

scale *number*

The overall scale factor for X, Y, and Z, if no individual scales are specified Defaults to 1

scale-x *number*

The scale factor for X Defaults to 1

scale-y *number*

The scale factor for Y Defaults to 1

scale-z *number*

The scale factor for Z Defaults to 1

show-box? *boolean*

This determines whether the reference box is displayed along with the curve Default is t (will be changed to nil)

show-control-polygon? *boolean*

This determines whether the control polygon is displayed along with the curve Default is nil

show-tight-box? *boolean*

This determines whether the tight box is displayed along with the curve Default is nil

translation-threshold *number*

Tolerance to determine whether the boxed-curve has moved with respect to the original Default is *zero-epsilon*

Computed-slots

control-points *list of 3d points*

   The control points

degree *integer*

   Degree of the curve Defaults to 3 (cubic)

height *number*

   Z-axis dimension of the reference box Defaults to zero

knot-vector *list of numbers*

   Knots of the curve Default is NIL, which indicates a uniform knot vector

length *number*

   Y-axis dimension of the reference box Defaults to zero

weights *list of numbers*

   A weight to match each control point Should be same length as control-points Default is a value of 1 0 for each weight, resulting in a nonrational curve

width *number*

   X-axis dimension of the reference box Defaults to zero

Examples

```
(define-object boxed-curves-test (base-object)

 :computed-slots ((b-spline (the b-splines (curves 2))))

 :objects
 ((b-splines :type 'test-b-spline-curves)

  (boxed :type 'boxed-curve
         :curve-in (the b-splines (curves 2)))

  (translated :type 'boxed-curve
              :curve-in (the b-spline)
              :center (translate (the center) :left 15))

  (twisted :type 'boxed-curve
           :curve-in (the boxed)
           :orientation
           (alignment :left (the (face-normal-vector :top))
                      :rear (rotate-vector-d (the (face-normal-vector :rear))
                                             30
                                             (the (face-normal-vector :top)))))
  (rotated :type 'boxed-curve
```

```
            :curve-in (the b-spline)
            :display-controls (list :color :purple)
            :orientation
            (alignment :left
                    (rotate-vector-d (the (face-normal-vector :left))
                                     50
                                     (the (face-normal-vector :rear)))))

(rotated-about :type 'boxed-curve
               :curve-in (the b-spline)
               :display-controls (list :color :purple)
               :orientation-center (translate (the center) :right 2.5)
               ;;;center (translate (the center) :up 5)
               :orientation
               (alignment :left
                          (rotate-vector-d (the (face-normal-vector :left))
                                           45
                                           (the (face-normal-vector :rear)))))

(moved-up :type 'boxed-curve
          :curve-in (the rotated-about)
          :center (translate (the rotated-about center)
                             :up 7
                             :left 5))

(straightened :type 'boxed-curve
              :curve-in (the moved-up)
              :orientation
              (alignment
               :left
               (rotate-vector-d
                (the-child curve-in (face-normal-vector :left))
                45
                (the-child curve-in (face-normal-vector :rear)))
               :rear (the-child curve-in (face-normal-vector :rear))))

(rotated-straightened :type 'boxed-curve
                      :curve-in (the straightened)
                      :orientation (the moved-up orientation)
                      :orientation-center
                      (translate (the-child curve-in center) :up 2.5))

(rotated-straightened-moved :type 'boxed-curve
                            :curve-in (the rotated-straightened)
                            :center (translate (the-child curve-in center)
                                              :right 5))
(center-sphere :type 'sphere
               :radius 0.3
               :center (the moved-up-and-straightened orientation-center))


(moved-up-and-straightened
 :type 'boxed-curve
 :curve-in (the straightened)
 :center (translate (the-child orientation-center) :right 7)
 :orientation
 (alignment :left
            (the-child curve-in  (face-normal-vector :rear))
```

```
                        :front
                        (rotate-vector-d (the-child curve-in (face-normal-vector :left))
                                         45
                                         (the-child curve-in (face-normal-vector :rear))))
           :orientation-center (translate (the straightened center) :up 2.5))

        (moved-up-and-straightened-1
         :type 'boxed-curve
         :curve-in (the straightened)
         :center (translate (the-child curve-in center) :right 14)
         :orientation (the rotated-straightened orientation)
         :orientation-center (translate (the straightened center) :up 2.5))

        (moved-up-and-straightened-2
         :type 'boxed-curve
         :curve-in (the straightened)
         :center (translate (the-child curve-in center) :right 21)
         :orientation (the rotated-straightened orientation)
         :orientation-center (translate (the straightened center) :up 2.5))

        (transformed
         :type 'boxed-curve
         :curve-in (the b-spline)
         :center (translate (the center) :left 50)
         :orientation
         (alignment :rear
                    (rotate-vector-d (the (face-normal-vector :rear))
                                     30
                                     (the (face-normal-vector :right)))))))))
```

7.0.4.9 Boxed-surface

Description

This object behaves as a hybrid of a surface and a normal box. You pass in a surface-in, and it essentially traps the surface in a box, which will respond to normal GDL :center and :orientation. You can also pass a scale, or scale-x, or scale-y, or scale-z as with a transformed-surface.

Required-input-slots

surface-in *gdl surface object*

This can be any type of surface, e g b-spline-surface, fitted-surface, or an edge from a solid brep Note that the reference-box of this surface (i e its center and orientation) will have an effect on the resulting boxed-surface If you want to consider the incoming surface-in as being in global space, then make sure its center is (0 0 0) and its orientation is nil or equivalent to geom-base::+identity-3x3+

Optional-input-slots

center *3d point in global space*

You can pass in a new center for the surface's reference box, which will move the whole box including the surface This defaults to the orientation-center (if given), otherwise to the (the surface-in center)

from-center *3d point in global space*

The center with respect to which this object should be positioned Normally this should not be specified by user code, unless you know what you are doing [e g to override the center of a surface-in which is meaningless and force it to be interpreted as a surface in global space, you could specify this as (the center) when passing it in from the parent] Default is (the surface-in orientation)

from-object *gdl object which mixes in base-object*

The current boxed-surface will be positioned and oriented with respect to the center and orientation of this object The default is (the surface-in)

from-orientation *3x3 transformation matrix*

The orientation with respect to which this object should be oriented Normally this should not be specified by user code, unless you know what you are doing [e g to override the orientation of a surface-in which is meaningless and force it to be interpreted as a surface in the parent's coordinate system, you could specify this as (the orientation) when passing it in from the parent] Default is (the surface-in orientation)

orientation *3x3 transformation matrix*

This will be the new orientation for the box and the contained surface Default is (the surface-in orientation) – i e identical orientation with the provided surface-in

orientation-center *3d point in global space*

If you provide this, the surface's reference box will be moved to have its center at this point, before any orientation is applied This will become the new center of the resulting boxed-surface, unless you explicitly pass in a different center Default is nil

scale *number*

The overall scale factor for X, Y, and Z, if no individual scales are specified Defaults to 1

scale-x *number*

The scale factor for X Defaults to 1

scale-y *number*

The scale factor for Y Defaults to 1

scale-z *number*

The scale factor for Z Defaults to 1

show-box? *boolean*

This determines whether the reference box is displayed along with the surface Default is nil

show-control-polygon? *boolean*

This determines whether the control polygon is displayed along with the surface Default is t (will be changed to nil)

show-tight-box? *boolean*

This determines whether the tight box is displayed along with the surface Default is nil

Computed-slots

control-points *list of lists of 3d points*

The control net

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

u-degree *integer*

Degree of surface in U direction Defaults to 3

u-knot-vector *list of numbers*

Knots in U direction Default is NIL, which indicates a uniform knot vector in U direction

v-degree *integer*

Degree of surface in V direction Defaults to 3

v-knot-vector *list of numbers*

Knots in V direction Default is NIL, which indicates a uniform knot vector in V direction

weights *list of lists of numbers*

A weight to match each control point Should be congruent with control-points (i e same number of rows and columns) Default is a value of 1 0 for each weight, resulting in a nonrational surface

width *number*

X-axis dimension of the reference box Defaults to zero

Examples

```
(define-object boxed-surfaces-test (base-object)

  ;;bounding-box-from-points gives errors

  :objects
  ((b-spline :type 'test-b-spline-surface)

   (boxed :type 'boxed-surface
```

```
              :surface-in (the b-spline))

   (translated :type 'boxed-surface
               :surface-in (the b-spline)
               :center (translate (the center) :left 15))

   (twisted :type 'boxed-surface
            :surface-in (the boxed)
            :orientation
            (alignment :left (the (face-normal-vector :top))
                       :rear (rotate-vector-d (the (face-normal-vector :rear))
                                              30
                                              (the (face-normal-vector :top)))))
   (rotated :type 'boxed-surface
            :surface-in (the b-spline)
            :display-controls (list :color :purple)
            :orientation
            (alignment :left
                       (rotate-vector-d (the (face-normal-vector :left))
                                        50
                                        (the (face-normal-vector :rear)))))

   (rotated-about :type 'boxed-surface
                  :surface-in (the b-spline)
                  :display-controls (list :color :purple)
                  :orientation-center (translate (the center) :right 2.5)
                  ;;;center (translate (the center) :up 5)
                  :orientation
                  (alignment :left
                             (rotate-vector-d (the (face-normal-vector :left))
                                              45
                                              (the (face-normal-vector :rear)))))

   (moved-up :type 'boxed-surface
             :surface-in (the rotated-about)
             :center (translate (the rotated-about center)
                                :up 7
                                :left 5))

   (straightened :type 'boxed-surface
                 :surface-in (the moved-up)
                 :orientation
                 (alignment :left
                            (rotate-vector-d
                             (the-child surface-in (face-normal-vector :left))
                             45
                             (the-child surface-in (face-normal-vector :rear)))
                            :rear (the-child surface-in (face-normal-vector
                                                         :rear))))

   (rotated-straightened :type 'boxed-surface
                         :surface-in (the straightened)
                         :orientation (the moved-up orientation)
                         :orientation-center
                         (translate (the-child surface-in center)  :up 2.5))

   (rotated-straightened-moved :type 'boxed-surface
                               :surface-in (the rotated-straightened)
```

```
                                      :center (translate (the-child surface-in center)
                                                         :right 5))

            (center-sphere :type 'sphere
                           :radius 0.3
                           :center (the moved-up-and-straightened orientation-center))

            (moved-up-and-straightened
             :type 'boxed-surface
             :surface-in (the straightened)
             :center (translate (the-child orientation-center) :right 7)
             :orientation
             (alignment :left (the-child surface-in (face-normal-vector :rear))
                        :front
                        (rotate-vector-d
                         (the-child surface-in (face-normal-vector :left))
                         45
                         (the-child surface-in (face-normal-vector :rear))))
             :orientation-center (translate (the straightened center) :up 2.5))

            (moved-up-and-straightened-1
             :type 'boxed-surface
             :surface-in (the straightened)
             :center (translate (the-child surface-in center) :right 14)
             :orientation (the rotated-straightened orientation)
             :orientation-center (translate (the straightened center) :up 2.5))

            (moved-up-and-straightened-2
             :type 'boxed-surface
             :surface-in (the straightened)
             :center (translate (the-child surface-in center) :right 21)
             :orientation (the rotated-straightened orientation)
             :orientation-center (translate (the straightened center) :up 2.5))

            (transformed :type 'boxed-surface
                         :surface-in (the b-spline)
                         :center (translate (the center) :left 50)
                         :orientation
                         (alignment :rear
                                    (rotate-vector-d (the (face-normal-vector :rear))
                                                     30
                                                     (the (face-normal-vector :right)))))))))
```

7.0.4.10 Brep

Description

A general superclass for all boundary representation geometric entities. This currently follows the smlib topology model, with breps containing regions, regions containing shells, and shells containing faces and edges. Shells which completely enclose a volume are considered to make up a solid brep.

