Genworks GDL: A User's Manual

Dave Cooper

December 2017

Contents

Chapter 1

Introduction

1.1 Welcome

Congratulations on your decision to work with Genworks® GDL™[1]. By investing time to learn this system you will be investing in your future productivity and, in the process, you will be joining a quiet revolution. Although you may have come to Genworks GDL because of an interest in 3D modeling or mechanical engineering, you will find that a whole new world, and a unique approach to *computing*, will now be at your fingertips as well.

1.2 Knowledge Base Concepts According to Genworks

You may have an idea about Knowledge Base Systems, or Knowledge *Based* Systems, from college textbooks or corporate marketing literature, and concluded that the concepts were too broad to be of practical use. Or you may have heard criticisms implicit in the pretentious-sounding name, "Knowledge-based Engineering," as in: "you mean as opposed to Ignorance-based Engineering?"

To provide a clearer picture, we hope you will concur that Genworks' concept of a KB system is straightforward, relatively uncomplicated, and practical. In this manual our goal is to make you both comfortable and motivated to explore the ideas we have built into our flagship system, Genworks GDL.

Our informal definition of a *Knowledge Base System* is a hybrid *Object-Oriented*[2] and *Functional*[3] programming environment, which implements the features of *Caching* and *Dependency tracking*. Caching means that once the KB system has computed something, it generally will not

[1]From time to time, you will also see references to "Gendl." This refers to "The Gendl Project" which is the name of an open-source software project from which Genworks GDL draws for its core technology. "The Gendl Project" code is free to use for any purpose, but it is released under the Gnu Affero General Public License, which stipulates that applications code compiled with The Gendl Project compiler must be distributed as open-source under a compatible license (if distributed at all). Commercial Genworks GDL, properly licensed for development and/or runtime distribution, does not have this "copyleft" open-sourcing requirement.

[2]An *Object-Oriented* programming environment supports named collections of values along with procedures to operate on those values, including the possibility to modify ("mutate") the data. See https://en.wikipedia.org/wiki/Object-oriented_programming

[3]A pure *Functional* programming environment supports only the evaluation of Functions which work by computing results, but do not modify (i.e. "mutate") the in-memory state of any objects. See http://en.wikipedia.org/wiki/Functional_programming

need to repeat that computation if the same question is asked again. Dependency tracking is the flip side of that coin — it ensures that if a cached result is *stale*, the result will be recomputed the next time it is *demanded*, in order to give a fresh result.

1.3 Classic Definition of Knowledge Based Engineering (KBE)

Sections 1.3 through 1.8 are sourced from [1].

> Knowlege based engineering (KBE) is a technology predicated on the use of dedicated software tools called KBE systems, which are able to capture and systematically re-use product and process engineering knowledge, with the final goal of reducing the time and costs of product development by means of the following:
>
> - Automation of repetitive and non-creative design tasks;
> - Support of multidisciplinary design optimization in all phases of the design process

1.4 Runtime Value Caching and Dependency Tracking

> Caching refers to the ability of the KBE system to memorize at runtime the results of computed values (e.g. computed slots and instantiated objects), so that they can be reused when required, without the need to re-compute them again and again, unless necessary. The dependency tracking mechanism serves to keep track of the current validity of the cached values. As soon as these values are no longer valid (stale), they are set to unbound and recomputed if and only at the very moment they are again demanded.
>
> This dependency tracking mechanism is at the base of associative modeling, which is of extreme interest for engineering design applications. For example, the shape of a wing rib can be defined according to the shape of the wing aerodynamic surface. In case the latter is modified, the dependency tracking mechanism will notify the system that the given rib instance is no longer valid and will be eliminated from the product tree, together with all the information (objects and attributes) depending on it. The new rib object, including its attributes and the rest of the affected information, will not be re-instantiated/updated/re-evaluated automatically, but only when and if needed (see demand driven instantiation in the next section)

1.5 Demand-Driven Evaluation

> KBE systems use the *demand-driven* approach. That is, they evaluate only those chains of expressions required to satisfy a direct request of the user (i.e. the evaluation of certain attributes for the instantiation of an object), or the indirect requests of another object, which is trying to satisfy a user demand. For example, the system will create an instance of the rib object only when the weight of the abovementioned wing rib is required. The reference wing surface will be generated only when the generation of the

rib object is required, and so on, until all the information required to respond to the user request will be made available.

It should be recognized that a typical object tree can be structured in hundreds of branches and include thousands of attributes. Hence, the ability to evaluate *specific* attributes and product model branches at demand, without the need to evaluate the entire model from its root, prevents waste of computational resources and in many cases brings seemingly intractible problems to a rapid solution.

1.6 Object-oriented Systems

An object-oriented system is composed of objects (i.e. concrete instantiations of *named* classes), and the behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in the sense that when a target object receives a message, it decides on its own what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the current state of the target object.

1.7 Object-oriented Analysis

Object-oriented analysis (OOA) is the process of analyzing a task (also known as a problem domain) to develop a conceptual model that can then be used to complete that task. A typical OOA model would describe computer software that could be used to satisfy a set of customer-defined requirements. During the analysis phase of problem-solving, the analyst might consider a Written Requirements Statement, a formal vision document, or interviews with stakeholders or other interested parties. The task to be addressed might be divided into several subtasks (or domains), each representing a different business, technological, or other area of interest. Each subtask would be analyzed separately. Implementation constraints (e.g. concurrency, distribution, persistence, or how the system is to be built) are not considered during the analysis phase; rather, they are addressed during the object-oriented design (OOD) phase.

The conceptual model that results from OOA will typically consist of a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some form of user interface.

1.8 Object-oriented Design

During the object-oriented design (OOD) phase, a developer applies implementation constraints to the conceptual model produced in the object-oriented analysis. Such constraints could include not only those imposed by the chosen architecture but also any non-functional — technological or environmental — constraints, such as data processing capacity, response time, run-time platform, development environment, or those inherent in the programming language. Concepts in the analysis model are mapped onto implementation classes and interfaces resulting in a model of the solution domain, i.e., a detailed description of *how* the system is to be built.

1.9 The Object-Oriented Paradigm meets the Functional paradigm

In order to model very complex products and efficiently manage large bodies of knowledge, KBE systems tap the potential of the object oriented nature of their underlying language (e.g. Common Lisp). "Object" in this context refers to an instantiated data structure *of a particular assigned data type*. As is well-known in the computing community, unrestricted modification of the state of objects leads to unmaintainable systems which are difficult to debug. KBE systems manage this drawback by strictly controlling and constraining any ability to modify or "change state" of objects.

In essence, a KBE system generates a tree of inspectable objects which is analogous to the function call tree of pure functional-language systems.

1.10 Goals for this Manual

This manual is designed as a companion to a live two-hour GDL/GWL tutorial, but you may also be relying on it independently of the tutorial. Portions of the live tutorial are available in "screencast" video form, in the Documentation section of `http://genworks.com` In any case, our fundamental goals of this Manual are:

- To get you motivated about using Genworks GDL
- Enable you to ascertain whether Genworks GDL is an appropriate tool for a given job
- Equip you with the ability to state the case for using GDL/GWL when appropriate
- Prepare you to begin authoring and maintaining GDL applications, or porting apps from similar KB systems into GDL.

The manual will begin with an introduction to the Common Lisp programming language. If you are new to Common Lisp: welcome! You are about to be introduced to a powerful tool backed by a rock-solid body of standard specifications, which will protect your development investment for decades to come. In addition to the overview provided in this manual, many resources are available to get you started in CL — for starters, we recommend Basic Lisp Techniques[4], which was written by the author.

1.11 What is GDL?

GDL is an acronym for "General-purpose Declarative Language."

- In a nutshell, GDL is a system for creating custom, targeted CAD systems and other applications. By "targeted," we mean these end-user CAD systems and applications are designed

[4] BLT is available at `http://www.franz.com/resources/educational_resources/cooper.book.pdf`

to complete a specific, rule-based set of design tasks, as distinct from a general-purpose CAD system which works as an electronic drafting board or interactive 3D modeling system.

- GDL is a superset of ANSI Common Lisp, and consists largely of automatic code-expanding extensions to Common Lisp implemented in the form of macros. When you write, for example, 20 lines in GDL, you might be writing the equivalent of 200 lines of Common Lisp. Given that GDL is a superset of Common Lisp, you of course retain the full power of the CL language at your disposal whenever you are working in GDL.

- Since GDL expands into CL, everything you write in GDL 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 from some other so-called KB systems on the market, which are essentially nothing more than interpreted *scripting languages* which often impose arbitrary limits on the size and complexity of the application].

- GDL is also a *declarative* language in the fullest sense. When you put together a GDL application, you think and write 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 propery, in what order this will happen, and so forth. Those details are managed automatically by the embedded language.

- Because GDL 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" complexities

- GDL 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, you are free to use the Generic Function paradigm. Do not be concerned at this point if you are not fully conversant with the differences between Message Passing and Generic Function models of object-orientation.[5].

1.12 Why GDL (i.e., what is GDL good for?)

- Organizing and integrating large amounts of information in ways which are impossible or impractical using conventional languages, CAD systems, and/or database technology alone;

- Evaluating many design or engineering alternatives and performing various kinds of optimizations within specified design spaces, and doing so *very rapidly;*

[5] See Paul Graham's ANSI Common Lisp, page 192, for an excellent discussion of the Two Models of Object-oriented Programming. Peter Siebel's Practical Common Lisp also covers the topic; see http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html.

- Capturing, i.e., implementing, the procedures and rules used to solve repetitive tasks in engineering and other fields;
- Applying rules you have specified to achieve intermediate and final outputs, which may include virtual models of wireframe, surface, and solid geometric objects.

1.13 What GDL is not

- A CAD system (although it may operate on and/or generate geometric entities, and it can generate *applications* which can be considered as special-purpose CAD systems);
- A drawing program (although it may operate on and/or generate geometric entities);
- An Artificial Intelligence system (although it is an excellent environment for developing capabilities which could qualify as such);
- An Expert System Shell (although one could be easily embedded within it).

Without further description, let's turn the page and get started with hands-on GDL...

Chapter 2

Installation [GDL and Gendl]

Please follow Section 2.1 if your email address is registered with Genworks and you will install a pre-packaged Genworks GDL distribution, including its own Common Lisp engine. The foundation of Genworks GDL is also available as open-source software through The Gendl Project[1]; if you elect to use that version, then please refer to Section 2.2.

2.1 Installation of pre-packaged GDL

This section will take you through the installation of Genworks GDL from a prepackaged distribution with the Allegro CL or LispWorks commercial Common Lisp engine and the Slime IDE (based on Gnu Emacs).

2.1.1 Download the Software and retrieve a license key

1. Visit the Downloads section of the Genworks Website;
2. Enter your email address[2];
3. Download the latest Payload for Windows, Linux, or Mac;
4. Click to receive the license key file by email.

2.1.2 Unpack the Distribution

Genworks GDL is currently distributed as a setup executable for Windows, a "dmg" application bundle for Mac, and a self-contained zip file for Linux.

- Run the installation executable. Accept the defaults when prompted.[3]
- Copy the license key file as gdl.lic (for Trial, Student, Professional editions), or devel.lic (for Enterprise edition) into the `program/` directory within the `gdl/gdl/program/` directory.

[1] http://github.com/genworks/gendl

[2] if your address is not on file, send mail to licensing@genworks.com

[3] For Linux, you have to install emacs and ghostscript yourself. Please use your distribution's package manager to complete this installation.

- Launch the application by finding the Genworks program group in the Start menu (Windows), or by double-clicking the application icon (Mac), or by running the `run-gdl` script (Linux).

2.2 Installation of open-source Gendl

This section is only germane if you have not received a pre-packaged Gendl or Genworks GDL distribution with its own Common Lisp engine. If you have received a pre-packaged Gendl distribution then you may skip this section. In case you want to use the open-source Gendl, you will use your own Common Lisp installation and obtain Gendl (Genworks-GDL) using a powerful and convenient CL package/library manager called *Quicklisp*.

2.2.1 Install and Configure your Common Lisp environment

Gendl is currently tested to build on the following Common Lisp engines:

- Allegro CL (commercial product from Franz Inc, free Express Edition available)
- LispWorks (commercial product from LispWorks Ltd, free Personal Edition available)
- Clozure CL (free CL engine from Clozure Associates, free for all use)
- Steel Bank Common Lisp (SBCL) (free open-source project with permissive license)

Please refer to the documentation for each of these systems for full information on installing and configuring the environment. Typically this will include a text editor, either Gnu Emacs with Superior Lisp Interaction Mode for Emacs (Slime), or a built-in text editing and development environment which comes with the Common Lisp system.

A convenient way to set up Emacs with Slime is to use the Quicklisp-slime-helper.

2.2.2 Load and Configure Quicklisp

Quicklisp is the defacto standard library manager for Common Lisp.

- Visit the Quicklisp website
- Follow the instructions there to download the `quicklisp.lisp` bootstrap file and load it to set up your Quicklisp environment.

2.2.3 Load and Start Gendl

invoke the following commands at the Common Lisp toplevel "repl" prompt:

1. `(ql:quickload :gendl)`
2. `(gendl:start-gendl!)`

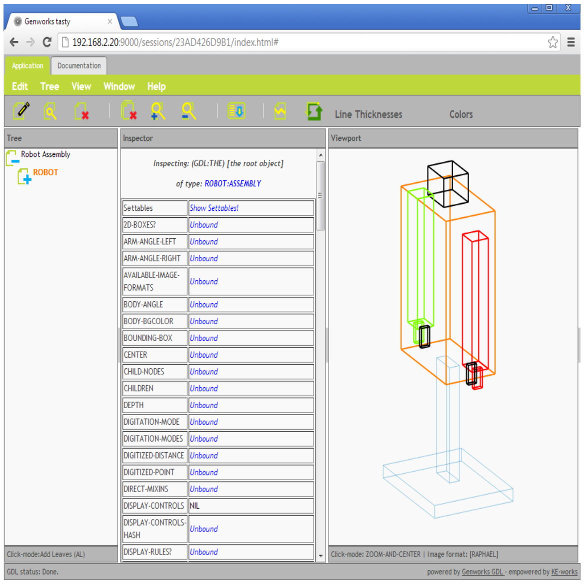

Figure 2.1: Robot displayed in Tasty

2.3 System Testing

2.3.1 Basic Sanity Test

You may test your installation using the following checklist. These tests are optional. You may perform some or all of them in order to ensure that your Gendl is installed correctly and running smoothly. In your Web Browser (e.g. Google Chrome, Firefox, Safari, Opera, Internet Explorer), perform the following steps:

1. visit http://localhost:9000/tasty.
2. accept default robot:assembly.
3. Select "Add Leaves" from the Tree menu.
4. Click on the top node in the tree.
5. Observe the wireframe graphics for the robot as shown in 2.1.
6. Click on the robot to zoom in.
7. Select "Clear View!" from the View menu.
8. Select "X3DOM" from the View menu.
9. Click on the top node in the tree.
10. "Refresh" or "Reload" your browser window (may not be necessary).

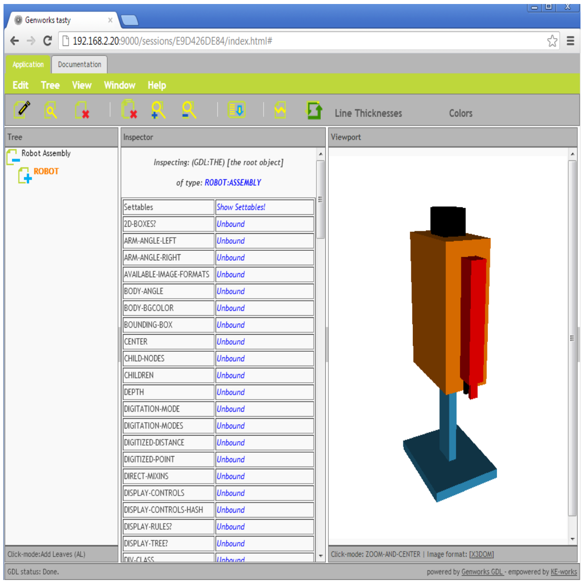

Figure 2.2: Robot x3dom

11. If your browser supports WebGL, you will see the robot in shaded dynamic view as shown in Figure 2.2.

12. Select "PNG" from the View menu. You will see the wireframe view of the robot as a PNG image.

13. Select "X3D" from the View menu. If your browser has an X3D plugin installed (e.g. BS Contact), you will see the robot in a shaded dynamic view.