Optional-input-slots

brep-tolerance *number*

Overall tolerance for the created brep solid Defaults to nil Note that a value of nil indicates for SMLib to use value of 1 0e-05 of the longest diagonal length of the brep

built-from *gdl brep object*

Defaults to nil Specify this if you want this brep to be a clone of an existing brep (note - this uses a shared underlying brep object, it does not make a copy)

face-brep-colors *list of color keywords*

These indicate the colors for the breps produced by (the face-breps) If the number of face-breps exceeds the length of this list, the colors will be repeated in order Defaults to a list with keys:

- :green
- :red
- :blue
- :purple-dark
- :violet
- :cyan

tessellation-parameters *plist of keyword symbols and numbers*

This controls tessellation for this brep The keys are as follows:

- :min-number-of-segments
- :max-3d-edge-factor
- :min-parametric-ratio
- :max-chord-height
- :max-angle-degrees
- :min-3d-edge
- :min-edge-ratio-uv
- :max-aspect-ratio

and the defaults come from the following parameters:

- (list
- :min-number-of-segments *tess-min-number-of-segments*
- :max-3d-edge-factor *tess-max-3d-edge-factor*
- :min-parametric-ratio *tess-min-parametric-ratio*
- :max-chord-height *tess-max-chord-height*
- :max-angle-degrees *tess-max-angle-degrees*
- :min-3d-edge *tess-min-3d-edge*
- :min-edge-ratio-uv *tess-min-edge-ratio-uv*
- :max-aspect-ratio *tess-max-aspect-ratio*)

Settable-optional-input-slots

density *number*

    The density per unit volume of the brep Defaults to 1

display-iso-curves-wireframe? *boolean*

    Determines whether the isoparametric curves of each face of the brep are used for wireframe display The default is T

display-tessellation-lines-wireframe? *boolean*

    Determines whether the tessellation grid-lines of the brep are used for wireframe display The default is NIL

max-3d-edge *number*

    Used for tessellations Computed from (the max-extent) and (the max-3d-edge-factor)

    WARNING: Modify this value at your peril Small values can cause intractable tessellations It is better to tweak max-3d-edge-factor to a small value like 0 1, as this will be taken relative to the max-extent of the brep

max-3d-edge-factor *number*

    Used for tessellations Default comes from (the tessellation-parameters)

max-angle-degrees *number*

    Used for tessellations Default comes from (the tessellation-parameters)

max-aspect-ratio *number*

    Used for tessellations Default comes from (the tessellation-parameters)

max-chord-height *number*

    Used for tessellations Default comes from (the tessellation-parameters)

min-3d-edge *number*

    Used for tessellations Default comes from (the tessellation-parameters)

min-edge-ratio-uv *number*

    Used for tessellations Default comes from (the tessellation-parameters)

min-number-of-segments *integer*

    Used for tessellations Default comes from (the tessellation-parameters)

min-parametric-ratio *number*

    Used for tessellations Default comes from (the tessellation-parameters)

poly-brep-smooth-results? *boolean*

    Smooth results for poly-brep? Defaults to t

Settable-defaulted-input-slots

isos *plist with keys :n-u and :n-v*

The number of isoparametric curves to be displayed in each direction This value comes from the value of :isos on the display-controls if that exists, and defaults to *isos-default* otherwise

Computed-slots

adaptive-tolerance *number*

This is the actual tolerance stored in the SMLib object

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

local-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding this geometric object

min-max-x-y-z *plist with :min-x :max-x :min-y :max-y :min-z :max-z*

Returns the extreme points of the brep in each direction (on the brep itself, not necessarily on the bounding box)

triangle-data *list of plists, one for each face, format still being determined*

Contains triangle and connectivity data for the tessellation of this brep Exact supported format will be documented here when ready

Hidden-objects

composed-edges *the composed edges contained within this brep, this is valid just if the brep does not contain holes*

edges-sequence *sequence of gdl edge objects*

The Edges contained within this brep aranged clockwise or anticlockwise, this is valid just if the brep does not contain holes

poly-brep *polygonal brep object*

This brep represented as a Polygonal Brep

Quantified-hidden-objects

edges *sequence of gdl edge objects*

The Edges contained within this brep

face-breps *sequence of gdl brep objects*

One brep for each face in the parent brep, containing only that face

faces *sequence of gdl face objects*

    The Faces contained within this brep

regions *sequence of gdl region objects*

    The Regions contained within this brep

shells *sequence of gdl shell objects*

    The Shells contained within this brep

vertices *sequence of gdl vertex objects*

    The Vertices contained within this brep

Functions

area *number*

    Area covered by the faces of the brep.

    Keyword Arguments:

      tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

area-moments-of-inertia *3d vector (i*

    e. 3D Point). Returns the Area Moments of Inertia of the brep.

    Keyword Arguments:

      tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

area-products-of-inertia *3d vector (i*

    e. 3D Point). Returns the Area Products of Inertia of the brep.

    Keyword Arguments:

      tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

area-second-moment-about-coordinate-axii *3d vector (i*

    e. 3D Point). Returns the Area Second Moment About Coordinate Axii of the brep.

    Keyword Arguments:

      tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

area-static-moments *3d vector (i*

    e. 3D Point). Returns the Area Static Moments of the brep.

    Keyword Arguments:

      tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

brep-intersect? *value nil or t*

This function performs an intersection between this brep and another brep. The function returns a NIL value if no intersection is found and T if a intersection is found.

Keyword Arguments:

tolerance Number, Controls how precisely the intersection is computed, Default Value: (the adaptive-tolerance),

angle-tolerance Number, in radians, Default Value: (radians-to-degrees *angle-tolerance-radians-default*), The angle tolerance for intersection.

center-of-gravity *3d point*

Center of gravity of the mass of the brep.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

in? *boolean*

Returns t or nil depending on whether the point given is within the boundary of the brep (including faces).

mass *number*

Mass represented by the brep, according to the density.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

moments *plist*

Returns the moments of the brep. The plist contains keys: :area-static-moments, :area-moments-of-inertia,

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

precise-properties *multiple values: number, number, number, and plist*

Returns the area, volume, mass, and moments for the brep. The moments are labeled as: :area-static-moments, :area-moments-of-inertia,

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

properties *plist with keys: :area, :volume, :barycenter*

Returns the approximate area, volume, and barycenter (center of mass) for the brep. These are computed with tessellation techniques, which may be less precise than the analytic techniques used in precise-properties, but should be faster to compute and exhibit more stability.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

tessellation *plist or list*

Contains tessellation data for the brep based on the values of the keyword args. This is used to produce the value of (the triangle-data).

Keyword Arguments:

min-number-of-segments nil, Default Value: (the min-number-of-segments),

max-3d-edge nil, Default Value: (the max-3d-edge),

min-parametric-ratio nil, Default Value: (the min-parametric-ratio),

max-chord-height nil, Default Value: (the max-chord-height),

max-angle-degrees nil, Default Value: (the max-angle-degrees),

min-3d-edge nil, Default Value: (the min-3d-edge),

min-edge-ratio-uv nil, Default Value: (the min-edge-ratio-uv),

max-aspect-ratio nil, Default Value: (the max-aspect-ratio),

in-memory? nil, Default Value: t,

volume *number*

Volume enclosed by the brep.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

volume-moments-of-inertia *3d vector (i*

e. 3D Point). Returns the Volume Moments of Inertia of the brep.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

volume-products-of-inertia *3d vector (i*

e. 3D Point). Returns the Volume Products of Inertia of the brep.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

volume-second-moment-about-coordinate-axii *3d vector (i*

e. 3D Point). Returns the Volume Second Moment about Coordinate Axii of the brep.

Keyword Arguments:

tolerance Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

volume-static-moments *3d vector (i*

e. 3D Point). Returns the Volume Static Moments of the brep.

Keyword Arguments:

> **tolerance** Controls how precisely the properties are computed, Default Value: (the adaptive-tolerance),

7.0.4.11 Brep-intersect

Description

This primitive takes two brep objects and attempts to intersect the faces of the one with the faces of the other, yielding a sequence of edges which also behave as curves.

Required-input-slots

brep *object of type brep*

The first brep for intersecting its faces

other-brep *object of type brep*

The other brep for intersecting faces

Optional-input-slots

angle-tolerance *number*

Defaults to (radians-to-degrees *angle-tolerance-radians-default*)

approximation-tolerance *number*

Defaults to the max of the adaptive-tolerance of any of the input breps

hide-edges? *boolean*

Should edges be children or hidden-children? Defaults to nil which makes them display as children

hide-points? *boolean*

Should points be children or hidden-children? Defaults to nil which makes them display as children

7.0.4.12 Compatible-curves

Description

Experimental. This primitive takes in a list of GDL curve objects and will compute a sequence of new curve objects which have been made to be compatible in terms of number of control points, knot vectors, and degree.

Optional-input-slots

curve-list *list of gdl curve objects*

tolerance *tolerance is used to check knots removability for data reduction*
  A nil value indicates that no data reduction is to be attempted Defaults to nil

Quantified-objects

curves *sequence of gdl curve objects*
  These are the resultant curves which are supposed to be compatible

Examples

```
(in-package :gdl-user)

(define-object test-compatible-curves ()

 :objects
 ((b-spline :type 'b-spline-curve
            :control-points (list (make-point 0 0 0)
                                  (make-point -1 0 0)
                                  (make-point -1 1 0)
                                  (make-point 0 1 0)))

  (linear-curve :type 'linear-curve
                :start (make-point 0 1 0)
                :end (make-point 4 1 0))

  (compatible-curves :type 'compatible-curves
                     :curve-list (list (the b-spline) (the linear-curve)))))

 (generate-sample-drawing :object-roots (list (make-object 'test-compatible-curves)))
```

7.0.4.13 Compatible-surfaces

Description
  This routine makes compatible a list of GDL surface, minimum 2 surfaces is required.

Required-input-slots

surface-list *list*
  A list of Gdl surface objects

Examples

```
(in-package :gdl-user)

(define-object compatible-surfaces-test (surface)
```

```
      :computed-slots ((surface-list (list (the surf-A) (the surf-B))))

      :objects
      ((make-compatible-A-and-B :type 'compatible-surfaces
                                :display-controls (list :line-thickness 2)
                                :surface-list (the surface-list))

       (surf-A :type 'rectangular-surface
               :display-controls (list :color :green-spring-medium)
               :length 10
               :width 10 )

       (surf-B :type 'rectangular-surface
               :display-controls (list :color :red)
               :center (make-point 10 0 0 )
               :length 10
               :width 10 )))

  (generate-sample-drawing :object-roots
                           (list (the-object (make-object 'join-surfaces-test)
                                             join-A-and-B )))
```

7.0.4.14 Composed-curve

Description

Creates a single NURBS curve from a list (ordered or unordered) NURBS curves. If the result is more than one curve, this object will throw an error and you should use `composed-curves` instead.

Required-input-slots

curves *list of gdl curve objects*

These are the curves to be composed into a single curve

Optional-input-slots

coincident-point-tolerance *number*

Distance two curve endpoints can be apart and be considered coincident, the composite will be built without doing anything to the endpoints Default is 0 01 Note: This input-slot must be non-zero

distance-to-create-line *number*

Distance two curve endpoints can be apart and have a linear curve segment automatically added between the points Default is 0 1

Examples

```
(in-package :surf)

(define-object test-composed-curve (composed-curve)
 :computed-slots
 ((curves (the filleted-polyline-curves ordered-curves)))

 :hidden-objects
 ((filleted-polyline-curves :type 'test-global-filleted-polyline-curves)))

(generate-sample-drawing :objects (the-object (make-object 'test-composed-curve))
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.15 Composed-curves

Description

Creates multiple NURBS curves by composing a list (ordered or unordered) NURBS curves. If the result is expected to be a single curve, you may wish to use `composed-curve` instead.

Required-input-slots

curves-in *list of gdl curve objects*

These are the curves to be composed into a single curve

Optional-input-slots

coincident-point-tolerance *number*

Distance two curve endpoints can be apart and be considered coincident, the composite will be built without doing anything to the endpoints Default is 0 01 Note: This input-slot must be non-zero

distance-to-create-line *number*

Distance two curve endpoints can be apart and have a linear curve segment automatically added between the points Default is 0 1

Quantified-objects

curves *sequence of gdl curve objects*

The curves resulting from composition

7.0.4.16 Cone-solid

Description

A right cone represented as a brep solid. Contains the union of messages (e.g. input-slots, computed-slots, etc) from brep and cone.

Examples

```
(in-package :surf)

(define-object test-cone-solid-hollow ()

 :objects
 ((cone-solid :type 'cone-solid
              :display-controls (list :isos (list :n-u 8 :n-v 8) :color :green)
              :length 100
              :radius-1 10
              :radius-2 20
              :inner-radius-1 8
              :inner-radius-2 16)))

(generate-sample-drawing :object-roots (make-object 'test-cone-solid-hollow)
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.17 Curve

Description

A generalized NURBS curve. Usually used as a mixin in more specific curves.

Optional-input-slots

built-from *gdl curve*

Specify this if you want this curve to be a clone of an existing curve (note - this uses a shared underlying curve object, it does not make a copy)

uv-curve *gdl curve object*

The corresponding UV curve for the primary surface on which this curve lies, if any If this is not a surface-curve, this will return an error

Settable-defaulted-input-slots

tolerance *number*

Approximation tolerance for display purposes Defaults to the tolerance of the built-from curve, if one exists, otherwise defaults to the *display-tolerance*

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

direction-vector *3d vector*

The direction pointing from the start to the end

end *3d point*

The point returned by evaluating the curve function at `u2`

on-surfaces *list of gdl surfaces*

The surfaces on which this curve lies

start *3d point*

The point returned by evaluating the curve function at `u1`

success? *boolean*

This will be t if the curve is generated successfully, nil otherwise

u-max *number*

The highest parameter value of the underlying mathematical definition for this parametric curve

u-min *number*

The lowest parameter value of the underlying mathematical definition for this parametric curve

u1 *number*

Equal to the natural `u-min` of the curve

u2 *number*

Equal to the natural `u-max` of the curve

Hidden-objects

first-derivative *gdl curve*

The first derivative of this curve The degree will be one less than the degree of this curve

second-derivative *gdl curve*

The second derivative of this curve The degree will be two less than the degree of this curve

Functions

acceleration *3d vector*

The given parameter evaluated on the second derivative curve of this curve. Note that this is only valid if this curve has degree of at least two (2), and will throw an error otherwise.

b-spline-data *list of 3d points, list of numbers, list of numbers, and integer*

Returns four values which are the control points, the weights, the knots, and the degree of the curve.

check-continuity *either t or a plist of numbers with keys <tt>:distance</tt>, <tt>angle</tt>, <tt>:length</tt>*

.

Keyword Arguments:

3d-tolerance Allowed maximum distance between curve segments, Default Value: *3d-tolerance-default*,

angle-tolerance Allowed maximum angle in radians between end/start tangents of curve segments, Default Value: *angle-tolerance-default,

minimum-segment-length Allowed minimum curve segment length, Default Value: *minimum-segment-length-default*,

closure *keyword symbol, :closed, :open, or :continuous*

curvature *number*

The reciprocal of the radius of curvature at the given parameter.

curve-intersection-point *surface point*

First point of intersection between this curve and the other curve given as the argument. This function also returns a second value, which is a surface point representing the end of a contiguous segment, if any, associated with the surface point given as the primary return value. A NIL value as this second return value indicates that there was no contiguous segment, only an intersecting point as indicated by the surface point given as the primary return value.

Keyword Arguments:

distance-tolerance Number, Default Value: *3d-tolerance-default*, Distance for two points to be considered coincident.

curve-intersection-points *surface points*

Points of intersection between this curve and another curve. This function also returns a second value, which is a list of surface points representing the ends of contiguous segments, if any, associated with the surface point in the same position in the primary returned list. NIL values in this second returned list indicate that there was no contiguous segment, only an intersecting point as indicated by the surface point in the primary returned list.

Keyword Arguments:

distance-tolerance Number, Default Value: *3d-tolerance-default*, Distance for two points to be considered coincident.

dropped-curve *list of plists*

The returned list of plists contains information about the points where the tangents of this curve and those of the curve given as the argument are equal.

dropped-point *surface point*

Given a 3D point, returns the point(s) projected normally onto the curve.

Keyword Arguments:

distance-tolerance Number, Default Value: [covers, The 3D point must be within this distance of the curve for a successful drop.

equi-spaced-parameters *list of numbers*

Returns the specified number of parameters equally spaced along the curve.

Keyword Arguments:

spacing Keyword Symbol, :parametric or :arc-length, Default Value: arc-length, :arc-length is the default.

equi-spaced-points *list of 3d points*

Returns the specified number of points equally spaced along the curve.

Keyword Arguments:

spacing Keyword Symbol, :parametric or :arc-length, Default Value: arc-length, Defaults to :arc-length.

arc-length-approximation-tolerance Number, Default Value: *zero-epsilon*, A smaller number gives tighter arc length approximation. Scaled to curve exent.

in-plane? *boolean*

Given a point and a vector defining a plane, returns T or NIL depending whether this curve lies in the plane. Also returns a second value which is the maximum distance of the curve from the plane.

Keyword Arguments:

distance-tolerance Number, Default Value: *3d-tolerance-default*, Allowed distance of any point on the curve from the plane.

Returns a bbox object, answering xmin, ymin, zmin, xmax, ymax, and zmax, for a box containing the convex hull (i.e. the control points) of this curve and oriented according to the given `center` and `orientation`.

maximum-distance-to-curve *plist*

The returned plist contains information about the maximum distance from this curve to the curve given as the argument.

maximum-distance-to-point *plist*

The returned plist contains information about the maximum distance from this curve to the point given as the argument.

minimum-distance-to-curve *plist*

The returned plist contains information about the minimum distance from this curve to the curve given as the argument.

minimum-distance-to-point *plist*

The returned plist contains information about the minimum distance from this curve to the point given as the argument.

minimum-radius *number*

The minimum radius of curvature for the curve. A second value is also returned, which is a surface point indicating the point on the curve where this minimum radius occurs. A third value is also returned, which is a list of additional curve parameters where similar minimum radii occur.

normal *3d vector*

The normal of the curve at the given parameter value, i.e. the vector pointing from the point on the curve at this parameter to the center of the osculating circle at that point. if the curve has no curvature at the given parameter, NIL is returned.

offset-point-along *surface point*

Returns point at given parameter offset by given distance.

on? *boolean*

Returns non-NIL if the given parameter lies within the parameter range of this curve.

parameter-at-length *number*

Returns the parameter representing a point offset from the start of the curve by the given length.

parameter-at-point *number*

Returns the parameter of the given point on the curve.

parameter-bounds *numbers (multiple return values)*

The minimum and maximum parameter values for this parametric curve.

plane-intersection-point *surface point*

First point of intersection between this curve and the plane denoted by plane-point and plane-normal

plane-intersection-points *surface points*

Points of intersection between this curve and the plane denoted by plane-point and plane-normal.

point *3d point*

The point on the curve corresponding to the given parameter value.

radius-of-curvature *number*

The radius of curvature (i.e. radius of the osculating circle) at the given parameter.

surface-intersection-point *surface point*

Returns the first point of intersection between this curve and the surface given as an argument.

Keyword Arguments:

3d-tolerance Number, Default Value: *3d-tolerance-default*, The tolerance to use for intersecting.

surface-intersection-points *list of surface points*

Returns the point(s) of intersection between this curve and the surface given as an argument.

Keyword Arguments:

3d-tolerance Number, Default Value: *3d-tolerance-default*, The tolerance to use for intersecting.

tangent *3d vector*

The curve tangent at the given parameter value. Supplementary values returned are: the 3D point at the parameter value. If keyword argument `:include-curvature?` is given as non-NIL, the radius of the osculating circle, the center for the osculating circle, the normal for the osculating circle, and the curve normal are also returned. Note: If `:include-curvature?` is given as non-NIL and the curve has no curvature at the specified parameter, NIL is returned for each of these four values.

Keyword Arguments:

include-curvature? Boolean (T or NIL), Default Value: nil, Indicates whether to compute curvature information.

tangent-points *list of plists*

The returned list of plists contains information about the points where the tangents of this curve and the vector given as the argument are equal.

total-length *number*

The total length of the curve from given start-parameter to given end-parameter.

Keyword Arguments:

u1 Number, Default Value: (the u1), The start parameter for computing the length.

u2 Number, Default Value: (the u2), The end parameter for computing the length.

tolerance Number, Default Value: 0.01, The tolerance (absolute or relative to curve extent) for computing the length.

tolerance-type Keyword Symbol, :relative or :absolute, Default Value: relative, The type of the specified tolerance.

trim *gdl curve object*

Returns a curve which is trimmed from parameter-1 to parameter-2.

Examples

```
(in-package :surf)

(define-object test-curve (curve)

 :input-slots
 ((built-from (the curv-in)))

 :computed-slots
  ((control-pts (list (make-point 3 5 1)
                      (make-point 5 8.0 1)
                      (make-point 7 10.0 1)
                      (make-point 8 5.0 1)
                      (make-point 7 0.0 1)
                      (make-point 5 0.0 1)
                      (make-point 3 5 1))))

   :hidden-objects
    ((curv-in  :type 'b-spline-curve
               :control-points (the control-pts))))

  (generate-sample-drawing :object-roots (make-object 'test-curve)
```

```
                              :projection-direction :top)
```

7.0.4.18 Cylinder-solid

Description

A right cylinder represented as a brep solid. Contains the union of messages (e.g. input-slots, computed-slots, etc) from brep and cylinder

Examples

```
(in-package :surf)

(define-object test-cone-solid (cone-solid)
  :computed-slots
  ((display-controls (list :isos (list :n-u 8 :n-v 8) :color :green))
   (length 100) (radius-1 10) (radius-2 20)))

(generate-sample-drawing :objects (the-object (make-object 'test-cone-solid))
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.19 Decomposed-curves

Description

Given an input curve, creates a sequence of curve segments that do not contain knots with degree-fold mutiplicity, i.e. each piece is at least C^ continuous.

Required-input-slots

curve-in *gdl curve*

Curve (presumably multi-segment) to be decomposed

Quantified-objects

curves *sequence of gdl curve objects*

The resulting segment curves after decomposition

Examples

```
(in-package :surf)

(define-object test-decomposed-curves (decomposed-curves)
:computed-slots
((curve-in (the composed-curve)))

:objects ((composed-curve :type 'test-composed-curve)))
```

```
(generate-sample-drawing :object-roots (make-object 'test-decomposed-curves))
```

7.0.4.20 Dropped-curve

Description

Creates a 3D curve which is the `curve-in` dropped normally to the `surface.`. The resulting 3D curve contains a uv-curve which is typically useful for trimming.

NOTE: Change from 1577p027 and forward – the dropped curve now is a 3D curve which can be drawn. It contains its uv representation on the surface. Previously, the uv-curve was the actual dropped-curve.

Required-input-slots

curve-in *gdl nurbs curve*

The curve to be dropped to the surface

surface *gdl nurbs surface*

The surface on which the `curve-in` is to be dropped

Computed-slots

built-from *gdl curve*

Specify this if you want this curve to be a clone of an existing curve (note - this uses a shared underlying curve object, it does not make a copy)

Hidden-objects

uv-curve *gdl curve object*

The corresponding UV curve for the primary surface on which this curve lies, if any If this is not a surface-curve, this will return an error

Examples

```
(in-package :surf)

(define-object test-trimmed-from-dropped-2 (trimmed-surface)

  :computed-slots
  ((uv-inputs t)
   (holes (list (the hole uv-curve)))
   (island (the island-3d uv-curve))
   (reverse-island? t) (reverse-holes? t)
```

```
          (display-controls (list :color :blue :line-thickness 2)))

        :hidden-objects
        ((basis-surface :type 'test-fitted-surface
                        :display-controls (list :color :grey-light-very)
                        :grid-length 10 :grid-width 10 :grid-height 5)

         (island-3d :type 'dropped-curve
                    :curve-in (the raised-island)
                    :surface (the basis-surface))

         (hole :type 'dropped-curve
               :curve-in (the raised-hole)
               :surface (the basis-surface))

         (raised-hole :type 'b-spline-curve
                      :display-controls (list :color :grey-light-very)
                      :control-points (list (make-point 3.5 4.5 7)
                                            (make-point 4.5 6 7)
                                            (make-point 5.5 7 7)
                                            (make-point 6 4.5 7)
                                            (make-point 5.5 2 7)
                                            (make-point 4.5 2 7)
                                            (make-point 3.5 4.5 7)))

         (raised-island :type 'b-spline-curve
                        :display-controls (list :color :grey-light-very)
                        :control-points (list (make-point 3 5 7)
                                              (make-point 5 8 7)
                                              (make-point 7 10 7)
                                              (make-point 8 5 7)
                                              (make-point 7 0 7)
                                              (make-point 5 0 7)
                                              (make-point 3 5 7)))))

      (generate-sample-drawing
       :objects (let ((self (make-object 'test-trimmed-from-dropped-2)))
                  (list (the basis-surface) self (the raised-hole)
                        (the raised-island)))
       :projection-direction :trimetric)
```

7.0.4.21 Edge

Description

!!! Not applicable for this object !!!

Computed-slots

faces *list of gdl face objects*

The faces connected to this edge

Functions

uv-curve *gdl curve object*

This represents the UV curve for this edge on the given surface. Note that you have to pass in the surface, which should be the basis-surface of a face connected to this edge. The GDL edge object will be supplemented with a sequence of faces which are connected with this edge.

7.0.4.22 Edge-blend-surface

Description

Creates a smooth blend-surface between curve-1, lying on surface-1 and curve-2 on surface-2.

Note that curve-1 and curve-2 have to be so-called on-surface curves, which means they must answer a uv-curve message which is the UV representation of the curve on the given surface. The most common way to establish an on-surface curve is to use an iso-curve to begin with, or to use a projected-curve or dropped-curve to ensure that the curve is indeed an on-surface curve.

The local start and end directions of this surface at any point along curve-1 and curve-2 are determined from the cross-product between the tangent to the surface's u- or v-iso-curve (the one that is closest to being parallel to curve-1 or curve-2) at this point and the corresponding surface-normal at the same point. In this fashion a tangent blend is created between surface-1 and surface-2 that extends in a direction that follows and smoothly interpolates both surface's iso-curves. Takes the same inputs as general-dual-blend-surface, except for f-tangent-1 and f-tangent-2.

Optional-input-slots

curve-1-uv *gdl uv curve object*

Defaults to the curve-1 uv-curve

curve-2-uv *gdl uv curve object*

Defaults to the curve-2 uv-curve

curve-side-1 *keyword*

Used to specify the side w r t curve-1 in which the tangent blend-surface is to extend Takes either :right-side or :left-side as input Defaults to :right-side

curve-side-2 *keyword*

Used to specify the side w r t curve-2 in which the tangent blend-surface is to extend Takes either :right-side or :left-side as input Defaults to :right-side

Computed-slots

f-tangent-1 *input-function*

Parametric function defined from 0 to 1 that outputs the blend-surface's local direction vector along curve-1 The input value of 0 corresponds to the start of curve-1, 1 to the end of curve-1

f-tangent-2 *input-function*

Parametric function defined from 0 to 1 that outputs the blend-surface's local direction vector along curve-2 The input value of 0 corresponds to the start of curve-2, 1 to the end of curve-2

Examples

```
(in-package :surf)

(define-object test-e-b-s (base-object)


 :objects
 ((e-b-s :type 'edge-blend-surface
         :display-controls (list :color :green)
         :curve-side-1 :left-side
         :curve-side-2 :left-side
         :pass-down (curve-1 surface-1 curve-2 surface-2))


  (surf-1-top :type 'linear-curve
              :hidden? t
              :start (make-point -5 -5 0)
              :end (make-point 5 -5 0))

  (surf-1-bottom :type 'linear-curve
                 :hidden? t
                 :start (make-point -7 -10 -2)
                 :end (make-point 7 -10 -2))

  (surface-1 :type 'ruled-surface
             :curve-1 (the surf-1-top)
             :curve-2 (the surf-1-bottom))

  (curve-1 :type 'iso-curve
           :display-controls (list :color :red :line-thickness 4)
           :surface (the surface-1)
           :parameter 0
           :u-or-v :v)

  (surf-2-bottom :type 'linear-curve
                 :hidden? t
                 :start (make-point -5 5 0)
                 :end (make-point 5 5 0))

  (surf-2-top :type 'linear-curve
              :hidden? t
              :start (make-point -7 10 2)
              :end (make-point 7 10 2))

  (surface-2 :type 'ruled-surface
```

```
                   :curve-1 (the surf-2-bottom)
                   :curve-2 (the surf-2-top))