2.3.2 Full Regression Test

The following commands will invoke a full regression test, including a test of the Surface and Solids primitives provided by the SMLib geometry kernel. Note that the SMLib geometry kernel is only available with proprietary Genworks GDL licenses — therefore, if you have open-source Gendl or a lite Trial version of Genworks GDL, these regression tests will not all function.

In Emacs at the `gdl-user` prompt in the `*slime-repl...*` buffer, type the following commands:

1. `(ql:quickload :regression)`
2. `(gdl-lift-utils::define-regression-tests)`
3. `(gdl-lift-utils::run-regression-tests-pass-fail)`
4. `(pprint gdl-lift-utils::*regression-test-report*)`

2.4 Getting Help and Support

If you encounter unexplained errors in the installation and startup process, please contact the following resources:

1. Make a posting to the Genworks Google Group
2. Join the #gendl IRC (Internet Relay Chat) channel on irc.freenode.net and discuss issues there.
3. For exclusively Common Lisp issues, join the #lisp IRC (Internet Relay Chat) channel on irc.freenode.net and discuss issues there.
4. Also for Common Lisp issues, follow the comp.lang.lisp Usenet group.
5. If you are a supported Genworks customer, send email to support@genworks.com
6. If you are not a supported Genworks customer but you want to report an apparent bug or have other suggestions or inquiries, you may also send email to support@genworks.com, but as a non-customer please understand that Genworks cannot guarantee a response or a particular time frame for a response. Also note that we are not able to offer guaranteed support for Trial and Student licenses

Chapter 3

Basic Operation of the GDL Environment

This chapter will lead you through all the basic steps of operating a typical GDL-based development environment. We will not in this section go into particular depth about the additional features of the environment or language syntax — this chapter is merely to familiarize you with, and start you practicing with the nuts and bolts of operating the environment with a keyboard.

3.1 What is Different about GDL?

GDL is a *dynamic* language environment with incremental compiling and in-memory definitions. This means that as long as the system is running you can *compile* new *definitions* of functions, objects, etc, and they will immediately become available as part of the running system, and you can begin testing them immediately, or update an existing set of objects to observe their new behavior.

In many other programming language systems, to introduce a new function or object, one has *to start the system from the beginning* and reload all the files in order to test new functionality.

In GDL, it is typical to keep the same development session up and running for an entire day or longer, making it unnecessary to repeatedly recompile and reload your definitions from scratch. Note, however, that if you do shut down and restart the system for some reason, then you will have to recompile and/or reload your application's definitions in order to bring the system back into a state where it can instantiate (or "run") your application.

While this can be done manually at the command-line, it is typically done *automatically* in one of two ways:

1. Using commands placed into the `gdlinit.cl` initialization file, as described in Section 3.4.

2. Alternatively, you can compile and load definitions into your session, then save the "world" in that state. That way it is possible to start a new GDL "world" which already has all your application's definitions loaded and ready for use, without having to procedurally reload any files. You can then begin to make and test new definitions (and re-definitions) starting from this new "world." You can think of a saved "world" like pre-made cookie dough: no need to add each ingredient one by one — just start making cookies!

3.2 Startup, "Hello, World!" and Shutdown

The typical GDL environment consists of three programs:

1. Gnu Emacs (the editor);
2. a Common Lisp engine with GDL system loaded or built into it (e.g. the `gdl.exe` executable in your `program/` directory); and
3. (optionally) a web browser such as Firefox, Google Chrome, Safari, Opera, or Internet Explorer

Emacs runs as the main *process*, and this in turn starts the CL engine with GDL as a *sub-process*. The CL engine typically runs an embedded *webserver*, enabling you to access your application through a standard web browser.

As described in Chapter 2, the typical way to start a pre-packaged GDL environment is with the `run-gdl.bat` (Windows), or `run-gdl` (MacOS, Linux) script files, or with the installed Start program item (Windows) or application bundle (MacOS). Invoke this script file from the Start menu (Windows), your computer's file manager, or from a desktop shortcut if you have created one. Your installation executable may also have created a Windows "Start" menu item for Genworks GDL. You can of course also invoke `run-gdl.bat` from the Windows "cmd" command-line, or from another command shell such as Cygwin.[1]

3.2.1 Startup

Startup of a typical GDL development session consists of two fundamental steps: (1) starting the Emacs editing environment, and (2) starting the actual GDL process as a "sub-process" or "inferior" process within Emacs. The GDL process should automatically establish a network connection back to Emacs, allowing you to interact directly with the GDL process from within Emacs.

1. Invoke the `run-gdl.bat`, `run-gdl.bat` startup script, or the provided executable from the Start menu (windows) or application bundle (Mac).
2. You should see an emacs window similar to that shown in Figure 3.1. (alternative colors are also possible).
3. (MS Windows): Look for the Genworks GDL Console window, or (Linux, Mac) use the Emacs "Buffer" menu to visit the "*inferior-lisp*" buffer. Note that the Genworks GDL Console window might start as a minimized icon; click or double-click it to un-minimize.
4. Watch the Genworks GDL Console window for any errors. Depending on your specific installation, it may take from a few seconds to several minutes for the Genworks GDL Console (or *inferior-lisp* buffer) to settle down and give you a `gdl-user():` prompt. This window is where you will see most of your program's textual output, any error messages, warnings, etc.

[1] Cygwin is also useful as a command-line tool on Windows for interacting with a version control system like Subversion (svn).

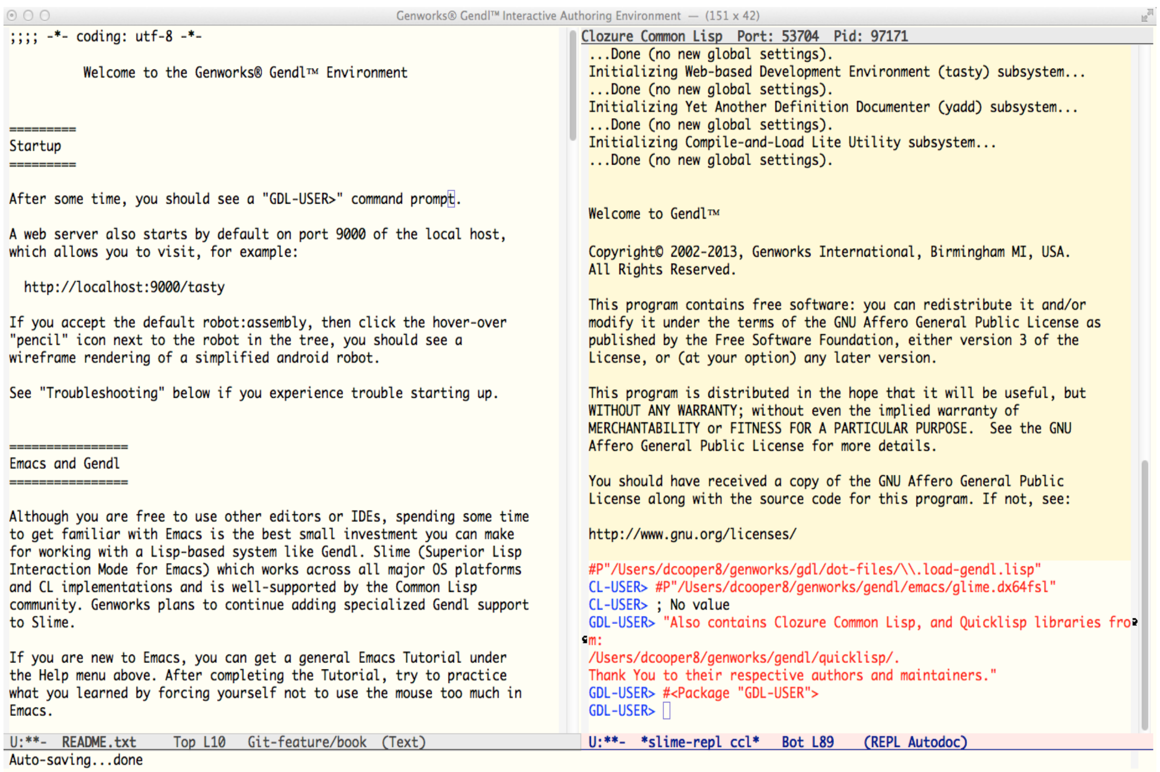

Figure 3.1: Startup of Emacs with GDL

5. In Emacs, type: `C-x &` (or select Emacs menu item *Buffers→*slime-repl...**) to visit the "*slime-repl ...*" buffer. The full name of this buffer depends on the specific CL/GDL platform which you are running. This buffer contains an interactive prompt, labeled `gdl-user>`, where you will enter most of your commands to interact with your running GDL session for testing, debugging, etc. There is also a web-based graphical interactive environment called *tasty* which will be discussed in Chapter 6.

6. To ensure that the GDL command prompt is up and running, type: `(+ 2 3)` and press [Enter].

7. You should see the result `5` echoed back to you below the prompt.

3.2.2 Developing and Testing a "Hello World" application

1. type C-x (Control-x) 2, or C-x 3, or use the "Split Screen" option of the File menu to split the Emacs frame into two "windows" ("windows" in Emacs are non-overlapping panels, or rectangular areas within the main Emacs window).

2. type C-x o several times to move from one window to the other, or move the mouse cursor and click in each window. Notice how the blinking insertion point moves from one window to the other.

3. In the top (or left) window, type C-x C-f (or select Emacs menu item "File→Open File") to get the "Find file" prompt in the mini-buffer.

4. Type C-a to move the point to the beginning of the mini-buffer line.

5. Type C-k to delete from the point to the end of the mini-buffer.

6. Type `~/hello.gdl` and press [Enter]

7. You are now editing a (presumably new) file of GDL code, located in your HOME directory, called `hello.gdl`

8. Enter the text from Figure 3.2 into the `hello.gdl` buffer. You do not have to match the line breaks and whitespace as shown in the example. You can auto-indent each new line by pressing [TAB] after pressing [Enter] for the newline.

   *Protip:*You can also try using `C-j` instead of [Enter], which will automatically give a newline and auto-indent.

9. type `C-x C-s` (or choose Emacs menu item *File→Save*) to save the contents of the buffer (i.e. the window) to the file in your HOME directory.

10. type `C-c C-k` (or choose Emacs menu item *SLIME→Compilation→Compile/Load File*) to compile & load the code from this file.

11. type `C-c o` (or move and click the mouse) to switch to the bottom window.

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

(define-object hello ()

  :computed-slots
  ((greeting "Hello, World!")))
```

Figure 3.2: Example of Simple Object Definition

12. In the bottom window, type `C-x &` (or choose Emacs menu item *Buffers→*slime-repl...**) to get the `*slime-repl ...*` buffer, which should contain a `gdl-user>` prompt. This is where you normally type interactive GDL commands.

13. If necessary, type `M` > (that is, hold down Meta (Alt), Shift, and the ">" key) to move the insertion point to the end of this buffer.

14. At the `gdl-user>` prompt, type

    ```
    (make-self 'hello)
    ```

    and press [Enter].

15. At the `gdl-user>` prompt, type

    ```
    (the greeting)
    ```

    and press [Enter].

16. You should see the words `Hello, World!` echoed back to you below the prompt.

3.2.3 Shutdown

To shut down a development session gracefully, you should first shut down the GDL process, then shut down your Emacs.

- Type `M-x quit-gdl` (that is, hold Alt and press X, then release both while you type `quit-gdl` in the mini-buffer), then press [Enter]

- alternatively, you can type `C-x &` (that is, hold Control and press X, then release both while you type &. This will visit the *slime-repl* buffer. Now type: `, q` to quit the GDL session.

- Finally, type `C-x C-c` to quit from Emacs. Emacs will prompt you to save any modified buffers before exiting.

```
apps/yoyodyne/booster-rocket/source/assembly.gdl
apps/yoyodyne/booster-rocket/source/package.gdl
apps/yoyodyne/booster-rocket/source/parameters.gdl
apps/yoyodyne/booster-rocket/source/rules.gdl
```

Figure 3.3: Example project directory with four source files

```
apps/yoyodyne/booster-rocket/source/assembly.gdl
apps/yoyodyne/booster-rocket/source/file-ordering.isc
apps/yoyodyne/booster-rocket/source/package.gdl
apps/yoyodyne/booster-rocket/source/parameters.gdl
apps/yoyodyne/booster-rocket/source/rules.gdl
```

Figure 3.4: Example project directory with file ordering configuration file

3.3 Working with Projects

GDL contains utilities which allow you to treat your application as a "project," with the ability to compile, incrementally compile, and load a "project" from a directory tree of source files representing your project. In this section we provide an overview of the expected directory structure and available control files, followed by a reference for each of the functions included in the bootstrap module.

3.3.1 Directory Structure

You should structure your applications in a modular fashion, with the directories containing actual Lisp sources called "source." You may have subdirectories which themselves contain "source" directories. We recommend keeping your codebase directories relatively flat, however.

In Figure 3.3 is an example application directory, with four source files.

3.3.2 Source Files within a source/ subdirectory

Enforcing Ordering

Within a source subdirectory, you may have a file called `file-ordering.isc`[2] to enforce a certain ordering of the files. Here are the contents of an example for the above application:

```
("package" "parameters")
```

This will force package.lisp to be compiled/loaded first, and parameters.lisp to be compiled/loaded next. The ordering on the rest of the files should not matter (although it will default to lexigraphical ordering).

Now our sample application directory appears as in Figure 3.4.

[2]`isc` stands for "Intelligent Source Configuration"

3.3.3 Generating an ASDF System

ASDF stands for Another System Definition Facility, which is the predominant system in use for Common Lisp third-party libraries. With GDL, you can use the `:create-asd-file?` keyword argument to make cl-lite generate an ASDF system file instead of actually compiling and loading the system. For example:

```
(cl-lite "apps/yoyodyne/" :create-asd-file? t)
```

In order to include a depends-on clause in your ASDF system file, create a file called `depends-on.isc` in the toplevel directory of your system. In this file, place a list of the systems your system depends on. This can be systems from your own local projects, or from third-party libraries. For example, if your system depends on the `:cl-json` third-party library, you would have the following contents in your `depends-on.isc`:

```
(:cl-json)
```

3.3.4 Compiling and Loading a System

Once you have generated an ASDF file, you can compile and load the system using Quicklisp. To do this for our example, follow these steps:

1. `(cl-lite "apps/yoyodyne/" :create-asd-file? t)`

   to generate the asdf file for the yoyodyne system. This only has to be done once after every time you add, remove, or rename a file or folder from the system.

2. `(pushnew "apps/yoyodyne/" ql:*local-project-directories* :test #'equalp)`

   This can be done in your `gdlinit.cl` for projects you want available during every development session. Note that you should include the full path prefix for the directory containing the ASDF system file.

3. `(ql:quickload :gdl-yoyodyne)`

   This will compile and load the actual system. Quicklisp uses ASDF at the low level to compile and load the systems, and Quicklisp will retrieve any depended-upon third-party libraries from the Internet on-demand. Source files will be compiled only if the corresponding binary (fasl) file does not exist or is older than the source file. By default, ASDF keeps its binary files in a *cache* directory, separated according to the CL platform and operating system. The location of this cache is system-dependent, but you can see where it is by observing the compile and load process.

3.4 Customizing your Environment

You may customize your environment in several different ways, for example by loading definitions and settings into your GDL "world" automatically when the system starts, and by specifying fonts, colors, and default buffers (to name a few) for your emacs editing environment.

3.5 Saving the World

"Saving the world" refers to a technique of saving a complete binary image of your GDL "world" which contains all the currently compiled and loaded definitions and settings. This allows you to start up a saved world almost instantly, without being required to reload all the definitions. You can then incrementally compile and load just the particular definitions which you are working on for your development session.

To save a world, follow these steps:

1. Load the base GDL code and (optionally) code for GDL modules (e.g. gdl-yadd, gdl-tasty) you want to be in your saved image. Note that in some implementations, this has step to be done in a plain session without multiprocessing (i.e. without an Emacs connection) - so you would do this loading step from a command shell e.g. Windows cmd prompt. For example:

   ```
   (ql:quickload :gdl-yadd)
   (ql:quickload :gdl-tasty)
   ```

2. (needed only for full GDL):

   ```
   (ff:unload-foreign-library (merge-pathnames "smlib.dll" "sys:smlib;"))
   ```

3. `(net.aserve:shutdown)`

4. (to save an image named yoyodyne.dxl) Invoke the command

   ```
   (ensure-directories-exist "~/gdl-images/")
   ```

   ```
   (uiop:dump-image dumplisp "~/gdl-images/yoyodyne")
   ```

   Note that the standard extension for Allegro CL images is `.dxl`. Prepend the file name with path information, to write the image to a specific location.

3.6 Starting up a Saved World

In order to start up GDL using a custom saved image, or "world," follow these steps

1. Exit GDL
2. Copy the supplied image file, e.g.`gdl.dxl` to `gdl-orig.dxl`.
3. Move the custom saved dxl image to `gdl.dxl` in the GDL application `"program/"` directory.
4. Start GDL as usual. Note: you may have to edit the system gdlinit.cl or your home gdlinit.cl to stop it from loading redundant code which is already in the saved image.

Chapter 4

Understanding Common Lisp

GDL is a superset of Common Lisp (CL) — that is, all of CL is available to you during development, and is available to your applications at runtime (i.e. after they are deployed). The lowest-level expressions in a GDL definition are CL "symbolic expressions," or "S-expressions." This chapter will familiarize you with CL S-expressions.

4.1 S-expression Fundamentals

S-expressions can be used in a similar manner to Formulas in a Spreadsheet to establish the value of a particular *slot* (i.e. named data value) in an object. However, unlike in a spreadsheet, these values are only computed on an as-needed basis (i.e. "on-demand"). You can also evaluate S-expressions at the toplevel `gdl-user>` prompt, and see the result immediately. In fact, this toplevel prompt is called a *read-eval-print* loop, because its purpose is to *read* each S-expression entered, *evaluate* the expression to yield a result (or *return-value*), and finally to *print* that result.

CL S-expressions use a *prefix* notation, which means that they consist of either an *atom* (e.g. number, text string, symbol) or a *list* (one or more items enclosed by parentheses, where the first item is taken as a symbol which names an operator). Here is an example:

```
(+ 2 2)
```

This expression consists of the function named by the symbol `+`, followed by the numeric arguments `2` and another `2`. As you may have suspected, 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 occurring here? When CL is asked to evaluate an expression, it processes the expression according to the following rules:

- If the expression is an *atom* (e.g. a non-list datatype such as a number, text string, or literal symbol), it simply returns itself as its evaluated value. Examples:
  - ```
    gdl-user> 99
    99
    ```
  - ```
    gdl-user> 99.9
    99.9
    ```

- ```
  gdl-user> 3/5
  3/5
  ```
- ```
  gdl-user> "Bob"
  "Bob"
  ```
- ```
  gdl-user> "Our golden rule is simplicity"
  "Our golden rule is simplicity"
  ```
- ```
  gdl-user> 'my-symbol
  my-symbol
  ```

Note that numbers are represented directly (with decimal points and slashes for fractions allowed), strings are surrounded by double-quotes, and literal symbols are introduced with a preceding single-quote. Symbols are allowed to have dashes ("-") and most other special characters. By convention, the dash is used as a word separator in CL symbols.

- If the expression is a *list* (i.e. is surrounded by parentheses), CL processes the *first* element in this list as an *operator name*, and the *rest* of the elements in the list represent the *arguments* to the operator. An operator can take zero or more arguments, and can return zero or more return-values. Some operators evaluate their arguments immediately and work directly on those values (these are called *functions*). Other operators expand into other code. These are called *special operators* or *macros*. Macros are what give Lisp (and CL in particular) its special power. Here are some examples of functional S-expressions:

  - ```
    gdl-user> (expt 2 5)
    32
    ```
  - ```
    gdl-user> (+ 2 5)
    7
    ```
  - ```
    gdl-user> (+ 2)
    2
    ```
  - ```
    gdl-user> (+ (+ 2 2) (+ 3 3 ))
    10
    ```

4.2 Fundamental CL Data Types

As has been noted, Common Lisp natively supports many data types[1] common to other languages, such as numbers and text strings. CL also contains several *compound* data types such as lists, arrays, and hash tables. CL contains *symbols* as well, which typically are used as names for other data elements.

Regarding data types, CL follows a system 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. For example, a variable (or slot name in GDL) called `length` could contain a value `42` (an integer), `42.43` (a floating-point number), `3/16` (a rational number), or even `:long` (a descriptive keyword symbol).

[1]See http://en.wikipedia.org/wiki/Data_type for a more detailed discussion of what is meant by "data types" in this context.

4.2.1 Numbers

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

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, for example, `1/3` is a true one-third, not `0.333333333` rounded off at some arbitrary precision short of infinity.

4.2.2 Strings

Strings are actually a specialized kind of array, namely a one-dimensional array (vector) made up of text characters. These characters can be letters, numbers, or punctuation, and in some cases can include characters from international character sets (e.g. Unicode or UTF-8) such as Chinese Hanzi or Japanese Kanji. The string delimiter in CL is the double-quote character.

Text strings in CL are a native data type which simply evaluate to themselves when included in an expression.

A common way to produce a string in CL is with the `format` function. Although the `format` function can be used to send output to any kind of destination, or *stream*, it will simply yield a string if you specify `nil` for the stream. Example:

```
gdl-user> (format nil "The time is: ~a" (get-universal-time))
"The time is: 3564156603"
gdl-user> (format nil "The time is: ~a" (iso-8601-date (get-universal-time)))
"The time is: 2012-12-10"
gdl-user> (format nil "The time is: ~a" (iso-8601-date (get-universal-time) :include-time? t))
"The time is: 2012-12-10T14:30:17"
```

As the above example demonstrates, `format` takes a *stream designator* or `nil` as its first argument, then a *format-string*, then enough arguments to match the *format directives* in the format-string. Format directives begin with the tilde character ( ). The format-directive `a` indicates that the printed representation of the corresonding argument should simpy be substituted into the format-string at the point where it occurs.

We will cover more details on `format` in a section on Input/Output, but for now, a familiarity with the simple use of `(format nil ...)` will be helpful for Chapter 5.

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 capacity 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 contains the actual function object for performing numeric addition. 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 fundamental 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 are located 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 an identified *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, for example:

```
'(a b c)
```

or

```
'(a (b c) d)
```

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

4.2.4 List Basics

Lisp derives its name from its strong support for the list data structure. The list concept is important to Common Lisp (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 items surrounded by parentheses. The items can be objects of any valid CL data types, such as numbers, strings, symbols, lists, or other kinds of objects. According to standard evaluation rules, you must quote a literal list to evaluate it as such, 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 concern yourself about analyzing the function definition 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?[2]

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 processed by 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.

4.2.5 The List as a Data Structure

In this section we will discuss a few of the fundamental native CL operators for manipulating lists as data structures. These include operators for doing things such as:

1. finding the length of a list;
2. accessing particular members of a list;
3. appending multiple lists together to make a new list.

Finding the Length of a List

The function `length` will return the length of any type of sequence, including a list:

```
gdl-user> (length '(a b c d e f g h i j)
10
gdl-user> (length nil)
0
```

Note that `nil` qualifies as a list (albeit the empty list), so taking its length yields the integer `0`.

[2]Answer: symbol, symbol, (empty) list, list with symbol and string.

Accessing the Elements of a List

Common Lisp defines the accessor functions `first` through `tenth` as a means of accessing the first ten elements in a list:

```
gdl-user> (first '(a b c))

a

gdl-user> (second '(a b c))

b

gdl-user> (third '(a b c))

c
```

For accessing elements in an arbitrary position in the list, you can use the function nth, which takes an integer and a list as its two arguments:

```
gdl-user> (nth 0 '(a b c))

a

gdl-user> (nth 1 '(a b c))

b

gdl-user> (nth 2 '(a b c))

c
```

Note that nth starts its indexing at zero (0), so `(nth 0 ...)` is equivalent to `(first ...)`and `(nth 1 ...)` is equivalent to `(second ...)`, etc.

Using a List to Store and Retrieve Named Values

Lists can also be used to store and retrieve named values. When a list is used in this way, it is called a *plist*. Plists contain pairs of elements, where each pair consists of a *key* and some *value*. The key is typically an actual keyword symbol — that is, a symbol preceded by a colon (:). The value can be any value, such as a number, a string, or even a GDL object representing something complex such as an aircraft.

A plist can be constructed in the same manner as any list, e.g. with the `list` operator:

```
(list :a 10 :b 20 :c 30)
```

In order to access any element in this list, you can use the getf operator. The getf operator is specially intended for use with plists:

```
gdl-user> (getf (list :a 10 :b 20 :c 30) :b
20
gdl-user> (getf (list :a 10 :b 20 :c 30) :c
30
```

Common Lisp contains several other data structures for mapping keywords or numbers to values, such as *arrays* and *hash tables*. But for relatively short lists, and especially for rapid prototyping and testing work, plists can be useful. Plists can also be written and read (i.e. saved and restored) to and from plain text files in your filesystem, in a very natural way.

Appending Lists

The function append takes any number of lists, and returns a new list which results from appending them together. Like many CL functions, append does not *side-effect*. That is, it simply returns a new list as a return-value, but does not modify its arguments in any way:

```
gdl-user> (defparameter my-slides '(introduction welcome lists functions))
(introduction welcome lists functions)

gdl-user> (append my-slides '(numbers))
(introduction welcome lists functions numbers)

gdl-user> my-slides
(introduction welcome lists functions)
```

Note that the simple call to append does not affect the variable my-slides. Later we will observe how one may alter the value of a variable such as my-slides.

4.3 Summary

In this chapter we have presented enough basics of Lisp's minimal syntax, and some particulars of Common Lisp, to enable you to start with the Genworks GDL framework. In keeping with the demand-driven philosophy of GDL, subsequent chapters will cover additional CL material on an as-needed basis.

Chapter 5

Understanding GDL — Core GDL Syntax

Now that you have a basic familiarity with Common Lisp syntax (or, more accurately, the *absence* of syntax), we will move directly into the Genworks GDL framework. By using GDL you can formulate most of your engineering and computing problems in a natural way, without becoming involved in the complexity of the Common Lisp Object System (CLOS).

As discussed in the previous chapter, GDL is based on and is a superset of ANSI Common Lisp. Because ANSI CL is unencumbered and is an open standard, with several commercial and free implementations, it is a good wager that applications written in it will continue to be usable for the balance of this century, and beyond. Many commercial products have a shelf life only until a new product comes along. Being based in ANSI Common Lisp ensures GDL's permanence.

[The GDL product is a commercially available KBE system with Proprietary licensing. The Gendl Project is an open-source Common Lisp library which contains the core language kernel of GDL, and is licensed under the terms of the Affero Gnu Public License. The core GDL language is a proposed standard for a vendor-neutral KBE language.]

5.1 Defining a Working Package

In Common Lisp, *packages* are a mechanism to separate symbols into namespaces. Using packages it is possible to avoid "naming" conflicts in large projects. Consider this analogy: in the United States, telephone numbers are preceded by a three-digit area code and then consist of a seven-digit number. The same seven-digit number can occur in two or more separate area codes, without causing a conflict.

The macro `gdl:define-package` is used to set up a new working package in GDL.

Example:

```
(gdl:define-package :yoyodyne)
```

will establish a new package (i.e. "area code") called `:yoyodyne` which has all the GDL operators available.

The `:gdl-user` package 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 it is recommended that you make and work in your own GDL package, defined as above with `gdl:define-package`.

A Note for advanced users: Packages defined with `gdl:define-package` will implicitly *use* the `:gdl` 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 GDL operators, Common Lisp operators, and symbols from the `:cl-json` package [1], you could set it up as follows:

```
(ql:quickload :cl-json)
(gdl:define-package :yoyodyne (:use :cl-json))
```

The first form ensures that the cl-json code module is actually fetched and loaded. The second form defines a package with the `:cl-json` operators available to it.

5.2 Define-Object

Define-object is the basic macro for defining objects in GDL. An object definition maps directly into a Lisp (CLOS) class definition.

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. doesn't 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 making up the specification-plist:

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

computed-slots are actually cached methods, with expressions to compute and return a value.

objects specify other instances to be "contained" within this instance.

functions are (uncached) functions "of" the object, i.e. they operate just as normal CL functions, and accept arguments just like normal CL functions, with the added feature that you can also use *the* referencing, to refer to messages or reference chains which are available to the current object.

```
(define-object hello ()
  :input-slots (first-name last-name)

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

Figure 5.1: Example of Simple Object Definition

Figure 5.1 shows a simple example, which contains two input-slots, `first-name` and `last-name`, and a single computed-slot, `greeting`. A GDL Object is analogous in some ways to a CL `defun`, where the input-slots are like arguments to the function, and the computed-slots are like return-values. But seen another way, each slot in a GDL object serves as function in its own right.

The referencing macro `the` shadows CL's `the` (which is a seldom-used type declaration operator). `The` in GDL is a macro which 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. preceded by a colon), and the `the` macro will process them appropriately.

5.3 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. *Instance*, in this context, means a single occurence of the object with tangible values assigned to its input-slots. By way of analogy: an *object definition* is like a blueprint for a house; an *instance* is like an actual house. The `make-object` function is very similar to the CLOS `make-instance` function. Here 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(16): (setq my-instance
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f39c2>
```

Note that keyword symbols are used to "tag" the input values. And the return-value of *make-object* is an instance of class `hello`. Now that we have an instance, we can use the operator `the-object` to send messages to this instance:

[1] CL-JSON is a free third-party library for handling JSON format, a common data format used for Internet applications.

```
GDL-USER(17): (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. `The`, by contrast, assumes that the object instance is the lexical variable `self`, which is automatically set within the lexical context of a `define-object`.

Like `the`, `the-object` evaluates all but the first of its arguments as package-immune symbols, so although keyword symbols may be used, this is not a requirement, and plain, unquoted symbols will work just fine.

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

```
GDL-USER(18): (setq self
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f406a>

GDL-USER(19): (the greeting)
"Hello, John Doe!!"
```

In actual fact, `(the ...)` simply expands into `(the-object self ...)`.

5.4 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.

Inputs to each instance are specified as a plist of keywords and value expressions, spliced in after the object's name and type specification. These inputs must match the inputs protocol (i.e. the input-slots) of the object being instantiated. Figure 5.2 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*, which means that the first time these instances, or any of their messages, are referenced, the actual instance will be created *and* cached for future reference.

```
(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 5.2: Object Containing Child Objects

```
(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 5.3: Sample Data and Object Definition to Contain U.S. Presidents

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

Objects may be *sequenced*, to specify, in effect, an array or list of object instances. The most common type of sequence is called a *fixed size* sequence. See Figure 5.3 for an example of an object which contains a sequenced set of instances representing successive 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. A passed-in value will override the default expression.