    (curve-2 :type 'iso-curve
             :display-controls (list :color :blue :line-thickness 4)
             :surface (the surface-2)
             :parameter 0
             :u-or-v :v)))

  (generate-sample-drawing :object-roots (list (make-object 'test-e-b-s))
                             :projection-direction (getf *standard-views* :top))
```

7.0.4.23 Elliptical-curve

Description

An ellipse represented exactly as a quintic NURBS curve. Inputs are the same as for ellipse. Messages are the union of those for ellipse and those for curve.

Examples

```
    (in-package :surf)

    (define-object test-elliptical-curve (elliptical-curve)
      :computed-slots
      ((center (make-point 0 0 0))
       (major-axis-length 10)
       (minor-axis-length 5)
       (start-angle 0)
       (end-angle 2pi)))

    (generate-sample-drawing :objects (make-object 'test-elliptical-curve))
```

7.0.4.24 Extended-curve

Description

Creates an extended curve extending its start or end (`u1` and `u2`).

Required-input-slots

curve-in *gdl curve object*

The underlying curve from which to build this curve

Optional-input-slots

continuity *keyword*

Specified the extention continuity If :g1 the curve is extended by a linear segment Curve-in is at least G1(possibly C1) where the extension joins the original curve If :g2 the curve is extended by reflection, yielding a G2 continuous extension If :cmax the extension yields infinite(C) continuity at the join point(no knot there) Defaults to the `:g2`

distance *number*

Specified the distance to which the curve is extended

distance-type *keyword*

Specified if the distance is an absolute distance `:absolute` or the distance is scaled by the curve's arc length to yield the desired extension distance `:relative` Defaults to the `:absolute`

extending-from *keyword*

Specified from which end the curve to be extended If :start the curve is extended back from its start point If :end the the curve is extended forward from its end point Defaults to the `:start`

Examples

```
(in-package :gdl-user)

(define-object test-extended-curve ()

  :objects
  ((b-spline-curve :type 'b-spline-curve
                   :display-controls (list :color :black :line-thickness 3.0)
                   :control-points (list (make-point -2.0 0.0 0.0)
                                         (make-point  0.0 1.0 0.0)
                                         (make-point  1.0 0.0 0.0)
                                         (make-point  0.0 -1.0 0.0)))


   (extended-curve-G1 :type 'extended-curve
                      :curve-in (the b-spline-curve)
                      :distance 2.5
                      :distance-type :absolute
                      :extending-from :start
                      :continuity :g1
                      :display-controls (list :color :red))

   (extended-curve-G2 :type 'extended-curve
                      :curve-in (the b-spline-curve)
                      :distance 2.5
                      :distance-type :absolute
                      :extending-from :start
                      :continuity :g2
                      :display-controls (list :color :green))

   (extended-curve-Cmax :type 'extended-curve
                        :curve-in (the b-spline-curve)
                        :distance 2.5
                        :distance-type :absolute
```

```
                              :extending-from :start
                              :continuity :cmax
                              :display-controls (list :color :blue))))

       (generate-sample-drawing :object-roots (make-object 'test-extended-curve))
```

7.0.4.25 Extended-surface

Description

Extends a surface to a curve, so that curve will become one of the new boundaries for the surface. Continuity is controlled via options. Note that in the example, extended and extended-2 do not give a smooth transition to the extended part of the surface because the original surface is only degree 1 in the direction of extension.

Required-input-slots

curve *gdl curve object*

   The curve to which the surface should be extended

surface *gdl surface object*

   The surface to be extended

Optional-input-slots

continuity *keyword symbol, if deformation-param is given this can be one of :c1, :c2, or :cmax, and if deformation-param is not given this can be one of :g1 or :cmax*

   Default is :c1 if deformation-param is given and :g1 if deformation-param is not given

deformation-param *number, either a u or a v surface parameter*

   This value, if given, controls how far inward from the affected boundary the surface is modified If (the direction) is :u, then this will be a U parameter, and likewise if (the direction) is :v, then this will be a V parameter Default is nil which indicates no specific control over any deformation

direction *keyword symbol, one of :u or :v*

   The direction of extension Note that if deformation-param is given, it will be a U parameter if this input :u, and a V parameter if this input is :v Default is :u

which-end *keyword symbol, one of :start or :end*

   Default is :start

Examples

```
(in-package :gdl-user)

(define-object extended-surface-test (base-object)

  :computed-slots
  ((regression-test-data (list (multiple-value-list (the extended b-spline-data))
                               (multiple-value-list (the extended-2 b-spline-data))
                               (multiple-value-list (the extended-3 b-spline-data))
                               (multiple-value-list (the extended-4 b-spline-data)))))

   (display-list-objects (list (the loft)
                               (the extended)
                               (the extended-2))))

  :objects
  ((test3 :type 'linear-curve
          :start (make-point 0 0 0)
          :end (make-point 10 0 0))

   (test4 :type 'linear-curve
          :start (make-point 0 10 0)
          :end (make-point 10 10 0))


   (mid-1 :type 'linear-curve
          :start (make-point 0 5.0 1)
          :end (make-point 10 5.0 1))

   (mid-2 :type 'linear-curve
          :start (make-point 0 8.0 1)
          :end (make-point 10 8.0 1))

   (bridge-1 :type 'b-spline-curve
             :control-points (list (make-point 0 0 0)
                                   (make-point -2 5.0 3)
                                   (make-point -2 8.0 3)
                                   (make-point 0 10 0)))
   (bridge-2 :type 'b-spline-curve
             :control-points (list (make-point 10 0 0)
                                   (make-point 12 5.0 5)
                                   (make-point 12 8.0 5)
                                   (make-point 10 10 0)))

   (bridge-3 :type 'b-spline-curve
             :control-points (list (make-point 0 -1 0)
                                   (make-point 3 -1 5)
                                   (make-point 7 -1 5)
                                   (make-point 10 -1 0)))


   (loft :type 'lofted-surface
         :curves (list (the test3) (the mid-1)
                       (the mid-2) (the test4)))

   (extended :type 'extended-surface
             :display-controls (list :color :red :line-thickness 2)
             :surface (the loft)
```

```
                    :curve (the bridge-1)
                    :direction :v
                    :which-end :start)

         (extended-2 :type 'extended-surface
                    :display-controls (list :color :green :line-thickness 2)
                    :surface (the loft)
                    :curve (the bridge-1)
                    :direction :v
                    :which-end :start
                    :deformation-param (+ (* 0.25 (- (the-child surface v-max)
                                                      (the-child surface v-min)))
                                          (the-child surface u-min)))

         (extended-3 :type 'extended-surface
                    :display-controls (list :color :orange
                                           :isos (list :n-u 25 :n-v 25))
                    :surface (the loft)
                    :curve (the bridge-3)
                    :direction :u
                    :continuity :cmax
                    :which-end :start)

         (extended-4 :type 'extended-surface
                    :display-controls (list :color :blue
                                           :isos (list :n-u 25 :n-v 25))
                    :surface (the loft)
                    :curve (the bridge-3)
                    :direction :u
                    :deformation-param (+ (* 0.25 (- (the-child surface u-max)
                                                      (the-child surface u-min)))
                                          (the-child surface u-min))
                    :continuity :cmax
                    :which-end :start)))

    (generate-sample-drawing :objects (the-object (make-object 'extended-surface-test)
                                                  display-list-objects)
                            :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.26 Extruded-solid

Description

!!! Not applicable for this object !!!

Required-input-slots

profile *gdl curve object*

   The profile to be extruded into a solid

Optional-input-slots

axis-vector *3d vector*

    The direction of extrusion Defaults to (the (face-normal-vector :top))

brep-tolerance *number*

    Overall tolerance for the created brep solid Defaults to nil Note that a value of nil indicates for SMLib a value of 1 0e-05 of the longest diagonal length of the brep

distance *number*

    The distance to extrude the profile along the axis-vector Defaults to (the (height)

7.0.4.27 Face

Description

    This object represents a (possibly) trimmed surface contained within a brep. It answers all the local messages of face, and it has surface mixed in as well, so it will answer all the surface messages. Note however that the local surface messages will operate on the basis, not on the trimmed representation. The messages for face will operate on the trimmed representation.

    This object is not meant for direct instantiation; rather, a brep will contain a quantified set of faces (called "faces"), and trimmed surface also mixes in face, so a trimmed-surface will answer all the face messages.

Required-input-slots

brep *gdl brep object*

    This is the brep object which contains this face object

Optional-input-slots

basis-surface *gdl nurbs surface*

    The underlying surface, before any trimming takes place

Computed-slots

bounding-box *list of two 3d points*

    The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

u-iso-curves *sequence of curve objects*

    The isoparametric curves in the U direction

v-iso-curves *sequence of curve objects*

The isoparametric curves in the V direction

Quantified-hidden-objects

edges *sequence of gdl edge objects*
   The Edges contained within this brep

Functions

area *number*
   Returns the area of the face.
   Keyword Arguments:
     desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).
     face-thickness Number, Default Value: 0.0, Optional thickness.

area-moments-of-inertia *3d vector (ie 3d point)*
   Returns the Area Moments of Inertia of the face.
   Keyword Arguments:
     desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).
     face-thickness Number, Default Value: 0.0, Optional thickness.

area-products-of-inertia *3d vector (ie 3d point)*
   Returns the Area Products of Inertia of the face.
   Keyword Arguments:
     desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).
     face-thickness Number, Default Value: 0.0, Optional thickness.

area-second-moment-about-coordinate-axii *3d vector (ie 3d point)*
   Returns the Area Second Moment About Coordinate Axii of the face.
   Keyword Arguments:
     desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).
     face-thickness Number, Default Value: 0.0, Optional thickness.

area-static-moments *3d vector (ie 3d point)*
   Returns the Area Static Moments of the face.
   Keyword Arguments:
     desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

moments *plist*

Returns the moments of the face. The plist contains keys: :area-static-moments, :area-moments-of-inertia, :area-products-of-inertia, :area-second-moment-about-coordinate-axii, :volume-static-moments, :volume-moments-of-inertia, :volume-products-of-inertia, and :volume-second-moment-about-coordinate-axii.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

precise-properties *multiple values: number, number, and plist*

Returns the area, volume, and moments of the face. The moments are labeled as: :area-static-moments, :area-moments-of-inertia, :area-products-of-inertia, :area-second-moment-about-coordinate-axii, :volume-static-moments, :volume-moments-of-inertia, :volume-products-of-inertia, and :volume-second-moment-about-coordinate-axii.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness of the trimmed surface.

volume *number*

Returns the volume of the face.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

volume-moments-of-inertia *3d vector (ie 3d point)*

Returns the Volume Moments of Inertia of the face.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

volume-products-of-inertia *3d vector (ie 3d point)*

Returns the Volume Products of Inertia of the face.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

volume-second-moment-about-coordinate-axii *3d vector (ie 3d point)*

Returns the Volume Second Moment about Coordinate Axii of the face.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

volume-static-moments *3d vector (ie 3d point)*

Returns the Volume Static Moments of the face.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

face-thickness Number, Default Value: 0.0, Optional thickness.

7.0.4.28 Fitted-curve

Description

Fits a curve through a set of points with given degree and parameterization.

Required-input-slots

points *list of 3d points*

The points for fitting

Optional-input-slots

degree *integer*

The desired degree of the resultant curve Default is 3, unless there are fewer than four control point given, in which case it one less than the number of control points

interpolant? *boolean*

Indicates whether the curve will interpolate the points Defaults to T

parameterization *keyword symbol, one of :uniform, :chord-length, :centripetal*

The parameterization to use in the resultant curve Default is :centripetal Note that the NLib documentation states that when specifying vectors and a vector-type of `:tangents` or `:first-last`, `:chord-length` is a recommended value for parameterization If vectors are used and the vector-type is `:normals`, this input has no effect The default is `:chord-length`

tolerance *number or nil*

The allowed tolerance for doing data reduction after the initial fitting A `nil` value indicates that no data reduction is to be attempted Defaults to nil

vector-type *keyword symbol, one of <tt>:tangents</tt>, <tt>:normals</tt>, or <tt>:first-last</tt>*

:Tangents indicates that the :vectors specify a tangent vector at each point (there should be one vector for each point), :normals indicates that the :vectors specify a normal vector at each point (there should be one vector for each point), and :first-last indicates that the :vectors specify the starting and ending tangent (in this case there should be two vectors in the :vectors list Default is :tangents)

vectors *list of 3d vectors*

Optional list of vectors used to influence the fitting Default is NIL

Examples

```
(in-package :surf)

(define-object test-fitted-curve (fitted-curve)

  :computed-slots
  ((points (the circle (equi-spaced-points 20)))))

  :hidden-objects ((circle :type 'circle :radius 10)
                   (spheres  :type 'sphere
                             :sequence (:size (length (the points)))
                             :radius 0.2
                             :center (nth (the-child :index) (the points))
                             :display-controls (list :color :blue-neon))))


  (generate-sample-drawing :objects (let ((self (make-object 'test-fitted-curve)))
                                       (cons self (list-elements (the spheres)))))
```

7.0.4.29 Fitted-surface

Description

Fits a surface through a net of points with given degrees and parameterizations. Currently only interpolated surfaces are supported, this will be extended to allow smooth fitting without the surface necessarily interpolating (going through) each of the points.