5.6 Summary

This chapter has provided an introduction to the core GDL syntax. As with any language, practice (that is, usage) makes perfect. The chapters that follow will cover more specialized aspects of the GDL language, introducing additional Common Lisp concepts as they are required along the way.

Chapter 6

The Tasty Development Environment

Tasty[1] is a web based testing and tracking utility. Note that Tasty is designed for developers of GDL applications — that is, it is not intended as an end-user application interface (see Chapter 8 for the recommended steps to create end-user interfaces).

Tasty allows one to visualize and inspect any object defined in GDL which mixes at least `base-object` into the definition of its root[2]

First, make sure you have compiled and loaded the code for the Chapter 5 examples, contained in

```
.../src/documentation/tutorial/examples/chapter-5/
```

in your GDL distribution. If you are not sure how to do this, you may want to leave this section temporarily and review Chapter 3, and then return.

Now you should have the Chapter 5 example definitions compiled and loaded into the system. To access Tasty, point your web browser to the URL in figure 6.1. This will produce the start-up page, as seen in Figure 6.2[3]. To access an instance of a specific object definition, you specify the class package and the object type, separated by a colon (":") (or a double-colon ("::") in the event the symbol naming the type is not exported from the package). For example, consider the simple

[1] "Tasty" is an acronym of acronyms - it stands for TAtu with STYle (sheets), where tatu comes from Testing And Tracking Utility.

[2] 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 eliminated in a future GDL release so the user 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.

[3] This page may look slightly different, e.g. different icon images, depending on your specific GDL version.

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

;; for example:

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

Figure 6.1: Web Browser address for Tasty development environment

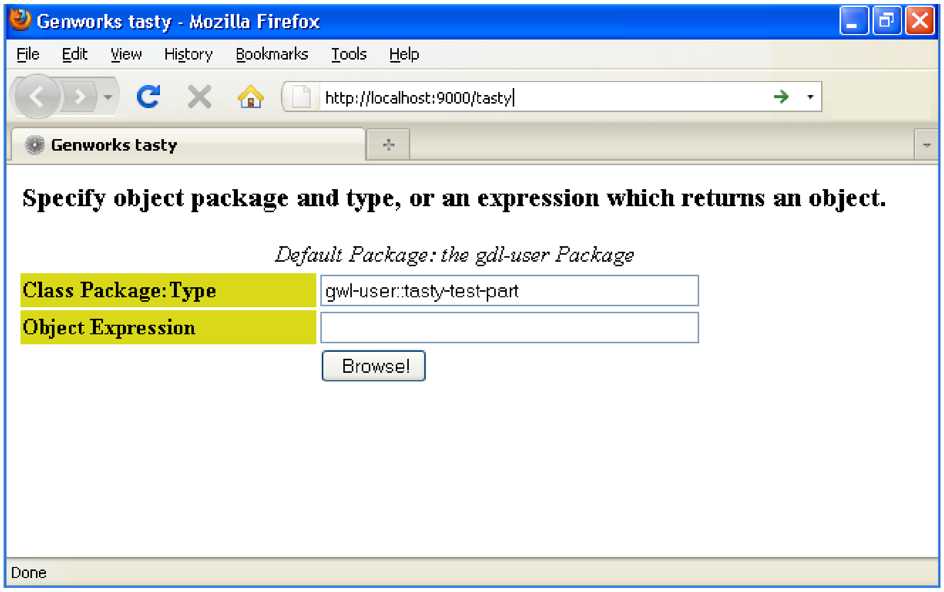

Figure 6.2: Tasty start-up

`tower1` definition in Figure 7.18. This definition is in the `:gdl-user` package. Consequently, the specification will be `gdl-user:tower1`

Note that if the `assembly` symbol had not been exported from the `:chapter-5` package, then a double-colon would have been needed: `chapter-5::tower1`[4]

After you specify the class package and the object type and press the "browse" button, the browser will produce the Tasty interface with an instance of the specified type. The utility interface by default is composed of three toolbars and three view frames (tree frame, inspector frame and viewport frame "graphical view port").

6.0.1 The Toolbars

The first toolbar consists of two "tabs" which allow the user to select between the display of the application itself or the GDL 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[5].

The *tree menu* allows the user to customize the "click mode" of the mouse (or "tap mode" for other pointing devices) for objects in the tree, inspector, or viewport frames. The behavior follows the *select-and-match* behavior – 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:

[4] 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.

[5] 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.

- Tree: Graphical modes

  Add Node (AN) Node in graphics viewport

  Add Leaves (AL) Add Leaves in graphics viewport

  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

- Tree: Inspect & debug modes

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

  Set self to Object (B) Sets a global `self` variable to the selected object, so you can interact by sending messages to the object at the command prompt e.g. by typing `(the length)` or `(the children)`.

  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.

- View: Viewport Actions

  Fit to Window! Fits to the graphics viewport size the displayed objects (use after a Zoom)

  Clear View! (CL!) Clear all the objects displayed in the graphics viewport.

- View: Image Format

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

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

  VRML/X3D Sets the displayed format in the graphics viewport to VRML with default lighting and viewpoint (these can be changed programmatically). This requires a compatible plugin such as BS Contact

X3DOM This experimental mode sets the displayed format in the graphics viewport to use the x3dom.js Javascript library, which attempts to render X3D format directly in-browser without the need for plugins. This works best in WebGL-enabled browsers such as a recent version of Google Chrome[6].

SVG/VML Sets the displayed format in the graphics viewport to SVG/VML[7], which is a vector graphics image format displaying isoparametric curves for surfaces and brep faces.

- View: Click Modes

Zoom in Sets the mouse left-click in the graphics viewport to zoom in.

Zoom out Sets the mouse left-click in the graphics viewport to zoom out.

Measure distance Calculates the distance between two selected points from the graphics viewport.

Get coordinates Displays the coordinates of the selected point from the graphics viewport.

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

- View: Perspective

Trimetric Sets the displayed perspective in the graphics viewport to trimetric.

Front Sets the displayed perspective in the graphics viewport to Front (negative Y axis).

Rear Sets the displayed perspective in the graphics viewport to Rear (positive Y axis).

Left Sets the displayed perspective in the graphics viewport to Left (negative X axis).

Right Sets the displayed perspective in the graphics viewport to Right (positive X axis).

Top Sets the displayed perspective in the graphics viewport to Top (positive Z axis).

Bottom Sets the displayed perspective in the graphics viewport to Bottom (negative Z axis).

The third toolbar hosts the most frequently used buttons. These buttons have tooltips which will pop up when you hover the mouse over them. However, these buttons are found in the second toolbar as well, except for line thickness and color buttons. The line thickness and color buttons[8] expand and contract when clicked, and allows the user to select a desired line thickness and color for the objects displayed in the graphics viewport.

6.0.2 View Frames

The *tree frame* contains a hierarchical representation of your defined object.

To draw the graphics (geometry) for the tower leaf-level objects, you can select the "Add Leaves (AL)" item from the Tree menu, then click the desired leaf to be displayed from the tree. Alternatively, you can select the "rapid" button from the third toolbar which is symbolized by a

[6] Currently, it is necessary to "Reload" or "Refresh" the browser window to display the geometry in this mode.

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

[8] The design of the line thickness and color buttons is being refined and may appear differently in your installation.

pencil icon. Because this operation (draw leaves) is frequently used, the operation is also directly available as a direct-click icon which will appear when you hover the mouse over any leaf or node in the tree.

A direct-click icon is also available for "inspect object," as the second icon when you hover the mouse over a leaf or node.

The "inspector" frame allows the user to inspect (and in some cases modify) the object instance being inspected.

For example, we can make the `number-of-blocks` of the tower to be "settable," by adding the keyword `:settable` after its default expression (please stay tuned for a Chapter on Advanced GDL if you are interested in more details on this GDL syntax). We will also pass in the number-of-blocks as the :size of the `blocks` sequence, rather than using a hard-coded value as previously. The new assembly definition is now:

```
;;
 ;; FLAG -- insert verbatim or ref to new tower code
 ;;
```

In this new version of the tower, the number-of-blocks 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 the Tasty environment.

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. Once the inspector is displaying 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.

Chapter 7

Working with Geometry in GDL

Although Genworks GDL is a powerful framework for a variety of general-purpose undertakings, one of its particular strong points is generating geometry and processing geometric entities. Geometric capabilities are provided by a library of *low-level primitives*, or LLPs. LLPs are pre-defined GDL objects which you can extend by "mixing in" with your own definitions, and/or instantiate as child objects in your definitions.

The names of the geometric LLPs are in the `:geom-base` package, and here are some examples:

- `base-coordinate-system` provides an empty 3D Cartesian coordinate system.[1]
- Simple 2-dimensional primitives include `line`, `arc`, and `ellipse`.
- Simple 3-dimensional primitives include `box`, `sphere`, and `cylinder`.
- Advanced 3-dimensional primitives (which depend on optional add-on Geometry Kernel module) include `b-spline-curve`, `b-spline-surface`, and `merged-solid`.

This chapter will cover the default coordinate system of GDL as well as the built-in simple 2D and 3D LLPs. Chapter **??** will cover the advanced Surfaces and Solids primitives.

7.1 The Default Coordinate System in GDL

GDL's default coordinate system comes with the standard mixin `base-coordinate-system` and represents a standard three-dimensional Cartesian Coordinate system with X, Y, and Z dimensions.

Figure 7.1 shows the coordinate system in a 3D Trimetric view.

Figure 7.2 shows the coordinate system in a Front View.

Figure 7.3 shows the coordinate system in a Top View.

Figure 7.4 shows each face of the reference box labeled with its symbolic direction:

- `Right` for the **positive X** direction
- `Left` for the **negative X** direction
- `Rear` for the **positive Y** direction

[1] `base-coordinate-system` is also known by its legacy name `base-object`

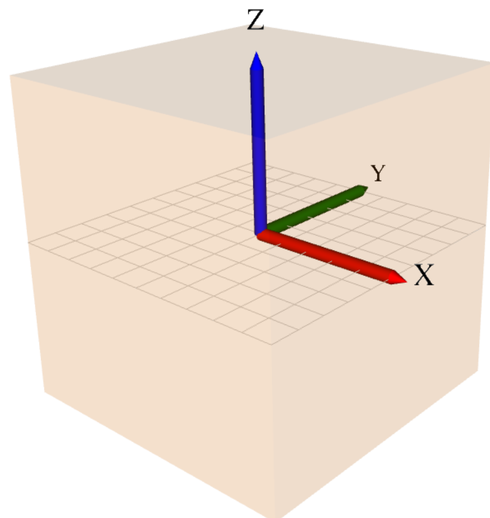

Figure 7.1: Coordinate System in Trimetric View

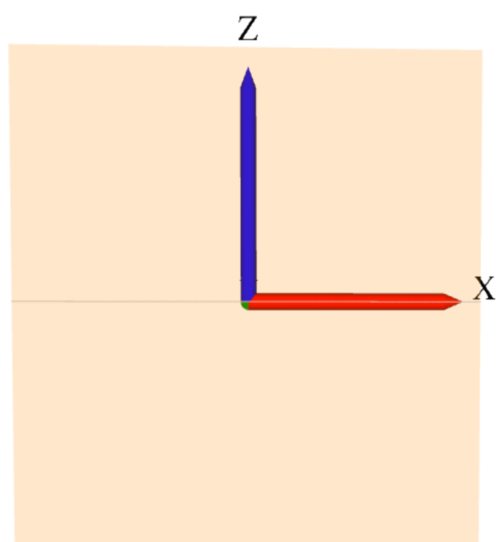

Figure 7.2: Coordinate System in Front View

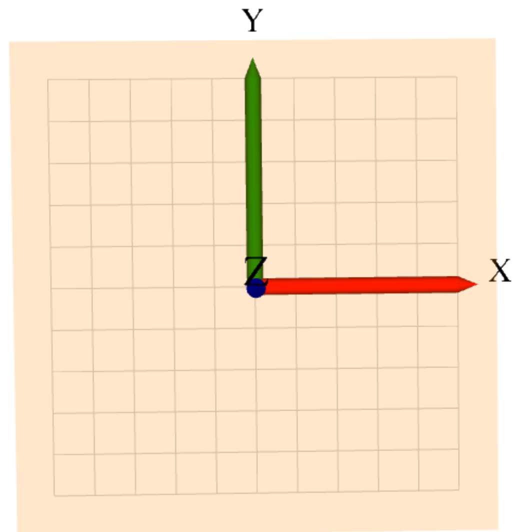

Figure 7.3: Coordinate System in Top View

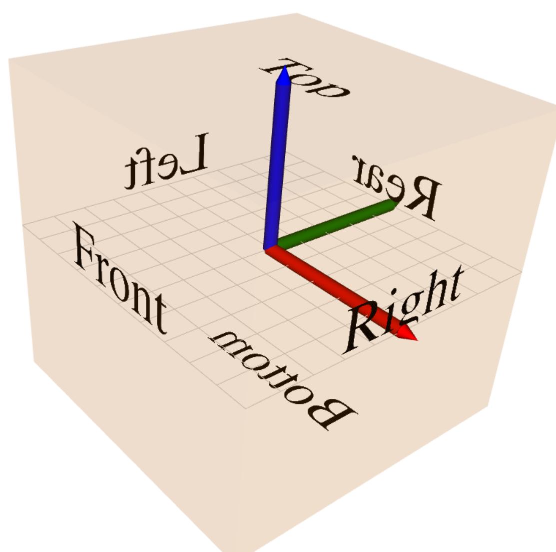

Figure 7.4: Coordinate System with Symbolically Labeled Faces

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

(define-object box-assembly-1 (base-object)

  :computed-slots ((length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+)))
  :objects ((box :type 'box)))
```

Figure 7.5: Definition of a Box

- `Front` for the **negative Y** direction
- `Top` for the **positive Z** direction
- `Bottom` for the **negative Z** direction

7.2 Building a Geometric GDL Model from LLPs

The simplest geometric entity in GDL is a `box`, and in fact all entities are associated with an imaginary *reference box* which shares the same slots as a normal box. The `box` primitive type in GDL inherits its inputs from `base-coordinate-system`, and the fundamental inputs are:

- `center` Default: `#(0.0 0.0 0.0)`
- `orientation` Default: `nil`
- `height` Default: `0`
- `length` Default: `0`
- `width` Default: `0`

Figure 7.5 defines an example box, and Figure 7.6 demonstrates how it will display in Tasty. Note the following from the example in 7.5:

- The symbol `+phi+`[2] holds a global constant containing the "golden ratio" number, which is approximated as 1.618.
- The slots `length`, `width`, and `height` are defined in `base-object` as *trickle-down-slots*. For this reason they are automatically being passed down into into the `box` child object. Therefore it is not necessary to pass them down explicitly.

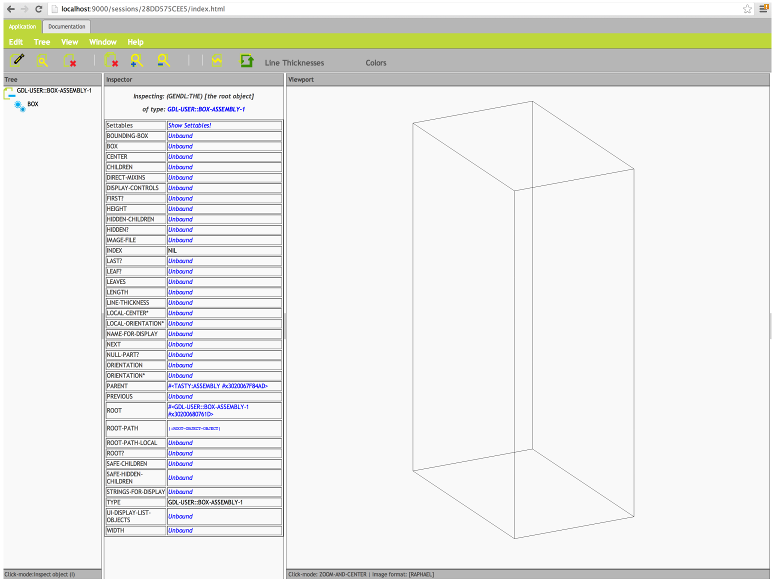

Figure 7.6: Simple box displayed in Tasty

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

(define-object positioned-boxes (base-object)

  :computed-slots ((length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+)))

  :objects ((box-1 :type 'box)
            (box-2 :type 'box
                   :center (make-point (the width)  0 0))))
```