Required-input-slots

points *list of lists of 3d points*

The points for fitting, with inner lists representing U direction and outer lists V direction

Optional-input-slots

c11? *boolean*

If interpolated, indicates whether to compute a C11 continuous nonrational bicubic NURBS surface Defaults to nil

interpolant? *boolean*

Indicates whether the surface will interpolate the points Defaults to t

normals *list of 3d vectors of same length as points, or nil*

If given, these are the surface normals at each point

parameterization *keyword symbol, one of :uniform, :chord-length, :centripetal*

The parameterization to use in the resultant surface if interpolant? is t Default is :chord-length

tangent-method *keyword symbol, one of :bessel, :akima*

The method used to compute tangents Defaults to :akima

tolerance *number*

Tolerance for fit Defaults to *3d-approximation-tolerance-default*

u-degree *integer*

The desired degree of the resultant surface in the U direction Default is 3

u-start *integer*

The starting degree for the fit algorithm in the U direction Default is 1

v-degree *integer*

The desired degree of the resultant surface in the V direction Default is 3

v-start *integer*

The starting degree for the fit algorithm in the V direction Default is 1

Computed-slots

bounding-box *list of two 3d points*

The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

Examples

```
(in-package :gdl-user)

(define-object c11-test (surface)

 :input-slots ()

 :computed-slots ()

 :objects
 ((surf-test :type 'fitted-surface
             :hidden nil
             :c11? t
             :points (list (list (make-point -1 0 0)
```

```
                              (make-point 0 0 0)
                              (make-point 0.001 0.0 0)
                              (make-point 1 1 0)
                              (make-point 1.001 1 0)
                              (make-point 2 1 0)
                              (make-point 2.001 1 0)
                              (make-point 3 2 0)
                              (make-point 3.001 2.001 0)
                              (make-point 4 3 0)
                              (make-point 5 4 0))
                         (list
                          (make-point -1 0 1)
                          (make-point 0 0 1)
                          (make-point 0.001 0.0 1)
                          (make-point 1 1 1)
                          (make-point 1.001 1 1)
                          (make-point 2 1 1)
                          (make-point 2.001 1 1)
                          (make-point 3 2 1)
                          (make-point 3.001 2.001 1)
                          (make-point 4 3 1)
                          (make-point 5 4 1))))))


(define-object test-fitted-surface (fitted-surface)

  :input-slots
  ((display-controls (list :color :green-spring :isos (list :n-v 19 :n-u 19)))

   (grid-width 4 :settable) (grid-length 4 :settable) (grid-height 4 :settable))

  :computed-slots
  ((points (list (list (make-point 0 0 0)
                       (make-point (/ (the grid-width) 4) 0 0)
                       (make-point (half (the grid-width)) 0 0)
                       (make-point (* 3/4 (the grid-width)) 0 0)
                       (make-point (the grid-width) 0 0))

                 (list (make-point 0 (/ (the grid-length) 4) 0)
                       (make-point (/ (the grid-width) 4)
                                   (/ (the grid-length) 4)
                                   (/ (the grid-height) 4))
                       (make-point (half (the grid-width))
                                   (/ (the grid-length) 4)
                                   (* (/ (the grid-height) 4) 1.6))
                       (make-point (* 3/4 (the grid-width))
                                   (/ (the grid-length) 4)
                                   (/ (the grid-height) 4))
                       (make-point (the grid-width)
                                   (/ (the grid-length) 4) 0))

                 (list (make-point 0 (half (the grid-length)) 0)
                       (make-point (/ (the grid-width) 4)
                                   (half (the grid-length))
                                   (* (/ (the grid-height) 4) 1.8))
                       (make-point (half (the grid-width))
                                   (half (the grid-length))
                                   (the grid-height))
```

```
                             (make-point (* 3/4 (the grid-width))
                                         (half (the grid-length))
                                         (* 3/4 (the grid-height)))
                             (make-point (the grid-width) (half (the grid-length)) 0))

                       (list (make-point 0 (* 3/4 (the grid-length)) 0)
                             (make-point (/ (the grid-width) 4)
                                         (* 3/4 (the grid-length))
                                         (min (* (/ (the grid-height) 4)
                                                 (* (/ (the grid-height) 4) 1.4))
                                              (the grid-height)))
                             (make-point (half (the grid-width))
                                         (* 3/4 (the grid-length))
                                         (min (* (/ (the grid-height) 4)
                                                 (* (/ (the grid-height) 4) 1.8))
                                              (the grid-height)))
                             (make-point (* 3/4 (the grid-width))
                                         (* 3/4 (the grid-length))
                                         (/ (the grid-height) 4))
                             (make-point (the grid-width)
                                         (* 3/4 (the grid-length)) 0))

                       (list (make-point 0 (the grid-length) 0)
                             (make-point (/ (the grid-width) 4) (the grid-length) 0)
                             (make-point (half (the grid-width)) (the grid-length) 0)
                             (make-point (* 3/4 (the grid-width)) (the grid-length) 0)
                             (make-point (the grid-width) (the grid-length) 0))))))

     (generate-sample-drawing :objects (make-object 'test-fitted-surface)
                              :projection-direction :trimetric)
```

7.0.4.30 Global-filleted-polyline-curve

Description

Provides a singular composed curve made from a global-filleted-polyline-curves object

Computed-slots

curves *list of gdl curve objects*

These are the curves to be composed into a single curve

7.0.4.31 Global-filleted-polyline-curves

Description

Produces a list of linear-curves and arc-curves which represent the straight sections and fillets of a global-filleted-polyline. Note also global-filleted-polyline-curve, which composes the segments together into a single curve.

Computed-slots

ordered-curves *list of gdl nurbs curve objects*

    The curve segments in the right order for chaining together

Quantified-hidden-objects

fillet-curves *gdl sequence of gdl nurbs curve objects*

    The arc-curves representing the fillets

straight-curves *gdl sequence of gdl nurbs curve objects*

    The linear-curves representing the straights

Examples

```
(in-package :surf)

(define-object test-global-filleted-polyline-curves (global-filleted-polyline-curves)

 :computed-slots
 ((default-radius 5)
  (vertex-list (list (make-point 0 0 0)
                     (make-point 10 10 0)
                     (make-point 30 10 0)
                     (make-point 40 0 0)
                     (make-point 30 -10 0)
                     (make-point 10 -10 0)
                     (make-point 0 0 0))))

 :hidden-objects
 ((points :type 'point
          :sequence (:size (length (rest (the vertex-list))))
          :center (nth (the-child index) (rest (the vertex-list))))

  (view :type 'base-view
        :page-width (* 5 72) :page-height (* 5 72)
        :objects (cons self (list-elements (the points))))))

(with-format (pdf "/tmp/example.pdf" :page-width (* 5 72) :page-length (* 5 72))
   (let ((obj (make-object 'test-global-filleted-polyline-curves)))
     (write-the-object obj view cad-output)))
```

7.0.4.32 Iges-reader

Description

    This object will reflect the contents of an iges file containing points, curves, surfaces, and/or breps (including trimmed surfaces) as sequences of GDL objects.

    The HarmonyWare reader creates a log file in a temporary directory. The location of this log file is printed on the console during the reading operation. Currently this log file is not automatically deleted, and its name is determined by the system.

Required-input-slots

file-name *string or pathname*

   The location of the IGES file to be read

Optional-input-slots

group-trimmed-surfaces-into-brep? *boolean*

   If true, group all trimmed surfaces in the file into one B-rep If some trimmed surfaces are blanked, they are grouped into a second, blanked B-rep Default is nil

make-all-surfaces-trimmed? *boolean*

   If true, treat all untrimmed surfaces in the file as if they are trimmed surfaces with the natural outer boundary of the surface as the trimming loop If used, no standalone IwSurface surface objects will ever be returned by the reader Default is nil

make-single-brep? *boolean*

   If true, group all trimmed surfaces and B-reps in the file into one B-rep If some trimmed surfaces or B-reps are blanked, they are grouped into a second, blanked B-rep Default is nil

sew-brep-faces? *boolean*

   Indicates whether each resulting brep should have its faces sewn together Default is (the make-single-brep?)

Quantified-objects

breps *sequence of gdl brep objects*

   The breps and trimmed surfaces (represented as breps) found in the IGES file

curves *sequence of gdl curve objects*

   The curves found in the IGES file

points *sequence of gdl point objects*

   The points found in the IGES file

surfaces *sequence of gdl surface objects*

   The untrimmed "standalone" surfaces found in the IGES file

7.0.4.33 Intersected-solid

Description

   Given two brep solids, performs the intersect Boolean between the brep and the other-brep

7.0.4.34 Iso-curve

Description

Represents an exact iso curve on the given surface in given direction at given parameter value.

Required-input-slots

parameter *number*

The u or v will be fixed at this value This should be between the min and max values for the surface in the given direction Note that you can check the min and max for a surface with (the u-min), (the u-max), (the v-min), or (the v-max)

surface *gdl object of type surface*

The surface on which you want an iso curve

Optional-input-slots

fixed-parameter *keyword symbol, one of :u or :v*

This is the parameter direction which will be fixed to the parameter value given

Default is :u

u-or-v *keyword symbol, one of :u or :v*

This is the direction of the iso-curve The other parameter will be fixed on the surface at the given parameter value

:note The name of this input-slot is somewhat anti-intuitive If you want to specify the U parameter, you give :v here, and if you want to specify the V parameter, you give :u here For an intuitive input which matches the parameter value that you are giving, use fixed-parameter instead of this input-slot

Default is (ecase (the fixed-parameter) (:u :v) (:v :u))

Computed-slots

on-surfaces *list of gdl surfaces*

For iso curve, this will contain the single surface on which the iso curve lies

Hidden-objects

uv-curve *nil*

Examples

```
(in-package :gdl-user)

(define-object iso-curve-example (base-object)
  :computed-slots
  ((parameter 0.5))

  :objects
  ((surface :type 'surf::test-b-spline-surface)
   (iso-curve :type 'iso-curve
              :strings-for-display "Iso at 0.5"
              :display-controls (list :color :red :line-thickness 3)
              :surface (the surface)
              :parameter (the parameter))))


(generate-sample-drawing :object-roots (list (make-object 'iso-curve-example))
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.35 Joined-surfaces

Description

This routine joins two surfaces at a common boundary. The surfaces must already be compatible in the direction of the common boundary (same knots). If the surfaces are not compatible you can use first compatible-surfaces if applicable

Required-input-slots

other-surface *gdl surface object*

The second surface to be joined Its u-min or v-min lays at the common boundary

surface *gdl surface object*

The first surface to be joined Its u-max or v-max lays at the common boundary

Optional-input-slots

direction *keyword symbol, one of :u or :v*

If :u the common boundary is for first surface u-max and for the second surface u-min Surfaces must already be compatible in the u-direction If :v the common boundary is for first surface v-max and for the second surface v-min Surfaces must already be compatible in the v-direction Default is :u

tolerance *number*

This is a tolerance used for Knot removal The knot corresponding to the merged boundary has multiplicity equal to the degree Knot removal will be attempted using this tolerance Default is *3d-tolerance-default*

Examples

```
(in-package :gdl-user)

(define-object join-surfaces-test (base-object)

  :computed-slots ((surface-list (list (the surf-A) (the surf-B))))

  :objects
  ((surf-A :type 'rectangular-surface
           :display-controls (list :color :green-spring-medium)
           :length 10
           :width 10 )

   (surf-B :type 'rectangular-surface
           :display-controls (list :color :red)
           :center (make-point 10 0 0 )
           :length 10
           :width 10 )

   (join-A-and-B :type 'joined-surfaces
                 :display-controls (list :line-thickness 2)
                 :surface (the surf-A)
                 :other-surface (the surf-B))))


(generate-sample-drawing :object-roots
 (list (the-object (make-object 'join-surfaces-test)
   join-A-and-B)))
```

7.0.4.36 Linear-curve

Description

A GDL NURBS Curve representing a straight line segment. The inputs are the same as for l-line, namely `start` and `end` (3d points).

Optional-input-slots

end *3d point*

   The end point of the line, in global coordinates

start *3d point*

   The start point of the line, in global coordinates

Examples

```
(in-package :surf)
```

```
(define-object test-linear-curve (linear-curve)
  :computed-slots ((start (make-point 0 0 0)) (end (make-point 10 10 0))))

(generate-sample-drawing :objects (make-object 'test-linear-curve))
```

7.0.4.37 Lofted-surface

Description

Loft of a surface through a list of curves with various controls.

Required-input-slots

curves *list of gdl curve objects*

The curves through which the surface will be lofted

Optional-input-slots

rail-1 *gdl curve object*

Guide rail corresponding to minimum U parameter of resulting surface Defaults to nil

rail-1-is-spine? *boolean*

If specified as non-nil, then the rail-1 will be used as a spine curve similar to what is described on page 462 of the NURBS book Default is nil

rail-1-params *list of curve parameters*

The parameter value on the rail-1 which should correspond to each respective profile curve Defaults to the evenly spaced parameters between the u-min and u-max of the rail-1, one for each profile curve

rail-2 *gdl curve object*

Guide rail corresponding to maximum U parameter of resulting surface Defaults to nil If both rail-1 and rail2 are given, then they must be synchronized

spine *gdl curve object*

Curve to use as spine for calling ascssk

synchronized? *boolean*

Set this to t if the curves already have synchronized control points It makes the lofted-surface much lighter-weight in terms of its control mesh Default is NIL (which means the lofted-surface does not assume synchronized control points on the profile curves)

tolerance *number*

The fitting tolerance to fit to the loft curves 0 means to interpolate exactly Default is 0

use-ascssk? *boolean*

If non-nil, we use the low-level nlib ascssk directly If nil, we use the SMLib make-skinned-surface routine Default is nil

v-degree *integer*

The degree of the surface in the lofting direction Defaults to 3

Examples

```
(in-package :surf)

 (define-object test-lofted-surface (base-object)

 :input-slots
 ((curves (list (the curve-1) (the curve-2) (the curve-3) (the curve-4)  )))

 :objects
 ((lofted-surface :type 'lofted-surface
                  :curves (the curves))

  (curve-1 :type 'b-spline-curve
           :display-controls (list :color :red :line-thickness 3)
           :control-points (list (make-point 0 0 0)
                                  (make-point 1 1 0)
                                  (make-point 0 1 0)
                                  (make-point 0 0 0) ))

   (curve-2 :type 'b-spline-curve
            :display-controls (list :color :red :line-thickness 3)
            :control-points (list (make-point 0 0 1)
                                   (make-point -1 1 1)
                                   (make-point 0 1 1)
                                   (make-point 0 0 1) ))

   (curve-3 :type 'b-spline-curve
            :display-controls (list :color :red :line-thickness 3)
            :control-points (list (make-point 0 0 2)
                                   (make-point -1 -1 2)
                                   (make-point 0 -1 2)
                                   (make-point 0 0 2) ))

   (curve-4 :type 'b-spline-curve
            :display-controls (list :color :red :line-thickness 3)
            :control-points (list (make-point 0 0 3)
                                   (make-point 1 -1 3)
                                   (make-point 0 -1 3)
                                   (make-point 0 0 3) ))))

  (generate-sample-drawing :object-roots (make-object 'test-lofted-surface)
                           :projection-direction :trimetric)
```

7.0.4.38 Manifold-solid

Description

!!! Not applicable for this object !!!

Required-input-slots

brep *gdl brep object*

The brep to be represented as a manifold brep in this instance

Optional-input-slots

keep-internal-faces? *boolean*

Indicates whether faces between two non-void regions should be kept Defaults to nil

7.0.4.39 Merged-solid

Description

Given two brep solids or a brep solid and an open face represented as a brep, performs a merge operation. Optionally (with make-manifold? t) makes the result manifold by trimming and throwing away extra pieces of faces and edges.

Optional-input-slots

make-manifold? *boolean*

Indicates whether the resulting brep should be made into a manifold brep, with one or more regions

sew-and-orient? *boolean*

Indicates whether we should try to sew and orient the resulting brep Usually a good idea and this is defaulted to t, except for merged-solid where we default this to nil

7.0.4.40 Native-reader

Description

This object will reflect the contents of an iwp file containing curves, surfaces, breps, and brep trees as sequences of GDL objects.

Optional-input-slots

file-name *string or pathname*

   The location of the IWP file to be read

smlib-string *string*

   Contains output from a call to (with-format (native) (write-the cad-output)) for an SMLib object (e g curve, surface, brep) Defaults to nil If you specify this as well as a file-name, the file-name will take precedence

Quantified-objects

breps *sequence of gdl brep objects*

   The breps found in the IGES file

curves *sequence of gdl curve objects*

   The curves found in the IWP file

surfaces *sequence of gdl surface objects*

   The untrimmed "standalone" surfaces found in the IWP file

7.0.4.41 Normalized-curve

Description

   This object creates a new curve from an exiting curve by reasigning the lowest and highest parameter value of the underlying mathematical definition of the curve. This is a precise method, it does not change the curve geometry.

Optional-input-slots

curve-in *gdl nurbs curve*

   The curve to be normalized

tolerance *number*

   Approximation tolerance for display purposes Defaults to the tolerance of the built-from curve, if one exists, otherwise defaults to the *display-tolerance*

u-max *the highest parameter value of the underlying mathematical definition for this parametric curve*

u-min *the lowest parameter value of the underlying mathematical definition for this parametric curve*

7.0.4.42 Offset-solid

Description

!!! Not applicable for this object !!!

Required-input-slots

brep *gdl brep object*

The brep to be offset

distance *number*

The distance to offset Can be negative

Optional-input-slots

tolerance *number*

The tolerance to use for the shelling operation Defaults to (the adaptive-tolerance) of the input brep

7.0.4.43 Offset-surface

Description

!!! Not applicable for this object !!!

Required-input-slots

distance *number*

The distance to offset Positive or negative, depending on which direction you want

surface-in *gdl surface object*

The original surface from which to make the offset

Optional-input-slots

approximation-tolerance *number*

The tolerance of approximation for the re-fitting of points after the offsetting Defaults to *3d-approximation-tolerance-default*

parameterization *keyword symbol, one of :uniform, :chord-length, :centripetal, or :inherited*

The parameterization method used to re-fit the points after offsetting Defaults to :uniform

u-degree *integer*

The desired u-degree of the resulting surface Defaults to the u-degree of the input surface-in

v-degree *integer*

The desired v-degree of the resulting surface Defaults to the v-degree of the input surface-in

7.0.4.44 Planar-offset-curve

Description

Creates a curve which is the result of offsetting a curve by its normals along a plane.

Required-input-slots

curve-in *gdl curve*

The curve to be offset

distance *number*

The left-hand distance to offset with respect to curve direction To get the opposite direction, you can either negate this number or reverse the `plane-normal`

plane-normal *3d vector*

The normal for the plane

Optional-input-slots

tolerance *number*

The tolerance for approximating the resulting offset curve Defaults to *3d-approximation-tolerance-default*

Examples

```
(in-package :surf)

(define-object test-planar-offset-curve ()
 :computed-slots
 ((curve-in (the b-spline-curves (curves 3)))

  (plane-normal (make-vector 0 0 -1))

  (distance 1) )
```

```
    :objects
    ((b-spline-curves :type 'test-b-spline-curves
                      :hidden? t)

     (curve-to-be-offset :type 'curve
                         :built-from (the b-spline-curves (curves 3)))


     (planar-offset-curve :type 'surf:planar-offset-curve
                          :curve-in (the curve-to-be-offset)
                          :plane-normal (make-vector 0 0 -1)
                          :distance 1)))

    (generate-sample-drawing :object-roots (list (make-object 'test-planar-offset-curve))
                             :projection-direction (getf *standard-views* :top))
```

7.0.4.45 Planar-section-curve

Description

Produces a single curve by sectioning a surface with a plane. If multiple results are expected, use planar-section-curves instead.

Required-input-slots

surface *gdl surface, face, or trimmed surface*
    The surface to be sectioned with a plane

Optional-input-slots

3d-approximation-tolerance *number*
    Tolerance used when approximating in e g Newton-Raphson iterations Default is *3d-approximation-tolerance-default*

angle-tolerance-radians *number*
    Angular tolerance (in radians) used when approximating in e g Newton-Raphson iterations Default is *angle-tolerance-radians-default*

plane-normal *vector*
    The normal of the sectioning plane Defaults to the top vector of the local reference box

plane-point *3d point*
    A point on the sectioning plane Defaults to the center

Computed-slots

on-surfaces *list of gdl surfaces*

> The surfaces on which this curve lies

success? *boolean*

> This will be non-nil if the curve was generated successfully

Functions

uv-curve *gdl curve object*

> The UV curve for this curve in the context of the surface.
>
> *Optional Arguments:*
>
> **surface** (the surface), Default Value: `nil`,

Examples

```
(in-package :surf)

(define-object test-planar-section-curve (base-object)

 :objects
 ((planar-section-curve :type 'planar-section-curve
                        :surface (the test-surf)
                        :plane-normal (the (face-normal-vector :top))
                        :plane-point (make-point 0 0 2)
                        :display-controls (list :color :red :line-thickness 4))

  (test-surf :type 'test-fitted-surface )))

(generate-sample-drawing :object-roots (list (make-object 'test-planar-section-curve))
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.46 Planar-section-curves

Description

Produces multiple curves by sectioning a surface or a brep with a plane. If a single result is expected, use planar-section-curve instead.

Optional-input-slots

3d-approximation-tolerance *number*

Tolerance used when approximating in e g Newton-Raphson iterations Default is *3d-approximation-tolerance-default*

angle-tolerance-radians *number*

Angular tolerance (in radians) used when approximating in e g Newton-Raphson iterations Default is *angle-tolerance-radians-default*

brep *gdl brep object*

The brep to be sectioned with a plane Specify this or surface, not both

plane-normal *vector*

The normal of the sectioning plane Defaults to the top vector of the local reference box

plane-point *3d point*

A point on the sectioning plane Defaults to the center

surface *gdl surface object*

The surface to be sectioned with a plane Specify this or brep, not both

Computed-slots

3d-approximation-tolerance-achieved *number*

The actual tolerance achieved by the operation

angle-tolerance-radians-achieved *number*

The actual angle tolerance achieved by the operation

Quantified-objects

curves *sequence of gdl curve objects*

The curves resulting from sectioning

uv-curves *sequence of gdl uv curve objects*

The UV curves for each returned curve This is also passed into each curve object and available from there

Examples

```
(in-package :surf)

(define-object test-planar-section-curves (base-object)

 :computed-slots
 ((points-data '(((0 0 0)(0 1 0)(1 1 0)(1 0 0))
                 ((0 0 1) (0 1 1) (1 1 1) (1 0 1) )
                 ((0 0 2) (0 1 2) (1 1 2) (1 0 2) )
                 ((0 0 3) (0 1 3) (1 1 3) (1 0 3) )))

  (control-points (mapcar #'(lambda(list) (mapcar #'apply-make-point list))
                          (the points-data))))

  :objects
```

```
((planar-section-curve :type 'planar-section-curves
                       :surface (the test-surf)
                       :plane-normal (the (face-normal-vector :front))
                       :plane-point (make-point 0 0.5 0)
                       :display-controls (list :color :red :line-thickness 4))

 (test-surf :type 'b-spline-surface
            :control-points (the control-points)) ))

(generate-sample-drawing :object-roots
                         (list (make-object 'test-planar-section-curves))
                          :projection-direction
                          (getf *standard-views* :trimetric))
```

7.0.4.47 Planar-surface

Description

Creates a flat quadrilateral surface specified by its four corner points.

Required-input-slots

p00 *3d point*

Front-left corner of the planar surface

p01 *3d point*

Front-right corner of the planar surface

p10 *3d point*

Rear-left corner of the planar surface

p11 *3d point*

Rear-right corner of the planar surface

Examples

```
(in-package :surf)

(define-object test-planar-surface (planar-surface)
  :computed-slots
  ((display-controls (list :color :fuchsia :transparency 0.5))
   (p00 (make-point 0 0 0))
   (p01 (make-point 0 1 0))
   (p10 (make-point 1 0 0))
   (p11 (make-point 1.5 1.5 0))))

 (generate-sample-drawing :objects (make-object 'test-planar-surface)
                          :projection-direction :trimetric)
```

7.0.4.48 Projected-curve

Description

Creates a 3D curve which is the `curve-in` projected onto the `surface.` according to the `projection-vector`. The resulting curve contains a uv-curve which is typically useful for trimming.

NOTE: Change from 1577p027 and forward – the projected curve now is a 3D curve which can be drawn. It contains its uv representation on the surface. Previously, the uv-curve was the actual projected-curve.

Required-input-slots

curve-in *gdl nurbs curve*

The curve to be projected to the surface

projection-vector *3d vector*

The direction of projection

surface *gdl nurbs surface*

The surface on which the `curve-in` is to be projected

Optional-input-slots

angle-tolerance-radians *number or nil*

The angle tolerance used when projecting and creating new curves Defaults to nil, which uses the default of the geometry kernel

approximation-tolerance *number or nil*

The tolerance used when projecting and creating new curves Defaults to nil, which uses the default of the geometry kernel

Computed-slots

built-from *gdl curve*

Specify this if you want this curve to be a clone of an existing curve (note - this uses a shared underlying curve object, it does not make a copy)

Hidden-objects

uv-curve *gdl uv curve object*

    The resultant projected-curve in the UV space of the surface

Examples

```
(in-package :surf)

(define-object test-trimmed-from-projected-2 (trimmed-surface)
 :computed-slots
 ((uv-inputs t)
  (island (the island-3d uv-curve))
  (holes (list (the hole uv-curve)))
  (display-controls (list :color :blue :line-thickness 2)))

 :objects
 ((basis-surface :type 'test-fitted-surface
                 :display-controls (list :color :pink)
                 :grid-length 10 :grid-width 10 :grid-height 5
                 )

  (raised-hole :type 'b-spline-curve
               :display-controls (list :color :grey-light-very)
               :control-points (list (make-point 3.5 4.5 7)
                                      (make-point 4.5 6 7)
                                      (make-point 5.5 7 7)
                                      (make-point 6 4.5 7)
                                      (make-point 5.5 2 7)
                                      (make-point 4.5 2 7)
                                      (make-point 3.5 4.5 7)))

  (raised-island :type 'b-spline-curve
                 :display-controls (list :color :grey-light-very)
                 :control-points (list (make-point 3 5 7)
                                        (make-point 5 8 7)
                                        (make-point 7 10 7)
                                        (make-point 8 5 7)
                                        (make-point 7 0 7)
                                        (make-point 5 0 7)
                                        (make-point 3 5 7))))
 :hidden-objects
 ((island-3d :type 'projected-curve
             :curve-in (the raised-island)
             :surface (the basis-surface)
             :projection-vector (make-vector 0 0 -1))


  (hole :type 'projected-curve
        :curve-in (the raised-hole)
        :surface (the basis-surface)
        :projection-vector (make-vector 0 0 -1))))


(generate-sample-drawing
 :objects (let ((self (make-object 'test-trimmed-from-projected-2)))
            (list (the basis-surface) self (the raised-island)
                  (the raised-hole)))
 :projection-direction :trimetric)
```

7.0.4.49 Rectangular-surface

Description

Creates a flat rectangular surface specified by the same inputs as box or base-object.