Figure 7.7: Positioned Boxes source

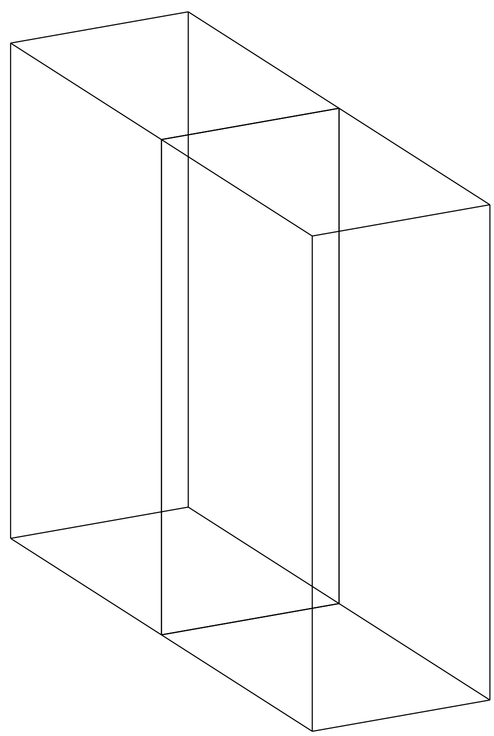

Figure 7.8: Positioned Boxes

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

(define-object positioned-by-index (base-object)

  :input-slots ((number-of-boxes 5))

  :computed-slots ((length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+)))

  :objects ((boxes :type 'box
                   :sequence (:size (the number-of-boxes))
                   :center (make-point (* (the width) (the-child index))
                                       0 0))))
```

Figure 7.9: Positioned by Index source

7.2.1 Positioning a child object using the center input

By default, a child object will be positioned at the same `center` as its parent, and the `center` defaults to the point `#(0.0 0.0 0.0)`. Figure 7.7 (rendered in Figure 7.8) shows a second box being positioned adjacent to the first, by using the `:center` input.

7.2.2 Positioning Sequence Elements using (the-child index)

When specifying a sequence of child objects, each individual sequence element can be referenced from within its `:objects` section using the operator `the-child`. By using `the-child` to send the `index` message, you can obtain the index[3] of each individual child object as it is being processed. In this manner it is possible to compute a distinct position for each child, as a function of its index, as demonstrated in Figures 7.9 and 7.10.

7.2.3 Relative positioning using the translate operator

It is usually preferable to position child objects in a *relative* rather than *absolute* manner with respect to the parent. For example, in our positioned-by-index example in Figure 7.9, each child box object is being positioned using an absolute coordinate produced by `make-point`. This will work as long as the center of the current parent is `#(0.0 0.0 0.0)` (which it is, by default). But imagine if this parent itself is a child of a larger assembly. Imagine further that the larger assembly specifies a non-default center for this instance of `positioned-by-index`. At this point, the strategy fails.

[2]By convention, constants in Common Lisp are named with a leading and trailing + as a way to make them recognizable as constants.

[3]Indices in GDL "size" sequences are integers which start with 0 (zero).

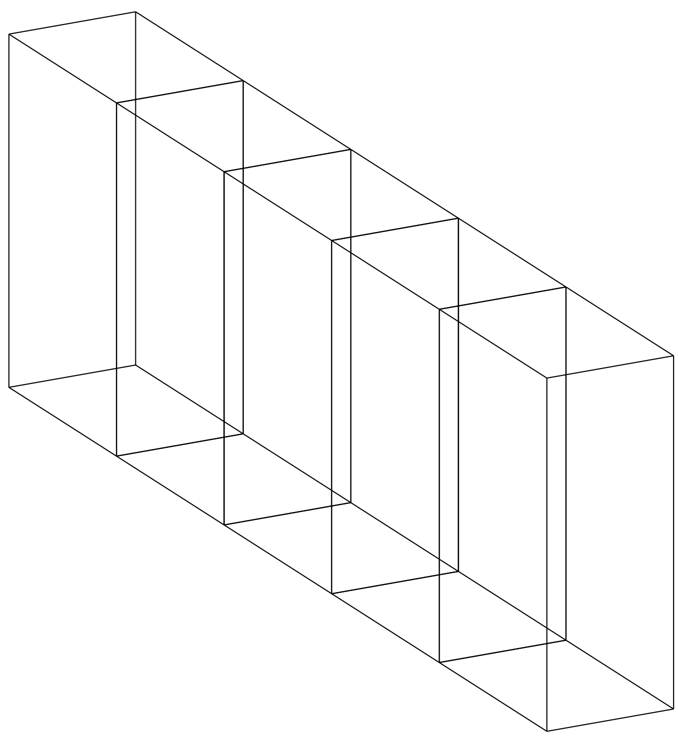

Figure 7.10: Positioned by Index

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

(define-object translate-by-index (base-object)

  :input-slots ((number-of-boxes 5))

  :computed-slots ((length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+)))

  :objects ((boxes :type 'box
                   :sequence (:size (the number-of-boxes))
                   :center (translate (the center)
                                      :right (* (the width) (the-child index))))))
```

Figure 7.11: Translated by Index source

The solution is to adhere to a consistent Best Practice of positioning child objects according to the `center` (or some other known datum point) of the parent object. This can easily be accomplished through the use of the *translate* operator. The `translate` operator works within the context of a GDL object, and allows a 3D point to be translated in up to three directions, selected from: `:up`, `:down`, `:left`, `:right`, `:front`, `:rear`. Figures 7.11 and 7.12 show the equivalent of our positioned-by-index example, but with all the positioning done relative to the parent's center.

7.2.4 Display Controls

It is possible to specify particular default display characteristics[4]

The most common display-control is probably `:color`. Color in GDL can be specified in one of three formats:

1. By name. The names can be seen at the URL `http://localhost:9000/color-map` as seen in Figure 7.13

2. By hexadecimal Red-Green-Blue values, in the form of a string beginning with the "#" character. Each two-digit hex number represents a component of Red, Green, or Blue (to

[4]In addition to display-controls attached to a geometric entity itself, GDL also supports the concept of *lenses*, which capture the program code used to output a particular class of entities (e.g. `box` in a particular output format (e.g. `pdf`. for objects in GDL, such as:

- color
- line-thickness (for line-based output formats like PDF)
- transparency (for shaded graphics outputs like X3D)

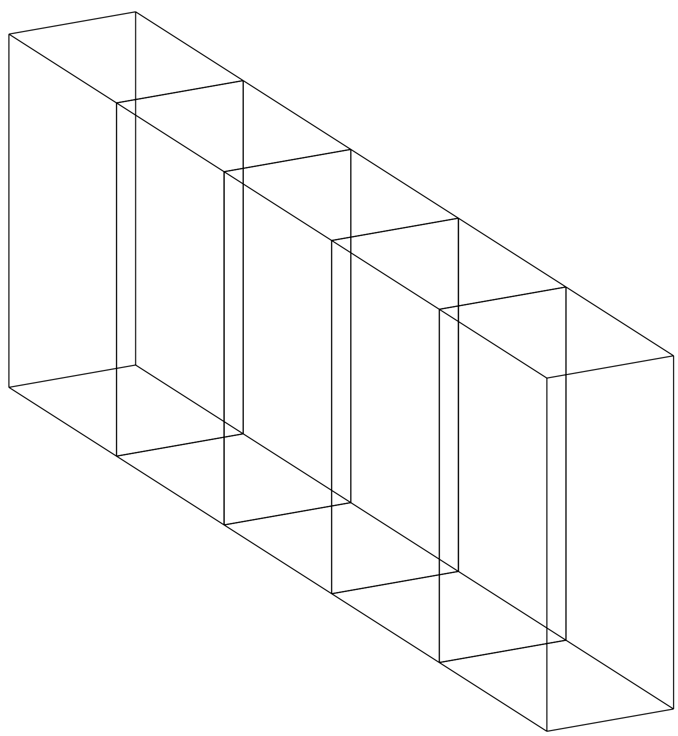

Figure 7.12: Translated by Index

localhost:9000/sessions/48D57C2C73/index.html

Color	Hex	Decimal
:GREEN-PALE	"#8fbc8f"	0.56078434S0, 0.7372549S0
:BLUE-NEON	"#4d4dff"	0.3019608S0, 0.3019608S0,
:GREY	"#c0c0c0"	0.7529412S0, 0.7529412S0,
:GREEN-COPPER	"#527f76"	0.32156864S0, 0.49803922S
:TEAL	"#008080"	0.0S0, 0.5019608S0, 0.5019
:BLUE-STEEL-LIGHT	"#8f8fbd"	0.56078434S0, 0.56078434S
:GREEN-OLIVE-DARK	"#4f4f2f"	0.30980393S0, 0.30980393S
:PURPLE	"#800080"	0.5019608S0, 0.0S0, 0.5019
:BLUE-SLATE	"#007fff"	0.0S0, 0.49803922S0, 1.0S0
:GREEN-LIME	"#32cd32"	0.19607843S0, 0.8039216S0
:SCARLET	"#8c1717"	0.54901963S0, 0.09019608S
:WHITE	"#ffffff"	1.0S0, 1.0S0, 1.0S0
:GREEN-SPRING	"#00ff7f"	0.0S0, 1.0S0, 0.49803922S0
:RED-VIOLET	"#cc3299"	0.8S0, 0.19607843S0, 0.6S0
:WOOD-DARK	"#855e42"	0.52156866S0, 0.36862746S
:BLUE-MEDIUM	"#3232cd"	0.19607843S0, 0.19607843S
:TAN-DARK	"#97694f"	0.5921569S0, 0.4117647S0,
:GOLD-BRIGHT	"#d9d919"	0.8509804S0, 0.8509804S0,
:BLUE	"#0000ff"	0.0S0, 0.0S0, 1.0S0
:YELLOW	"#ffff00"	1.0S0, 1.0S0, 0.0S0

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

(define-object display-color (base-object)

  :input-slots ((number-of-boxes 5))

  :computed-slots ((length 10)
                   (width (* (the length) +phi+))
                   (height (* (the width) +phi+))

                   (color-list (list :red :orange :yellow :blue :indigo :violet)))

  :objects ((boxes :type 'box
                   :sequence (:size (the number-of-boxes))
                   :display-controls (list :color (or (nth (the-child index)
                                                           (the color-list)) :black)
                                           :line-thickness 2)
                   :center (translate (the center)
                                      :right (* (the width) (the-child index))))))
```

Figure 7.14: Color controlled by display-controls source

make this easy to remember, use the mnemonic "Roy G. Biv" from the rainbow colors). For example, `#000000` represents pure Black, and `#FFFFFF` represents pure White. `#FF0000` would be pure Red, `#00FF00` would be pure Green, and `#FF00FF` would be Purple (a mix of Red and Blue). Note that this is also a standard for HTML and the World Wide Web.

3. By a list of three decimal numbers between 0.0 and 1.0, again representing values for Red, Green, and Blue. For example, `(1.0 1.0 1.0)` would be pure White, and `(0.0 0.0 0.0)` would be pure Black.

The `display-controls` is an optional input-slot for any geometric entity in GDL, and is expected to be a *Property List* containing alternating keywords and values. Common keywords for the `display-controls`, corresponding to the display characteristics listed above, are:

- `:color`
- `:line-thickness`
- `:transparency`

Figures 7.15 and 7.14 demonstrate the use of the `:color` keyword in the `display-controls` for our positioned boxes example.

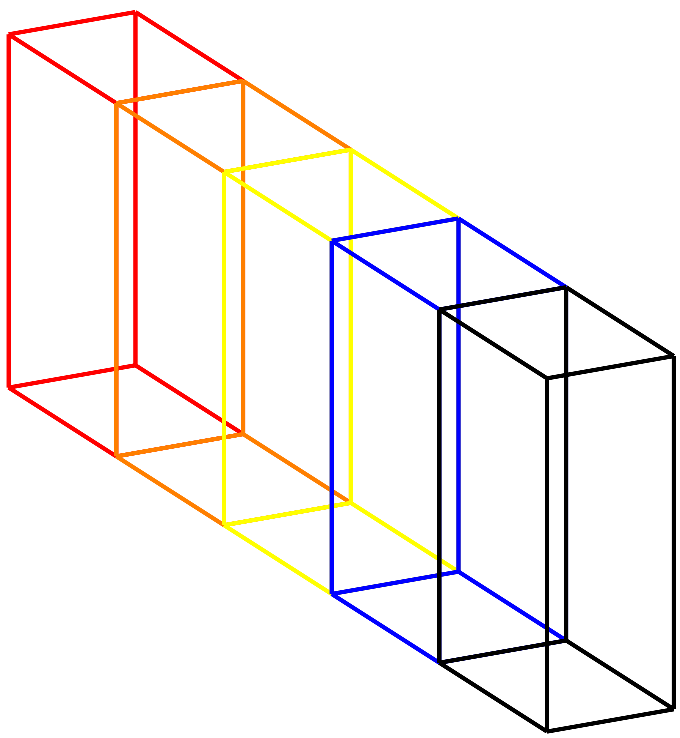

Figure 7.15: Color controlled by display-controls

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

(define-object vertical-cylinder (base-object)

  :objects
  ((horizontal-cylinder :type 'cylinder
                        :display-controls (list :color :green)
                        :length 10 :radius 3)

   (vertical-cylinder :type 'cylinder
                      :length 10 :radius 3
                      :display-controls (list :color :red)
                      :orientation (alignment :rear
                                              (the (face-normal-vector :top))))))
```

Figure 7.16: Cylinder aligned vertically source

7.2.5 Orientation and the Alignment function

Orientations in GDL are specified using a 3x3 orientation matrix. The simplest way to compute an orientation matrix is to use the use the `alignment` function. The `alignment` function accepts up to three direction keywords, and corresponding vectors to which these directions should be aligned. For example, to obtain an orientation matrix specifying that the Rear of a reference box should be aligned with the vector `#(1.0 0.0 0.0)`, you could call

```
(alignment :rear (make-vector 1 0 0))
```

Generally, you will want the orientation of a child object to be specified in a *relative* manner to that of the current (parent) object. The concept here is similar to that for positioning with respect to `(the center)`. For relative orientation, you can utilize the various `face-normal-vectors` of the parent object. For example, by default, cylinders are aligned with their flat ends along the longitudinal (Y) axis. Figures 7.16 and 7.17 show the red cylinder which is turned to be vertical (aligned to the Z axis), by aligning its `:rear` face with `(the (face-normal-vector :top))` of the parent base-object.

7.2.6 Rotating vectors with the rotate-vector-d function

In order to specify a vector which is not aligned exactly with one of the major axes, you can use the `rotate-vector-d` function to yield a new vector which is the result of "rotating" one vector about another vector. Figures 7.18 and 7.19 show a stack of boxes, where the rear face of each box is rotated 2 degrees with respect to the box under it.