Required-input-slots

height *number*

Z-axis dimension of the reference box Defaults to zero

length *number*

Y-axis dimension of the reference box Defaults to zero

width *number*

X-axis dimension of the reference box Defaults to zero

Defaulted-input-slots

center *3d point*

Indicates in global coordinates where the center of the reference box of this object should be located

Computed-slots

p00 *3d point*

Front-left corner of the planar surface

p01 *3d point*

Front-right corner of the planar surface

p10 *3d point*

Rear-left corner of the planar surface

p11 *3d point*

Rear-right corner of the planar surface

Examples

```
(in-package :surf)
```

```
(define-object test-rectangular-surface (rectangular-surface)
  :computed-slots ((display-controls (list :color :green-spring-medium))
                   (length 20) (width 30) (height 0)))

(generate-sample-drawing :objects (make-object 'test-rectangular-surface)
                         :projection-direction :trimetric)
```

7.0.4.50 Regioned-solid

Description

Given a brep solid that contains multiple regions, splits the regions into separate breps

Required-input-slots

brep *gdl brep object or object containing a brep*

The multi-region brep to be split

Optional-input-slots

section-colors *list of color keywords*

These indicate the colors for any child breps if the regioning operation results in multiple solids Defaults to a repeating (circular) list with keys:

- :green
- :red
- :blue
- :purple-dark
- :violet
- :cyan

7.0.4.51 Revolved-surface

Description

Creates a surface of revolution based on an arbitrary NURBS curve revolved by some angle about a central axis and axis point.

Required-input-slots

curve *gdl curve object*

The profile to be revolved

Optional-input-slots

arc *angle in radians*
The amount to revolve Default is twice pi (a full circle of revolution)

axis-point *3d point*
The center of revolution Default value is the `center`

axis-vector *3d vector*
The direction of axis of revolution Default is the top of the reference box

Defaulted-input-slots

center *3d point*
Indicates in global coordinates where the center of the reference box of this object should be located

Examples

```
(in-package :surf)

(define-object test-revolved-surface (revolved-surface)

  :computed-slots ((display-controls (list :color :green-yellow2)))

  :hidden-objects
  ((curve :type 'arc-curve
          :center (translate (the center) :right 50)
          :orientation (alignment :top (the (face-normal-vector :rear)))
          :start-angle (half pi)
          :end-angle (* 3/2 pi)
          :radius 10)))

(generate-sample-drawing :objects (make-object 'test-revolved-surface)
                         :projection-direction :trimetric)
```

7.0.4.52 Revolved-surfaces

Description
Creates a set of surfaces of revolution based on a list of arbitrary NURBS curves revolved by some angle about a central axis and axis point.

Required-input-slots

curves *list of gdl curve objects*
    The profiles to be revolved

Optional-input-slots

arc *angle in radians*
    The amount to revolve Default is twice pi (a full circle of revolution)

axis-point *3d point*
    The center of revolution Default value is the `center`

axis-vector *3d vector*
    The direction of axis of revolution Default is the top of the reference box

Quantified-objects

surfaces *sequence of gdl surfaces*
    The resultant revolved surfaces

Examples

```
(in-package :gdl-user)

(define-object test-revolved-surfaces (revolved-surfaces)

  :computed-slots ((curves (list (the curve-1) (the curve-2))))

  :hidden-objects
  ((curve-1 :type 'arc-curve
            :center (translate (the center) :right 50)
            :orientation (alignment :top (the (face-normal-vector :rear)))
            :start-angle 0
            :end-angle (/ pi 4)
            :radius 10)

   (curve-2 :type 'arc-curve
            :center (translate (the center) :right 50)
            :orientation (alignment :top (the (face-normal-vector :rear)))
            :start-angle pi
            :end-angle (* 5/4 pi)
            :radius 10)

   (view :type 'base-view
         :projection-vector (getf *standard-views* :trimetric)
         :page-width (* 5 72) :page-length (* 5 72)
         :object-roots (list self))))

(generate-sample-drawing
 :objects (list-elements (the-object (make-object 'test-revolved-surfaces)
                                     surfaces))
 :projection-direction :trimetric)
```

7.0.4.53 Ruled-surface

Description

Creates a surface between two NURBS curves.:

Required-input-slots

curve-1 *gdl curve object*

First boundary of the ruled surface

curve-2 *gdl curve object*

Second boundary of the ruled surface

Optional-input-slots

direction *keyword symbol, either :u or :v*

The direction of parameterization of the surface between the two curves

Examples

```
(in-package :surf)

(define-object test-ruled-surface (ruled-surface)

  :computed-slots ((display-controls (list :color :orchid-dark)))

  :hidden-objects
  ((curve-1 :type 'linear-curve
            :start (make-point -1 -1 0)
            :end (make-point -1 1 0))

   (curve-2 :type 'fitted-curve
            :points (list (make-point 1 -1 0) (make-point 0 0 0)
                          (make-point .5 .5 0) (make-point 1 2 0)))))


(generate-sample-drawing :objects (make-object 'test-ruled-surface)
                         :projection-direction :trimetric)
```

7.0.4.54 Separated-solid

Description

Given two brep solids or a brep solid and an open face represented as a brep, performs a split operation

Optional-input-slots

cap-results? *boolean*

Indicates whether the resulting split pieces should be made into watertight solids (ends capped, etc)

section-colors *list of color keywords*

These indicate the colors for any child breps if the boolean operation results in a separated solid If the number of breps exceeds the length of this list, the colors will be repeated in order Defaults to a list with keys:

- :green
- :red
- :blue
- :purple-dark
- :violet
- :cyan

Quantified-objects

breps *sequence of gdl brep objects*

The resulting breps yielded from the separate operation These are colored using section-colors

7.0.4.55 Shelled-solid

Description

!!! Not applicable for this object !!!

Required-input-slots

brep *gdl brep object*

Should be an open shell The brep to be shelled into a solid

distance *number*

The distance to offset Can be negative

Optional-input-slots

tolerance *number*

The tolerance to use for the shelling operation Defaults to (the adaptive-tolerance) of the input brep

7.0.4.56 Spherical-surface

Description

A surface representation of the sphere. Takes the same inputs as native GDL sphere. Partial spheres are not yet implmented. Note that some VRML browsers, e.g. Cortona v. 4.2, show some spurious artifacts with NURBS created as spherical surfaces. BS Contact does not appear to have this problem.

Defaulted-input-slots

center *3d point*

Indicates in global coordinates where the center of the reference box of this object should be located

Examples

```
(in-package :surf)

(define-object test-spherical-surface (spherical-surface)
  :computed-slots ((display-controls (list :color :sky-summer))
                   (radius 10)))

(generate-sample-drawing :objects (make-object 'test-spherical-surface)
                         :projection-direction :trimetric)
```

7.0.4.57 Split-surface

Description

Given a NURBS and a parameter in U or V direction, split the surface at the parameter and return one section or the other as the toplevel self of this instance. Note that both resulting sections are also reflected in (the surfaces) sequence which is a hidden child in this object.

As an alternative to a parameter, a projection-point and projection-vector can also be given, and the U or V parameter at the resulting surface point will be used as the parameter.

Optional-input-slots

keep-side *keyword symbol, one of :left or :right*

Determines which half of the split surface to reflect in this instance Both halves will be reflected in (the surfaces) hidden-object sequence which is a child of this instance

parameter *number*

The parameter in U or V direction at which to do the split This must lie in the domain between (the u-min) [or (the v-min)] and (the u-max) [or (the v-max)] of the surface-in If it is outside this domain, a warning will be thrown and the value will be pinned to the nearest value within the domain

If this input is not specified and you specify a projection-point and projection-vector, then this projected point will be used to establish the parameter for splitting

projection-point *3d point or nil*

If given and parameter is not given, this point will be projected onto the surface using (the projection-vector) to establish the split parameter Defaults to nil

projection-vector *3d vector or nil*

If given and parameter is not given, this will be used to project (the projection-point) onto the surface to establish the split parameter Defaults to nil

surface-in *gdl surface object*

The surface to be split

u-or-v *keyword symbol, one of :u or :v*

Determines the direction of the split

Computed-slots

projected-point *surface point*

Returns the first result of the given point projected along the given vector intersected with the surface

:arguments (point "3D Point The point to be projected " vector "3D Vector The vector along which to project ")

projected-points *list of surface points*

Returns the given point projected along the given vector intersected with the surface

:arguments (point "3D Point The point to be projected " vector "3D Vector The vector along which to project ")

Examples

```
(in-package :gdl-user)

(define-object test-split-surface (base-object)

  :computed-slots ((projection-point (make-point 10 0 3))
                   (projection-vector (make-vector 1 0 0))
                   (u-or-v :u :settable)
```

```
                          (keep-side :left :settable))

      :objects
      ((test-surface :type 'test-b-spline-surface
                     :display-controls nil)


       (split :type 'split-surface
              :display-controls (list :color :red :line-thickness 3)
              :surface-in (the test-surface)
              :pass-down (keep-side u-or-v projection-point projection-vector))))


   (generate-sample-drawing :object-roots (make-object 'test-split-surface)
                              :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.58 Step-reader

Description

This object will reflect the contents of a STEP file containing points, curves, surfaces, and/or trimmed surfaces as sequences of GDL objects. Currently all surfaces are treated as trimmed, where actual untrimmed surfaces have their natural outer boundaries as the result-island, i.e. no standalone surfaces will be produced by this part.

This is a default option in the HarmonyWare STEP Translator which will be exposed in GDL in a future release.

The HarmonyWare reader creates a log file in a temporary directory. The location of this log file is printed on the console during the reading operation. Currently this log file is not automatically deleted, and its name is determined by the system.

Required-input-slots

file-name *string or pathname*

The location of the STEP file to be read

Optional-input-slots

group-trimmed-surfaces-into-brep? *boolean*

If true, group all trimmed surfaces in the file into one B-rep If some trimmed surfaces are blanked, they are grouped into a second, blanked B-rep Default is nil

make-all-surfaces-trimmed? *boolean*

If true, treat all untrimmed surfaces in the file as if they are trimmed surfaces with the natural outer boundary of the surface as the trimming loop If used, no standalone IwSurface surface objects will ever be returned by the reader Default is nil

make-single-brep? *boolean*

If true, group all trimmed surfaces and B-reps in the file into one B-rep If some trimmed surfaces or B-reps are blanked, they are grouped into a second, blanked B-rep Default is nil

sew-brep-faces? *boolean*

Indicates whether each resulting brep should have its faces sewn together Default is (the make-single-brep?)

Quantified-objects

breps *sequence of gdl brep objects*

The breps and trimmed surfaces (represented as breps) found in the STEP file

curves *sequence of gdl curve objects*

The curves found in the STEP file

points *sequence of gdl point objects*

The points found in the STEP file

surfaces *sequence of gdl surface objects*

The untrimmed "standalone" surfaces found in the STEP file

7.0.4.59 Stitched-solid

Description

!!! Not applicable for this object !!!

Required-input-slots

faces-in *list of gdl surface or face objects*

These will be stitched together into an open shell or possibly a Solid

7.0.4.60 Subtracted-solid

Description

Given two brep solids, performs the subtract Boolean of the other-brep from the brep

7.0.4.61 Surface

Description

A generalized NURBS surface. Usually used as a mixin in more specific surfaces.

Optional-input-slots

brep-tolerance *number*

Overall tolerance for the internal brep representation of this surface Defaults to nil Note that a value of nil indicates for SMLib a value of 1 0e-05 of the longest diagonal length of the brep

built-from *gdl surface*

Specify this if you want this surface to be a clone of an existing surface (note - this uses a shared underlying surface object, it does not make a copy)

end-caps-on-brep? *boolean*

Indicates whether to attempt automatic endcaps on conversion of this surface to a brep Note that this might change in future to a keyword value for :min, :max, or :both to provide more control

rational? *boolean*

Returns non-nil iff this surface is rational, i e all weights are not 1

sew-and-orient-brep? *boolean*

Indicates whether brep representation should undergo a sew-and-orient operation Defaults to nil

tessellation-parameters *plist of keyword symbols and numbers*

This controls tessellation for this brep The keys are as follows:
- :min-number-of-segments
- :max-3d-edge-factor
- :min-parametric-ratio
- :max-chord-height
- :max-angle-degrees
- :min-3d-edge
- :min-edge-ratio-uv
- :max-aspect-ratio

and the defaults come from the following parameters:
- (list
- :min-number-of-segments *tess-min-number-of-segments*
- :max-3d-edge-factor *tess-max-3d-edge-factor*
- :min-parametric-ratio *tess-min-parametric-ratio*
- :max-chord-height *tess-max-chord-height*
- :max-angle-degrees *tess-max-angle-degrees*

- :min-3d-edge *tess-min-3d-edge*
- :min-edge-ratio-uv *tess-min-edge-ratio-uv*
- :max-aspect-ratio *tess-max-aspect-ratio*)

tolerance *number*

    Approximation tolerance for display purposes

Defaulted-input-slots

isos *plist with keys :n-u and :n-v*

    The number of isoparametric curves to be displayed in each direction This value comes from the value of :isos on the display-controls if that exists, and defaults to *isos-default* otherwise

Computed-slots

bounding-box *list of two 3d points*

    The left front bottom and right rear top corners, in global coordinates, of the rectangular volume bounding the tree of geometric objects rooted at this object

height *number*

    Z-axis dimension of the reference box Defaults to zero

length *number*

    Y-axis dimension of the reference box Defaults to zero

width *number*

    X-axis dimension of the reference box Defaults to zero

Hidden-objects

brep *gdl brep object*

    This is the brep representation of this surface

swapped-uv-surface *gdl surface object*

    This surface object swaps the role of u and v- directional parameters, i e old-surface(u,v) = new-surface(v,u)

Quantified-hidden-objects

u-iso-curves *sequence of curve objects*

    The isoparametric curves in the U direction

v-iso-curves *sequence of curve objects*

The isoparametric curves in the V direction

Functions

area *number*

Returns the area of the surface.

Keyword Arguments:

desired-accuracy Number, Default Value: 10.0e-7, Desired accuracy of result. Should not be smaller than 0.00000001 (10e-08).

List of lists 3D points, List of lists numbers, List of numbers, List of numbers, Integer, and Integer. Returns six values which are the control points, the weights, the u-knots, the v-knots, the u-degree, and the v-degree of the surface.

curve-intersection-point *surface point*

Returns the first point of intersection between this surface and the curve given as an argument.

Keyword Arguments:

3d-tolerance Number, Default Value: *3d-tolerance-default*, The tolerance to use for intersecting.

curve-intersection-points *list of surface points*

Returns the point(s) of intersection between this surface and the curve given as an argument.

Keyword Arguments:

3d-tolerance Number, Default Value: *3d-tolerance-default*, The tolerance to use for intersecting.

domain *plist*

Returns plist containing :min and :max indicating min and max UV points for parameter space for this surface.

dropped-point *surface point*

Returns the given 3D point dropped normally to this surface, as close as possible to the given 3D point.

Keyword Arguments:

3d-tolerance The tolerance used for dropping, Default Value: *3d-tolerance-default*,

dropped-points *list of surface points*

Returns the given 3D point dropped normally to this surface.

Keyword Arguments:

3d-tolerance The tolerance used for dropping, Default Value: *3d-tolerance-default*,

Returns a bbox object, answering xmin, ymin, zmin, xmax, ymax, and zmax, for a box containing the convex hull (i.e. the control points) of this surface and oriented according to the given `center` and `orientation`.

maximum-distance-to-curve *plist*

The returned plist contains information about the maximum distance from this surface to the curve given as the argument.

maximum-distance-to-surface *plist*

The returned plist contains information about the maximum distance from this surface to the surface given as the argument.

minimum-distance-to-curve *plist*

The returned plist contains information about the minimum distance from this surface to the curve given as the argument.

minimum-distance-to-surface *plist*

The returned plist contains information about the minimum distance from this surface to the surface given as the argument.

normal *3d vector*

The surface normal vector at the given u and v values. Three other values are also returned: The 3D point, the U tangent, and the V tangent at the given parameter value.

offset-point *3d point*

Returns the surface point at the given parameters offset along the surface normal at that point by the given distance.

on? *boolean*

Returns non-nil if the given UV (2D) point lies within the parameter space of this surface. Currently this function works only on the basis surface ; it does not observe trimming island or holes.

point *3d point*

The point on the surface corresponding to the given u and v parameter values.

projected-point *surface point*

Returns the first result of the given point projected along the given vector intersected with the surface.

projected-points *list of surface points*

Returns the given point projected along the given vector intersected with the surface.

radius-of-curvature *number*

Returns the Gaussian curvature on the surface at the given parameter values. Three additional values are returned, which are the Normal Curvature at the point, the first Fundamental Principle Curvature, and the second Fundamental Principle Curvature.

u-max *number*

Returns maximum U component of the surface parameter space.

u-min *number*

Returns minimum U component of the surface parameter space.

v-max *number*

Returns maximum V component of the surface parameter space.

v-min *number*

Returns minimum V component of the surface parameter space.

7.0.4.62 Surface-knot-reduction

Description

This routine removes all removable knots from a GDL surface.

Optional-input-slots

direction *keyword symbol, one of :u, :v or :uv*

Default is :uv

surface *gdl surface object*

tolerance *number*

Examples