7.2.7 Assemblies

Objects which you define with `define-object` can be used no differently from the built-in primitives. This underscores why it is important for the positioning and orientation passed into a child

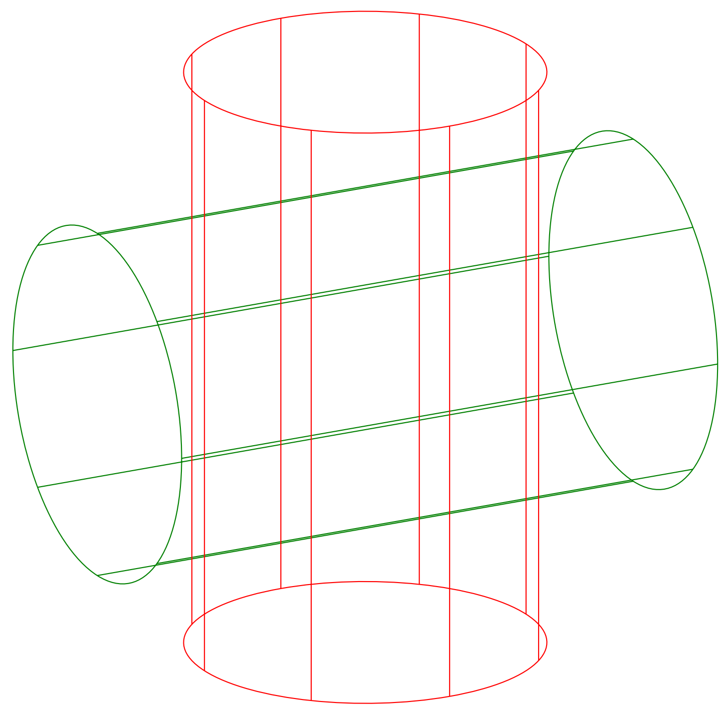

Figure 7.17: Cylinder aligned vertically

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

(define-object tower (base-object)
  :input-slots
  ((height 42)
   (block-height 1)
   (width +phi+)
   (length (* (the width) +phi+)))

  :computed-slots
  ((number-of-blocks (floor (the height)
                            (the block-height))))

  :objects
  ((blocks :type 'box
           :sequence (:size (the number-of-blocks))
           :length (the length)
           :height (the block-height)
           :width (the width)
           :center (translate (the center) :up
                              (* (the-child height)
                                 (the-child index)))
           :orientation (alignment :rear (rotate-vector-d
                                          (the (face-normal-vector :rear))
                                          (twice (the-child index))
                                          (the (face-normal-vector :top)))))))
```

Figure 7.18: Twisty Tower source

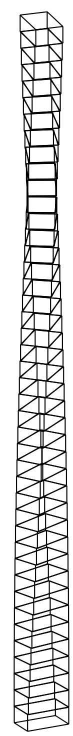

Figure 7.19: Twisty Tower

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

(define-object tower-assembly (base-object)
  :input-slots
  ((base-height 10)
   (height-deviation 5)
   (number-of-towers 5))


  :objects
  ((towers :type 'tower
           :sequence (:size (the number-of-towers))
           :height (+ (* (the-child index) (the height-deviation))
     (the base-height))
           :center (translate (the center) :right (* (twice
     (twice (the-child width)))
    (the-child index))))))
```

Figure 7.20: Tower Assembly source

object be *relative* to that present in the parent. Figures 7.21 and 7.20 show how several towers can be positioned side-by-side, while maintaining consistent internal positioning and orientation. Figure 7.22 shows how the child towers form an *assembly hierarchy* of objects.

7.2.8 Mechanisms

GDL supports mechanisms without need for any special features. By defining position and orientation of some objects to be dependent on others, you can set up a mechanism. Figure 7.23 shows a standard four-bar link mechanism which is defined in the code in the file 4-bar-assembly.gdl (this is in the examples directory[5] — due to its length, the source is not printed in the manual.

7.2.9 Other Geometric Primitives

This chapter has focused primarily on the `box` primitive, because every type of geometric primitive is based upon a *reference box*. Other primitives have their own sets of input-slots, and their own ways of being rendered in the various output formats. Basic 2D primitives include:

- `circle` described on page 116
- `line` described on page 138
- `arc` described on page 100

[5] http://github.com/genworks/gendl/tree/master/documentation/tutorial/examples/

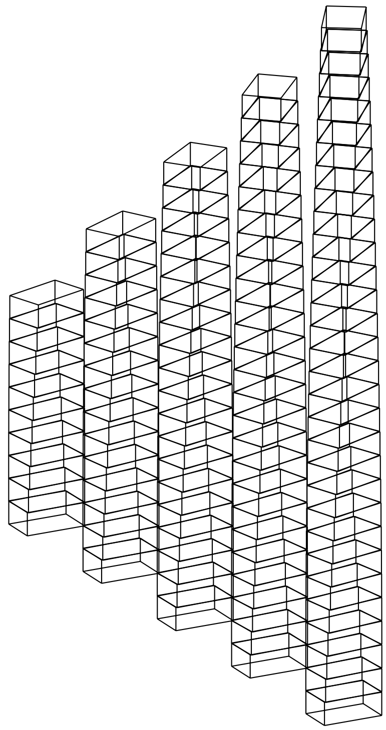

Figure 7.21: Tower Assembly

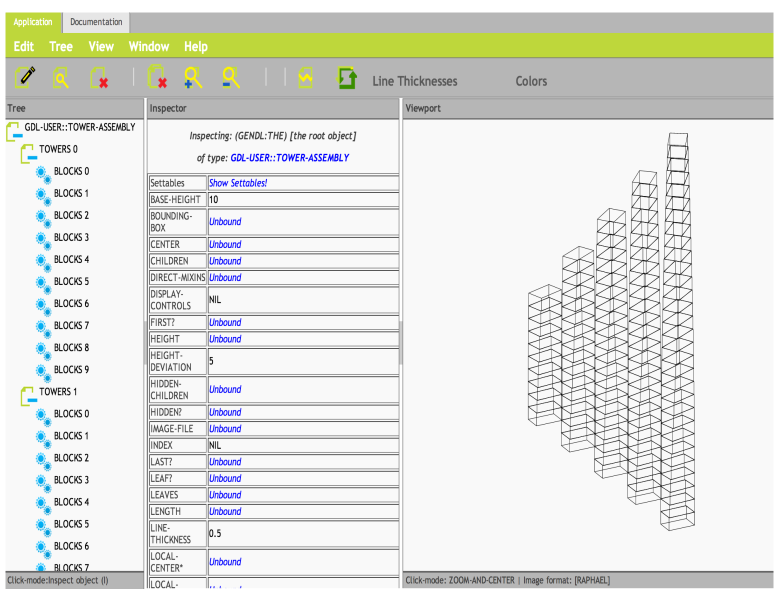

Figure 7.22: Tower Assembly as displayed in Tasty

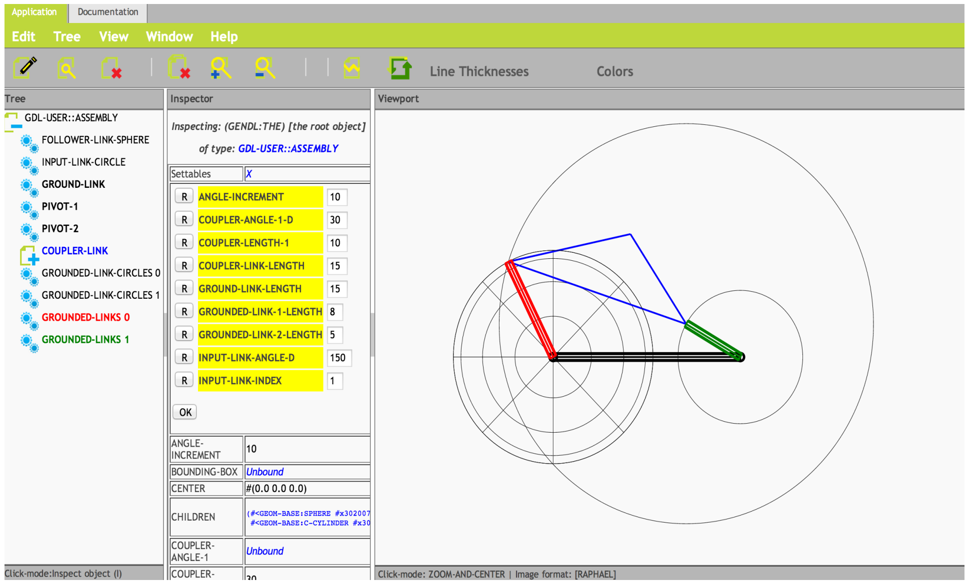

Figure 7.23: Four-bar Link Mechanism

- `ellipse` described on page 122
- `bezier-curve` described on page 110[6]

Basic 3D primitives include:

- `sphere` described on page 150
- `cylinder` described on page 120
- `cone` described on page 117
- `global-polyline` described on page 131
- `global-polygon-projection` described on page 130
- `global-filleted-polyline` described on page 131
- `torus` described on page 154
- `route-pipe` described on page 148

[6] The simple cubic bezier curve is supported in the basic GDL and open-source Gendl. More sophisticated NURBS-based curves and surfaces are supported in the commercial GDL product when accompanied with the SMLib geometry kernel.

Chapter 8

Custom User Interfaces in GDL

Another strength of GDL is the ability to create custom web-based user interfaces. GDL 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 encountered, you can define web pages, sections of web pages, and form elements. Your UI elements generate standard HTML, CSS, JavaScript, and can use any external client-side libraries as with any standard web application. AJAX support is built-in, making it easy for server and UI elements to communicate asynchronously.

In order to create a web application in GDL, you should have a working knowledge of the semantics of HTML, for which many explanations are available online and in print. For syntax, you can use an HTML-generation library such as CL-WHO[2] or HTMLGen[3], both of which are built into GDL. In this tutorial we will use CL-WHO, so in what folows, we will assume that you are already familiar with its features as documented in http://weitz.de/cl-who.

8.1 Package and Environment for Web Development

The `:gwl` package[4] contains a set of primitive objects and functions provided by GDL for building web applications.

Similarly to `gdl:define-package`, you can use `gwl:define-package` to create a working package which has access to the symbols you will need for building a web application (in addition to the other GDL 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 YADD[5] reference documentation for package GWL provides detailed specifications for all the primitive objects and functions.

---

[1] GDL 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, GDL does not provide specific support for them.

[2] http://weitz.de/cl-who

[3] http://www.franz.com/support/documentation/current/doc/aserve/htmlgen.html

[4] The acronym "GWL" stands for Generative Web Language

[5] YADD is accessible with `http://localhost:9000/yadd`.

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

;; http://localhost:9000/make?object=gwl-user::hello-world
(define-object hello-world (base-ajax-sheet)
  :computed-slots
    ((main-sheet-body
       (with-cl-who-string ()
         (:p "Hello World!")))))
```

Figure 8.1: Simple Static Page

8.2 Web Page Objects

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

1. Mix `base-ajax-sheet` into the object definition.
2. Within the object, define the GDL message `main-sheet-body` returning an HTML string.

As soon the object is defined, the object's page becomes accessible at the URL

`http://localhost:9000/make?object=`*`classname`*

where *`classname`* is the object name (including the package name). Connecting to this URL starts a new web server session and creates an instance of the object unique to that session[6]. This ensures that each user of your application site will see their own specific instance of the object.

The `main-sheet-body` message can be either a computed slot or a GDL function. It should produce a string of the HTML to be placed in the body of the page, i.e. between the <body> and </body> tags. The <head> of the page is filled in automatically by GDL, and can be customized in various ways.

The easiest way to produce a valid `main-sheet-body` string is with the GWL convenience macro `with-cl-who-string`. This is a wrapper for the CL-WHO macro `with-html-output` which additionally establishes the default environment for outputting an HTML string within a GWL application.

Figure 8.1 is an example of a simple static web page.

This simple framework can also be used to create dynamic content, as illustrated in figure 8.2. Note that CL-WHO symbols such as `htm`, `str`, and `fmt` are available in GWL without package qualification. See http://weitz.de/cl-who for an explanation of dynamic HTML generation in CL-WHO.

8.3 Page URLs

Objects based on `base-ajax-sheet` are automatically available from the server at the URL of the form `http://localhost:9000/make?object=`*`classname`*. This is useful during debugging,

[6]This is done by redirecting the response to a unique URL identified by a *session ID*. The session ID is constructed from a combination of the current date and time, along with a pseudo-random number.

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

(define-object president ()
  :input-slots
  (name term))

;; http://localhost:9000/make?object=gwl-user::presidents
(define-object presidents (base-ajax-sheet)
  :input-slots
  ((data (list (list :name "Carter" :term 1976)
               (list :name "Reagan" :term 1980)
               (list :name "Bush" :term 1988)
               (list :name "Clinton" :term 1992))))

  :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)))

  :computed-slots
  ((main-sheet-body
    (with-cl-who-string (:indent t)
      (htm
       (:p (:c (:h3 (fmt "Info on ~a Presidents:"
                         (length (list-elements (the presidents)))))))
       ((:table :border 1)
        (:tr (:th "Name") (:th "Term"))
        (dolist (president (list-elements (the presidents)))
          (htm
           (:tr (:td (str (the-object president name)))
                (:td (str (the-object president term)))))))))))))
```

Figure 8.2: Dynamic Content Using CL-WHO

but for real projects, you should define a more direct URL. You do that using the CL function `(gwl:publish-gwl-app` *`path classname`*`)`. For example,

```
(gwl:publish-gwl-app "/greeting" "gwl-user::hello-world")
```

will make the `gwl-user::hello-world` object accessible at the URL `http://localhost:9000/greeting`.

8.4 Page Customizations

In addition to specifying the body html via `main-sheet-body`, you can customize the page by optionally adding certain messages to your object:

- Message `title` can be used to specify the title of the web page
- Message `doctype-string` is the string to place at the very start of the document (it defaults to `"<!DOCTYPE HTML>"`)
- Message `additional-header-content` can contain any additional HTML you want to go into the page's `<head>` section.
- Message `body-class` can specify the CSS `class` attribute for the `<body>` tag.
- Message `body-onload` can be a string of Javascript to go into the `onload` event attribute of the `<body>` tag.
- Message `body-onpageshow` can be a string of Javascript to go into the `onpageshow` event attribute of the `<body>` tag.

We will be using some of these customizations in the examples below.

8.5 Debugging

`base-ajax-sheet` provides the `development-links` message with links for functionality useful during development, currently consisting of a Refresh! link and a Break link. It is typically used as follows:

```
(main-sheet-body
  (with-cl-who-string ()
    (when *developing?* (str (the development-links)))
    ;; Rest of page definition goes here
    ...))
```

8.6 Page Links

In order to allow HTML links between page objects, GDL implements a scheme that assigns URL's to individual page instances. In order to do this efficiently, GDL maintains a table of root-level instances, and identifies page instances relative to their root instance. For this reason, and in order for dependency-tracking to work properly, all pages in a GDL application must belong to the same tree, i.e. they must share a common root page object.

You can generate a hyperlink (an `<A>` tag) to a particular `base-ajax-sheet` page by invoking the `write-self-link` GDL function on the page instance. `write-self-link` accepts a number of keyword arguments to customize the link:

- You specify the text to be displayed as the link with the `:display-string` argument
- You can direct the link to a specific anchor within the page with the `:local-anchor` argument
- You can specify various tag attributes of the `<A>` tag:
  - Argument `:target` for the `target` attribute
  - Argument `:on-mouse-over` for the `onmouseover` attribute
  - Argument `:on-mouse-out` for the `onmouseout` attribute
  - Argument `:on-click` for the `onclick` attribute
  - Argument `:title` for the `title` attribute
  - Argument `:class` for the `class` attribute
  - Argument `:id` for the `id` attribute

In order to help create hierarchical multi-page sites, `base-ajax-sheet` also provides a `write-back-link` GDL function for the purpose of generating a link back to the parent page object. It accepts all the same arguments as `write-self-link`.

Figure 8.3 shows a modification of 8.2 where we put detailed information about each president on a separate page, and provide links to these individual pages from a summary page. Each child page contains a link to return back to the summary.

8.7 Input Using Forms

GWL provides a macro `with-html-form` to obtain input from the user with an HTML form. In the body of the form, you specify input fields using standard HTML form controls. When the user submits the form, GDL processes the form input values in two ways:

- When the name of an form field matches the name of a `:settable` computed-slot in the object, GDL will automatically infer its type, do appropriate conversions, and set the slot the its new value[7].