```
(in-package :gdl-user)

(define-object surface-knot-reduction-test (base-object)

 :input-slots
 ((control-points (list (make-point 0 0 0)
                        (make-point 2 3.0 0.0)
                        (make-point 4 2.0 0.0)
                        (make-point 5 0.0 0.0)
                        (make-point 4 -2.0 0.0)
                        (make-point 2 -3.0 0.0)
                        (make-point 0 0 0))))

 :objects
 ((curve-1 :type 'b-spline-curve
           :display-controls (list :line-thickness 2
                                     :color :green-spring-medium)
           :control-points (the control-points))

  (curve-2 :type 'boxed-curve
           :display-controls (list :line-thickness 2 :color :blue)
           :curve-in (the curve-1)
           :center (make-point 0 0 8))

  (curve-3 :type 'transformed-curve
           :display-controls (list :line-thickness 2 :color :green)
           :curve-in (the curve-1)
                     :to-location (translate
                                    (the center)
                                    :up 3)
```

```
                                  :center (the center)
                                  :scale-x 1.3
                                  :scale-y 1.3)

      (curve-4 :type 'transformed-curve
               :display-controls (list :line-thickness 2 :color :red)
               :curve-in (the curve-1)
                              :to-location (translate
                                               (the center)
                                               :up 7)
                              :center (the center)
                              :scale-x 2.2
                              :scale-y 2.2)

      (lofted-surface-test-simple :type 'lofted-surface
                                  :display-controls (list :color :red-violet
                                                          :isos (list :n-v 19
                                                                      :n-u 19))
                                  :tolerance 0.01
                                  :curves (list (the curve-1) (the curve-3)
                                                (the curve-4) (the curve-2)))

      (S-knot-reduction :type 'surface-knot-reduction
                        :surface (the lofted-surface-test-simple))))
```

7.0.4.63 Swept-solid

Description

This primitive will take a brep as input, and sweep all its faces in the given direction by the given distance, to produce another brep.

Required-input-slots

distance *number*

The distance over which the sweep is desired

facial-brep *gdl brep object*

The original brep, which can contain one or more faces, planar and/or non-planar

vector *gdl vector*

The direction in which the sweep is desired

Examples

```
      (in-package :gdl-user)

      (define-object swept-solid-example (swept-solid)

        :computed-slots
```

```
((facial-brep (the trimmed brep))
 (vector (make-vector 0 0 1))
 (distance 10)
 (display-controls (list :isos (list :n-u 8 :n-v 8) :color :blue :transparency 0.3)))


:hidden-objects
((rectangle :type 'rectangular-surface
            :width 20 :length 20)

 (trim-curve :type 'global-filleted-polyline-curve
             :vertex-list (list (translate (the center) :right 8 :rear 8)
                                (translate (the center) :left 8 :rear 8)
                                (translate (the center) :left 8 :front 8)
                                (translate (the center) :right 8 :front 8)
                                (translate (the center) :right 8 :rear 8))
             :default-radius 3)

 (trimmed :type 'trimmed-surface
          :basis-surface (the rectangle)
          :reverse-island? t
          :island (the trim-curve))))

(generate-sample-drawing :objects (make-object 'swept-solid-example)
                         :projection-direction (getf *standard-views* :trimetric))
```

7.0.4.64 Transformed-solid

Description

This primitive Translates, Orients, and optionally Scales a brep solid into another brep solid.

Required-input-slots

brep *gdl brep object*

Source Brep to be copied and transformed

Optional-input-slots

from-location *3d point*

Reference location from which to translate Defaults to the from-object center

from-object *gdl object*

Reference Object for from-location and from-orientation Defaults to the given brep

from-orientation *3x3 orientation matrix or nil*

Defaults to the from-object's orientation

scale *3d vector or nil*

    Scale to be applied before transform in each axis, or nil if no scale to be applied

sew-and-orient? *boolean*

    Controls whether to do the shrink cleanup step after the tranform Defaults to nil

shrink? *boolean*

    Controls whether to do the shrink cleanup step after the tranform Defaults to nil

to-location *3d point*

    Reference location to which to translate Defaults to the from-object center

to-orientation *3x3 orientation matrix or nil*

    Target orientation relative to the from-orientation Defaults to nil

7.0.4.65 Trimmed-curve

Description

    Creates a curve based on an existing curve but possibly with new start and end parameters (`u1` and `u2`).

Required-input-slots

built-from *gdl curve object*

    The underlying curve from which to build this curve

Optional-input-slots

u1 *number*

    Specified start parameter Defaults to the `u1` of the built-from

u2 *number*

    Specified end parameter Defaults to the `u2` of the built-from

Computed-slots

basis *gdl curve*

    The original untrimmed curve, same as the `built-from`

Examples

```
(in-package :surf)

(define-object test-trimmed-curve ()

  :objects
  ((b-spline-curve :type 'b-spline-curve
                   :control-points (list (make-point 0 0 0)
                                         (make-point 2 3.0 0.0)
                                         (make-point 4 2.0 0.0)
                                         (make-point 5 0.0 0.0)
                                         (make-point 4 -2.0 0.0)
                                         (make-point 2 -3.0 0.0)
                                         (make-point 0 0 0)))

   (trimmed-curve :type 'trimmed-curve
                  :built-from (the b-spline-curve)
                  :u1 0.2
                  :u2 0.8
                  :display-controls (list :color :red :line-thickness 1.5))))

(generate-sample-drawing :object-roots (make-object 'test-trimmed-curve))
```

7.0.4.66 Trimmed-surface

Description

Creates a surface which is trimmed by outer trimming loop curves (the "island"), and one or more hole curves within the outer trimming loop. The curves can be input either as u-v curves or 3D curves. If 3D curves are given, they must lie on the surface. If not, please use dropped-curve or projected-curve to ensure that the curves lie on the surface.

Note that this object mixes in face, which mixes in surface. So this object should answer all the messages of both face and surface. However, the local surface messages will operate on the basis, not on the trimmed representation. The messages for face will operate on the trimmed representation.

NOTE: the interface for this object is still under development so please stay apprised of any changes.

Required-input-slots

island *single gdl nurbs curve or list of same*

These curves make up the outer trimming loop Normally should be in counter-clockwise orientation; if not, please specify `reverse-island?` as non-NIL

Optional-input-slots

basis-surface *gdl nurbs surface*

The underlying surface to be trimmed

holes *list of gdl nurbs curves or list of lists of gdl nurbs curves*

These curves make up zero or more holes within the outer trimming loop Normally should be in clockwise orientation; if not, please specify `reverse-holes?` as non-NIL

reverse-holes? *boolean*

Specify this as non-NIL if the holes are given in counter-clockwise orientation Default is NIL

reverse-island? *boolean*

Specify this as non-NIL if the island is given in clockwise orientation Default is NIL

uv-inputs *boolean*

NIL if feeding in 3D curves, non-NIL if feeding in UV curves

Computed-slots

result-holes *list of gdl 3d nurbs curves or list of lists of gdl 3d nurbs curves*

These make up the inner holes of the resultant trimmed surface If you specified `holes` as an input-slot, normally these should be the same or very similar If the trimmed surface was read in from an outside system through a translator such as IGES, the `result-holes` might return a non-NIL value while the `island` will always return NIL

result-island *single gdl 3d nurbs curve or list of gdl 3d nurbs curves*

These make up the outer trimming loop of the resultant trimmed surface If you specified `island` as an input-slot, normally these should be the same or very similar If the trimmed surface was read in from an outside system through a translator such as IGES, the `result-island` should return a non-NIL value while the `island` will return NIL

Hidden-objects

brep *gdl brep object*

The Brep containing the face corresponding to this trimmed surface

Examples

```
(in-package :surf)

(define-object test-trimmed-surface-3 (trimmed-surface)
  :computed-slots
  ((reverse-holes? t)
   (island (the island-container ordered-curves))
   (holes (list (the hole ordered-curves)))
   (display-controls (list :color :periwinkle :line-thickness 2)))

  :hidden-objects
```

```
((basis-surface :type 'test-planar-surface
                :display-controls (list :color :grey-light-very))

 (island-container :type 'global-filleted-polyline-curves
                   :default-radius .05
                   :vertex-list (list (make-point 0 0 0)
                                      (make-point 0.3 0.6 0)
                                      (make-point 0 1 0)
                                      (make-point 1 1 0)
                                      (make-point 1 0 0)
                                      (make-point 0 0 0)))

 (island-2 :type 'b-spline-curve
           :control-points (list (make-point 0 0 0)
                                 (make-point 0 1 0)
                                 (make-point 1 1 0)
                                 (make-point 1 0 0)
                                 (make-point 0 0 0)))

 (hole :type 'global-filleted-polyline-curves
       :default-radius .05
       :vertex-list (list (make-point 0.5 0.5 0)
                          (make-point 0.75 0.5 0)
                          (make-point 0.75 0.75 0)
                          (make-point 0.5 0.75 0)
                          (make-point 0.5 0.5 0)))))

(generate-sample-drawing :objects (make-object 'test-trimmed-surface-3)
                         :projection-direction :trimetric)
```

7.0.4.67 United-solid

Description

Given two brep solids, performs the union Boolean between the brep and the other-brep

8 Customer Support

Evaluation and Student Licensees are not entitled to support directly from Genworks. *Please register for the Genworks Google Group then post your questions there.*

For **commercial licensees**, Genworks Customer support is available. Support for all included components of GenDL/GWL can be provided by Genworks as your single point of contact. Our VAR agreements with vendors such as Franz Inc. and SMS stipulate that Genworks' customers contact only Genworks for support, and not e.g. Franz Inc. or SMS directly. As necessary Genworks will follow up with our vendors for second-level support on Allegro CL, SMLib, etc.

Genworks Technical Support can be reached at: `support@genworks.com`*.*