[7] If the type of a slot can vary, it is best to make its default be a string and parse the string yourself e.g. with the `read-safe-string` function.

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

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

  :computed-slots
  ((title
    (format nil "Term for President ~a:" (the name)))
   (main-sheet-body
    (with-cl-who-string (:indent t)
      (the (write-back-link :display-string "&lt;Back"))
      (:p (:c (:h3 (the title))))
      (:p (str (the term)))))))


;; http://localhost:9000/make?object=gwl-user::presidents-with-links
(define-object presidents-with-links (base-ajax-sheet)
  :input-slots
  ((data (list (list :name "Carter" :term 1976)
               (list :name "Reagan" :term 1980)
               (list :name "Bush" :term 1988)
               (list :name "Clinton" :term 1992)))
   (table-border 1))

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

  :computed-slots
  ((title (format nil "Links to ~a Presidents:"
                  (length (list-elements (the presidents)))))
   (main-sheet-body
    (with-cl-who-string (:indent t)
      (htm
       (:p (:c (:h3 (the title))))
       ((:table :border (the table-border))
        (:tr (:th "Name"))
        (dolist (president (list-elements (the presidents)))
          (htm
           (:tr (:td (the-object
                      president
                      (write-self-link :display-string
                                       (the-object president name))))))))))))
```

Figure 8.3: Hyperlinking

- Any input values that do not match a suitable slot are available from the `query-plist` message, which returns a plist mapping keywords representing form field names to the corresponding input value strings

Figure 8.4 shows how we can extend the simple Hello World application of 8.1 to allow the user to customize both the name and the greeting.

8.7.1 Form Controls

GDL provides a set of primitives useful for generating the standard HTML form controls in the body of `with-html-form`. These should be instantiated as child objects in the page. They provide a `html-string` message which returns the HTML for the control.

The form controls provided by GDL are documented in YADD[8] and in the reference appendix of this manual. Examples of available form controls are:

- `text-form-control`
- `checkbox-form-control`
- `menu-form-control`
- `radio-form-control`
- `text-form-control`
- `button-form-control`

These form controls are customizable by mixing them into your own specific form controls (although this often is not necessary). New form controls such as for numbers, dates, etc will soon be added to correspond to latest HTML standards.

Figure 8.5 reimplements the Hello World form from the last section using form controls. The functionality is the same but the source is shorter and simpler to read and understand.

8.8 Interactive Applications using AJAX

Input using traditional HTML forms requires the user to explicitly "submit" each update request, and then the application responds by reloading the whole page. The AJAX[9] methodology allows web applications to autonomously contact the server, and to change page content dynamically in response, leading to a more interactive user experience.

GDL has built-in support for AJAX, using its usual convenient generative approach. You use child objects to model independently updatable sections of the web pgae, and the dependency tracking engine which is built into GDL automatically manages of which sections of the page need to be updated after a request.

Moreover, the state of the internal GDL 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 ability is not available in some other AJAX frameworks.

[8] YADD is accessible with `http://localhost:9000/yadd`.

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

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

;; http://localhost:9000/make?object=gwl-user::hello-world-form
(define-object hello-world-form (base-ajax-sheet)
  :input-slots
  ((greetings '("Hello"
                "Hola"
                "Bonjour"
                "Habari"
                "Namaste"
                "Aloha"
                "Ciao")))

  :computed-slots
  ((username "World" :settable)
   (greeting (first (the greetings)) :settable)

   (title (format nil "Greeting for ~a" (the username)))
   (main-sheet-body
    (with-cl-who-string ()
      (:p (fmt "~a ~a!" (the greeting) (the username)))
      (:hr)
      (with-html-form (:cl-who? t)
        (:p
         (:label :for 'greeting "Select greeting: ")
         ((:select :name 'greeting)
          (dolist (val (the greetings))
            (with-cl-who ()
              ((:option :value val
                        :selected (and (equalp val (the greeting)) "selected"))
               (esc val))))))
        (:p
         (:label :for 'username "Enter new name: ")
         (:input :type :text :name 'username :value (the username)))
        (:p (:button :type :submit "UPDATE")))))))
```

Figure 8.4: Input Using Forms

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

;; http://localhost:9000/make?object=gwl-user::hello-world-controls
(define-object hello-world-controls (base-ajax-sheet)
  :input-slots
  ((greetings '("Hello"
                "Hola"
                "Bonjour"
                "Habari"
                "Namaste"
                "Aloha"
                "Ciao")))

  :objects
  ((username-control :type 'text-form-control
                     :prompt "Enter new name: "
                     :default "World")
   (greeting-control :type 'menu-form-control
                     :prompt "Select Greeting: "
                     :size 1 :choice-list (the greetings)
                     :default (first (the greetings))))

  :computed-slots
  ((title (format nil "Greeting for ~a" (the username-control value)))
   (main-sheet-body
    (with-cl-who-string ()
      (:p (fmt "~a ~a!" (the greeting-control value) (the username-control value)))
      (:hr)
      (with-html-form (:cl-who? t)
        (:p (str (the greeting-control html-string)))
        (:p (str (the username-control html-string)))
        (:p (:button :type :submit "UPDATE")))))))
```

Figure 8.5: Using Form Controls

8.8.1 AJAX Event Handling

An AJAX application works by contacting the server in response to HTML events[10] such as "onclick" or "onfocus". GDL provides the function `gdl-ajax-call` to generate the JavaScript to handle such events by invoking GDL functionality on the server.

For example, the following CL-WHO snippet

```
((:span :onclick (the (gdl-ajax-call :function-key :restore-defaults!)))
       "Press Me")
```

will produce a piece of text "Press Me" which, when pressed, will call `restore-defaults!` in the page's object on the server. If `restore-defaults!` is not defined, an error will result.

`gdl-ajax-call` can also update form control values on the server by using the `:form-controls` keyword argument. For details, see the `gdl-ajax-call` documentation in YADD[11] and the reference appendix.

For convenience, GDL form-control objects provide direct support for AJAX with the `:ajax-submit-on-change` argument, which is equivalent to invoking `gdl-ajax-call` for the control's "onchange" event.

8.8.2 AJAX Page Updating

In order to have the capacity to change itself in response to AJAX calls, a page must be structured as one or more `sheet-section` child objects. Sheet sections provide a `main-div` message which computes the HTML for the section and registers the section as subject to AJAX update handling. The main page should include the `main-div` of each child sheet section in its `main-sheet-body`.

The sheet section definition should specify the section's HTML in the `inner-html` input slot. This plays the same role as the `main-sheet-body` does in the page.

Figure 8.6 reimplements the Hello World form from the last section using AJAX. Note that we don't need the explicit UPDATE button any more, as the changes the user makes take effect immediately due to the use of `:ajax-submit-on-change?` argument in each of the form controls.

8.8.3 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 are 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 example in Figure 8.7 contains a simple box with two graphics viewports and ability to modify the length, height, and and width of the box:

This will produce a web browser output similar to what is shown in Figure 8.8.

Note the following from this example:

- The `(:use-raphael? t)` enables raphael for SVG or VML output.

[10] https://en.wikipedia.org/wiki/DOM_events

[11] YADD is accessible with `http://localhost:9000/yadd`.

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

;; http://localhost:9000/make?object=gwl-user::hello-world-ajax
(define-object hello-world-ajax (base-ajax-sheet)
  :input-slots
  ((greetings '("Hello"
                "Hola"
                "Bonjour"
                "Habari"
                "Namaste"
                "Aloha"
                "Ciao")))

  :objects
  ((username-control :type 'text-form-control
                     :prompt "Enter new name: "
                     :default "World"
                     :ajax-submit-on-change? t)
   (greeting-control :type 'menu-form-control
                     :prompt "Select Greeting: "
                     :size 1 :choice-list (the greetings)
                     :default (first (the greetings))
                     :ajax-submit-on-change? t)

   (main-section
    :type 'sheet-section
    :inner-html (with-cl-who-string ()
                 (:p (fmt "~a ~a!" (the greeting-control value) (the username-control value)))
                 (:hr)
                 (with-html-form (:cl-who? t)
                   (:p (str (the greeting-control html-string)))
                   (:p (str (the username-control html-string)))))))

  :computed-slots
  ((title (format nil "Greeting for ~a" (the username-control value)))
   (main-sheet-body
    (with-cl-who-string ()
      (str (the main-section main-div))))))
```

Figure 8.6: Using AJAX

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

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

  :computed-slots
  ((use-raphael? t)
   (use-x3dom? 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
;;
```

Figure 8.7: Including Graphics in a Web Page

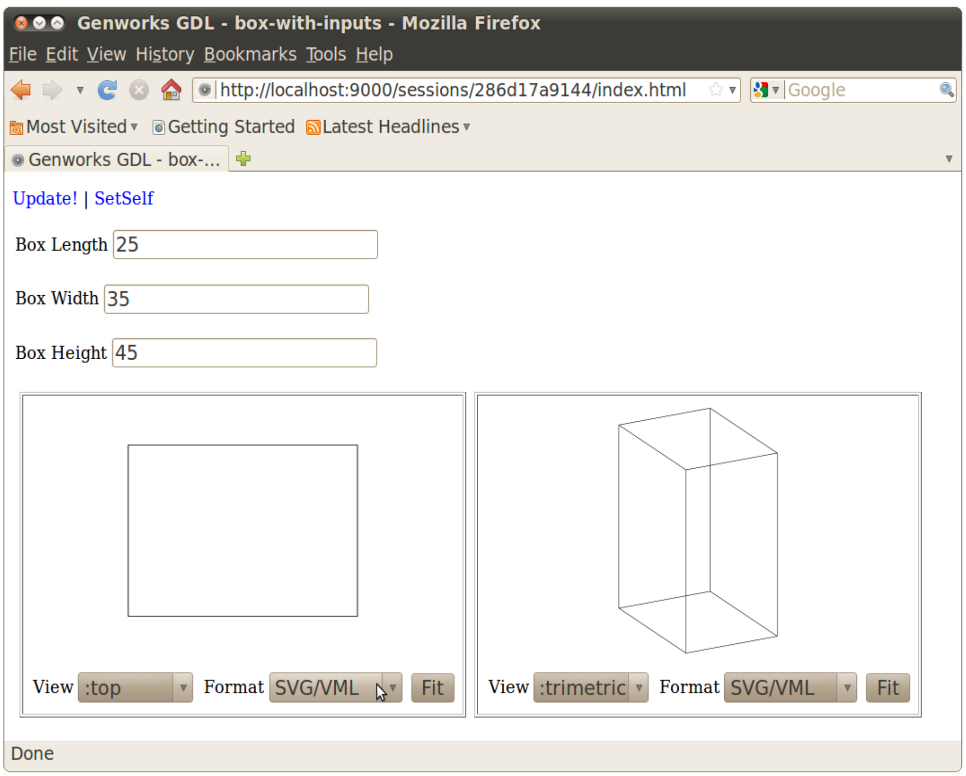

Figure 8.8: Including Graphics

- 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-body`, 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).

Upgrade Notes

GDL 1580 marked the end of a major branch of GDL development, and 1581 was an upgraded new version, which has now been supplanted by 1592.

This addendum lists the typical modifications you will want to consider for upgrading from versions of GDL prior to 1582.

- (make-gdl-app ..) is now available for 1582. We have made available an Enterprise Edition of 1582 which includes the make-gdl-app function, which creates Runtime applications without the compiler or GDL development facilities. If you are an Enterprise licensee, and are ready to release Runtime applications on 1582, and you have not received information on the Enterprise Edition, please contact support@genworks.com

- (register-asdf-systems) and the `"3rdpty/"` directory are no longer needed or available. Instead, we depend on the Quicklisp system. Details of Quicklisp are available at `http://www.quicklisp.org`. See Section 3.3.4 for information about how to use Quicklisp with GDL.

- There is a system-wide `gdlinit.cl` in the application directory, and depending on the particular release you have, this may have some default information which ships with GDL. There is a personal `gdlinit.cl` in home directory, which you should modify if you want to customize anything.

- Slime debugging is different from the ELI emacs debugger. The main thing to know is to press "a" or "q" to pop out of the current error. Full documentation for the Slime debug mode is available with the Slime documentation.

- color-themes – GDL now ships with the Emacs color-theme package. You can select a different color theme with `M-x color-theme-select`. Press [Enter] or middle-mouse on a color theme to apply it.

- GDL files can now end with `.lisp` or `.gdl`. The new `.gdl` extension will work for emacs Lisp mode and will work with cl-lite, ASDF, and Quicklisp for including source files in application systems. We recommend migrating to the new `.gdl` extension for files containing `define-object`, `define-format`, and `define-lens` forms, and any other future toplevel defining forms introduced by GDL, in order to distinguish from files containing raw Common Lisp code.

- in gdlAjax, HTML for a sheet-section is given in the slot called `inner-html` instead of `main-view`. This name change was made to clarify what exactly is expected in this slot – it is the innerHTML of the page division represented by the current sheet-section. If you want to make your code back-compatible with GDL 1580, you can use the following form in place of old occurences of `main-view`:

```
... #+allegro-v8.1 main-view #-allegro-v8.1 inner-html ...
```

- (update-gdl ..) is not available for 1592. Instead of updating incrementally with patches, the intention starting with GDL 1592 is for full GDL releases to be made available approximately monthly. Less frequent Long Term Maintenance (“LTS”) releases will also be made available along with a new simpler maintenance patch system.

Chapter 9

Reference for GDL Objects and Operators

9.1 cl-lite (Compile-and-Load Lite Utility)

9.1.1 Object Definitions

- **codebase-directory-node**

  Mixins: directory-node

  Description Models a filesystem directory for use by the cl-lite program.

  Input slots (optional):

Bin-subdir-names *List of strings*
: Identifies the names of directories considered to hold binaries. Default is (list "bin" "patch")

Create-fasl? *Boolean*
: Determines whether to write a concatenated fasl for the build. Defaults to nil. NOTE: this is not currently supported in cl-lite.

Fasl-output-name *String*
: Names the built concatenated fasl when `(the create-fasl?)` is non-nil. Defaults to `(the local-name)`

Fasl-output-path *String or pathname object*
: Designates the pathname for the filesystem directory in which the built concatenated fasls are written. Defaults to `(glisp:temporary-folder)`

Fasl-output-type *String*
: Names the fasl extension used by the compiler. Defaults to the local fasl output type.

Load-always? *Boolean*
: Determines whether to load the individual compiled fasls even if the source has not changed. Defaults to nil (i.e. we assume we are loading into a clean system and need all the initial definitions.).

Source-files-to-ignore *List of strings*

Lists directory names which should be ignored as having compilable source code for the build.

Special-subdir-names *List of strings*

Identifies the names of directories which are part of a vc-system control files and therefore should be treated as special subdirectories. Default is (list "CVS")

Type-mapping *Plist of keywords and lists of strings*

Maps directory names to their default type classifications.

Computed slots:

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.

9.1.2 Function and Macro Definitions

- **cl-patch**

  (type nil intro Traverses pathname in a manner identical to cl-lite, but only those files for which the source is newer than the corresponding fasl binary file (or for which the corresponding fasl binary file does not exist) will be loaded. Use this for incremental updates where the unmodified source files do not depend on the modified source files.)

9.2 com.genworks.dom

9.3 com.genworks.dom-html

9.4 com.genworks.dom-latex

9.5 com.genworks.dom-writers

9.6 com.yoyodyne.booster-rocket

9.7 enterprise

9.8 gendl (Base Core Kernel Engine) Nicknames: Gdl, Genworks, Base

9.8.1 Object Definitions

- **base-rule-object**

  Mixins: vanilla-mixin

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."

Input slots (optional):

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.

- **matrix-sequence**

Mixins: standard-sequence, vanilla-mixin

Description A matrix sequence quantification is generated as a result of specifying `:sequence (:matrix direction-keyword number direction-keyword number))` in an `:objects` specification. The direction-keywords can be one of `:lateral`, `:longitudinal`, and `:vertical`. The items will be arranged spread out evenly in the directions specified. Centers can also be provided explicitly based on the indices. The indices to a matrix sequence consist of a list of numbers rather than a single number as with a normal sequence.

Computed slots:

First *GDL Object*

Returns the first element of the aggregate.

Last *GDL Object*

Returns the last element of the aggregate.

- **null-object**

  Mixins: vanilla-mixin

  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.

- **quantification**

  Mixins: vanilla-mixin

  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.

  Index *Integer*
  Sequential index number for elements of a sequence, NIL for singular objects.

  Last *GDL Object*
  Returns the last element of the aggregate.

- **radial-sequence**

  Mixins: standard-sequence, vanilla-mixin

  Description A radial sequence quantification is generated as a result of specifying `:sequence (:radial [number-expression]))` in an `:objects` specification.

- **standard-query-row**

  Mixins: vanilla-mixin

  Description Represents a row of data.

  Input slots (optional):

  Type *Symbol*
  The GDL Type of this object.

- **standard-sequence**

  Mixins: quantification

Description A standard sequence quantification is generated as a result of specifying `:sequence (:size [number-expression]))` in an `:objects` specification. Unlike a variable-sequence quantification (specified with `:sequence (:indices ...))`), elements cannot be surgically inserted or deleted from a standard sequence. If a value upon which the [number-expression] depends becomes modified, each member of the sequence will be reinstantiated as it is demanded.

Computed slots:

First *GDL Object*

Returns the first element of the aggregate.

Last *GDL Object*

Returns the last element of the aggregate.

- **vanilla-mixin***

Mixins: standard-object

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 specified at the toplevel of the define-object form.

Input slots (optional):

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).

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

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-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.

Gdl 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.

Mixins *List of Symbols*

Returns the names of the immediate superclasses of this object.

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.

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.

Note also that this must not be called (either directly or indirectly) from within the body of a Gendl computed-slot. The caching and dependency tracking mechanism in Gendl will not work properly if this is called from the body of a computed-slot, and furthermore a runtime error will be generated.

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 throughout 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.

- **variable-sequence**

  Mixins: quantification

Description A variable-sequence quantification is generated as a result of specifying `:sequence (:indices ...))` in an `:objects` specification. Unlike a normal sequence quantification (specified with `:sequence (:size ...)))`, elements can be surgically inserted and deleted from a variable-sequence.

Computed slots:

First *GDL Object*

Returns the first element of the aggregate.

Last *GDL Object*

Returns the last element of the aggregate.

Gdl functions:

Delete! *Void*

Deletes the element identified with the given index.

Insert! *Void*

Inserts a new element identified with the given index.

Reset! *Void*

Resets the variable sequence to its default list of indices (i.e. clears out any inserted or deleted elements and re-evaluates the expression to compute the original list of indices)

9.8.2 Function and Macro Definitions

- **alist2plist**

  (type Plist intro Converts an assoc-list to a plist. arguments (alist Assoc-List))

- **always**

  (type T intro Always returns the value ¡tt¿T¡/tt¿ regardless of ¡b¿arg¡/b¿. arguments (arg Lisp object. Ignored))

- **append-elements [Macro]**

  (type List of Objects [Macro] intro Returns an appended list of ¡tt¿expression¡/tt¿ from each element of an aggregate, with an optional filter. arguments (aggregate GDL aggregate object. (e.g. from a ¡tt¿:sequence (:size ..)¡/tt¿ ¡tt¿:object¡/tt¿ specification).) &optional (expression Expression using ¡tt¿the-element¡/tt¿. Similar to a ¡tt¿the-object¡/tt¿ reference, which should return a list.))

- **check-computed-slots**

  (type Void intro computed-slots grammar: ¡form¿ = :computed-slots (¡token¿*) ¡token¿ = ¡string¿ — (¡string¿* ¡symbol¿ ¡expression¿+ ¡behavior¿+) ¡behavior¿ = :settable — :uncached Also check for special case in which only strings without a symbol following.)

- **check-documentation**

  (type nil intro plist containing keys :description :author :examples :date :version)

- **check-floating-string**

  (type nil intro check for special case in which documentation isn't followed by symbol spec)

- **check-form**

  (type nil intro general function that, given a predicate, validates all tokens in a slot declaration form)

- **check-functions**

  (type Void intro Checks :functions or :methods grammar according to following BNF: ¡form¿ = :functions — :methods (¡token¿*) ¡token¿ = (¡string¿* ¡symbol¿ ¡behavior¿+ ¡list¿ ¡body¿) ¡behavior¿ = :cached — :cached-eql — :cached-= — :cached-eq — :cached-equal — :cached-equalp)

- **check-input-slots**

  (type Void intro input-slots grammar: ¡form¿ = :input-slots (¡token¿*) ¡token¿ = ¡string¿ — ¡symbol¿ — (¡string¿* ¡symbol¿ ¡expression¿+ ¡behavior¿*) ¡behavior¿ = :settable — :defaulting Also check for special case in which only strings without a symbol following.)

- **check-objects**

  (type Void intro hidden-objects and :objects grammar: ¡form¿ = (hidden-):objects (¡token¿*) ¡token¿ = (¡string¿* ¡symbol¿ ¡rule¿*) ¡rule¿ = ¡keyword¿ ¡expression¿)

- **check-query-slots**

  (type nil intro unknown what is is)

- **check-trickle-down-slots**

  (type nil intro functions — :methods grammar: ¡form¿ = :trickle-down-slots (¡symbol¿*))

- **cl-lite**

  (type nil intro Traverses pathname in an alphabetical depth-first order, 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 (glisp:temporary-directory). [Note: this new documentation still needs proper formatting] If the :create-asd-file? keyword argument is specified as non-nil, a .asd file suitable for use with ASDF will be emitted into the directory indicated by the pathname argument. Note that ASDF (Another System Definition Utility), possibly with help of Quicklisp, is (as of 2013-03-12) the recommended way for handling Common Lisp system modules. As of version 2.31.9, ASDF is also capable of generating fasl "bundle" files as with the :create-fasl? argument to cl-lite. For the :author, :version, and :license arguments in the generated .asd file, the files author.isc, version.isc, and license.isc, respectively, are consulted, if they exist. They are searched for first in the codebase toplevel directory (the pathname argument to this function), then in the (user-homedir-pathname). The version defaults to the current ISO-8601 date without dashes, e.g. ”20130312”. Please see the Genworks Documentation for an overview of Quicklisp and ASDF, and see the Quicklisp and ASDF project documentation for detailed information. The source code for Quicklisp and ASDF should also be included with your Gendl distribution, and these are typically loaded by default into the development environment. For additional inputs to the cl-lite function, please see codebase-directory-node object for additional inputs (which can be given as keyword args to this function).)

- **cyclic-nth**

  (type Lisp object intro Returns nth from the list, or wraps around if nth is greater than the length of the list.)

- **defaulting [Macro]**

  (type Lisp object intro Returns a default value if the reference-chain is not handled. arguments (form Reference-chain with the or the-object default Lisp expression. Default value to return if reference-chain cannot be handled.))

- **define-format [Macro]**

  (type Standard-class [Macro] intro Defines a standard GDL output format for use with GDL views. arguments (name Symbol. mixin-list List of symbols.) &key (documentation Plist containing keys and strings for author, description, etc. slots List of lists or symbols. If a list, the list should contain a symbol, a default value, and optionally a documentation string. If a symbol, this is the name of the slot and there will be no default value. functions List of format-function definitions. Each definition is made up of a symbol, an argument-list, and a body.))

- **define-lens [Macro]**

  (type Void [Macro] intro Defines output-functions for the combination of the given output-format and GDL object. arguments (format-and-object List of two symbols. The first should name an output-format previously defined with ¡tt¿define-format¡/tt¿, and the second should name a GDL object previously defined with ¡tt¿define-object¡/tt¿. mixin-list NIL. This is not supported and should be left as NIL or an empty list for now.) &key ((skin t) Name of a skin defined with define-skin. This allows a class hierarchy of look and feel for each view combination. Defaults to T, a generic skin.))

- **define-object [Macro]**

  (type Defines a standard GDL object intro Please see the document USAGE.TXT for an overview of ¡tt¿define-object¡/tt¿ syntax.)

- **define-object-amendment [Macro]**

  (type Supplements or alters an existing GDL object definition intro Syntax is similar to that for ¡tt¿define-object¡/tt¿. Note that there is currently no way to undefine messages defined with this macro, other than redefining the original object or restarting the GDL session. Support for surgically removing messages will be added in a future GenDL release.)

- **div**

  (type Floating-point number intro Divides using rational division and converts the result (which may be a pure rational number) to a floating-point number. arguments (numerator Number. denominator Number.) &optional (more-denominators (&rest). More numbers to divide by.))

- **ensure-list**

  (type List intro If argument is not list, returns it in a list. If argument is a list, returns it unchanged. arguments (possible-list Lisp object))

- **find-dependants**

  (type List of pairs of instance/keyword intro Synonymous with find-messages-used-by.)

- **find-dependencies**

  (type List of pairs of instance/keyword intro Synonymous with find-messages-which-use.)

- **find-messages-used-by**

  (type List of pairs of instance/keyword intro This returns the list of direct dependants of a given message in a given instance. Note that this is not recursive; if you want to generate a tree, then you have to call this recursively yourself. If you want an easy way to remember the meaning of dependant and dependency: You have a dependency on caffeine. Your children are your dependants.)

- **find-messages-which-use**

  (type List of pairs of instance/keyword intro This returns the list of direct dependencies of a given message in a given instance. Note that this is not recursive; if you want to generate a tree, then you have to call this recursively yourself. If you want an easy way to remember the meaning of dependant and dependency: You have a dependency on caffeine. Your children are your dependants.)

- **flatten**

  (type List intro Returns a new list consisting of only the leaf-level atoms from ¡b¿list¡/b¿. Since nil is technically a list, ¡tt¿flatten¡/tt¿ also has the effect of removing nils from ¡b¿list¡/b¿, but may be inefficient if used only for this purpose. For removing nil values from a list, consider using ¡tt¿remove nil ...¡/tt¿ instead. note from Stack Overflow forum: http://stackoverflow.com/questions/258 a-list-using-common-lisp note Creative Commons license ¡pre¿ (defun flatten (lst &aux (result '())) (labels ((rflatten (lst1) (dolist (el lst1 result) (if (listp el) (rflatten el) (push el result))))) (nreverse (rflatten lst)))) ¡/pre¿ note This will not work with dotted lists, only with actual lists. If you need dotted lists, use the old definition of flatten, from Paul Graham On Lisp: ¡pre¿ (defun flatten (tree) (if (atom tree) (ensure-list tree) (nconc (flatten (car tree)) (if (cdr tree) (flatten (cdr tree)))))) ¡/pre¿ arguments (list List) see-also ¡tt¿remove¡/tt¿)

- **format-slot** [Macro]

  (type Lisp object [Macro] intro Returns the value of the given slot within the context of the current ¡tt¿with-format¡/tt¿ output format object. arguments (slot-name Symbol.))

- **fround-to-nearest**

  (type Number intro Rounds ¡b¿number¡/b¿ to the nearest ¡b¿interval¡/b¿, using type contagion rules for floating-point similar to the CL "fround" function. arguments (number Number interval Number))

- **half**

  (type Number intro Returns the result of dividing ¡b¿num¡/b¿ by the integer ¡tt¿2¡/tt¿. The type of the returned number will depend on the type of ¡b¿num¡/b¿. arguments (num Number))

- **ignore-errors-with-backtrace [Macro]**

  (type Like IGNORE-ERRORS, but in case of failure, return backtrace string as third value intro .)

- **index-filter**

  (type List intro Returns all elements of ¡b¿list¡/b¿ for whose index (starting at zero) the function ¡b¿fn¡/b¿ returns non-NIL. arguments (fn Function object (e.g. a lambda expression) list List))

- **iso-8601-date**

  (type String intro Returns the ISO8601 formatted date and possibly time from a Common Lisp universal time integer, e.g. 2007-11-30 or 2007-11-30T13:45:10)

- **lastcar**

  (type Lisp Object intro Returns the last element of ¡b¿list¡/b¿. arguments (list List))

- **least**

  (type List intro Returns the member of ¡b¿list¡/b¿ which returns the minimum numerical value when ¡b¿function¡/b¿ is applied to it. As second value is returned which is the actual minimum value (the return-value of ¡b¿function¡/b¿ as applied). This function comes from the Paul Graham book ¡u¿ANSI Common Lisp¡/u¿. arguments (function Function list List))

- **list-elements [Macro]**

  (type List of GDL Objects [Macro] intro Returns a listing of the elements of an aggregate, with an optional ¡tt¿the-element¡/tt¿ expression and filter. If an expression is given, the list of results from the expressions is returned. If no expression is given, a list of the objects themselves is returned. arguments (aggregate GDL aggregate object. (e.g. from a ¡tt¿:sequence (:size ..)¡/tt¿ ¡tt¿:object¡/tt¿ specification).) &optional (expression Expression using ¡tt¿the-element¡/tt¿. Similar to a ¡tt¿the-object¡/tt¿ reference. filter Function of one argument. Can be used to filter the result list.))

- **list-of-n-numbers**

  (type Returns a list of n numbers equally spaced between bounds num1 and num2, inclusive intro .)

- **list-of-numbers**

  (type List of Numbers intro Returns a list of incrementing numbers starting from ¡b¿num1¡/b¿ and ending with ¡b¿num2¡/b¿, inclusive. arguments (num1 Number num2 Number) &optional ((increment 1) Number. The distance between the returned listed numbers.))

- **load-glime**

  (type Void intro If the Glime (Slime Gendl auto-completion extensions) file exists, load it. Path is currently hardcoded to ¡tt¿(merge-pathnames ”emacs/glime.lisp” glime:*gdl-home*)¡/tt¿ or ¡tt¿ /genworks/gendl/emacs/glime.lisp¡/tt¿.)

- **load-quicklisp**

  (type Void intro This is intended for pre-built Gendl or GDL images. If the preconfigured quicklisp load file exists, load it. You can customize quicklisp location by setting global *quicklisp-home* or passing :path keyword argument to this function. &key ((path *quicklisp-home*) Pathname or string. Quicklisp location.))

- **make-keyword**

  (type Keyword symbol intro Converts given strings to a keyword. If any of the given arguments is not a string, it will be converted to one with (format nil ” a” string). arguments (strings &rest Strings))

- **make-object**

  (type GDL Object intro Instantiates an object with specified initial values for input-slots. arguments (object-name Symbol. Should name a GDL object type. arguments spliced-in plist. A plist of keyword symbols and values for initial ¡tt¿input-slots¡/tt¿.))

- **mapsend**

  (type List intro Returns a new list which is the result of sending ¡b¿message¡/b¿ to each GDL object in ¡b¿object-list¡/b¿. arguments (object-list List of GDL objects. message Keyword symbol.))

- **maptree**

  (type List intro Returns the results of applying ¡b¿fn¡/b¿ to each GDL object in the object tree rooted at ¡b¿node¡/b¿ in a “depth-first” tree traversal. arguments (node GDL object fn Function. Operates on a single argument which is a GDL object) &optional ((accept? #'always) Function. Determines which nodes to accept in the final result (prune? #'never) Function. Determines which nodes to prune from the tree traversal (get-children children) Keyword symbol :children or Function. Function applied to a given node to get its children. The default, keyword symbol :children, uses the node's normal children as returned by (the-object node children).))

- **max-of-elements [Macro]**

  (type Number [Macro] intro Returns the maximum of ¡tt¿expression¡/tt¿ from each element of an aggregate, with an optional filter. arguments (aggregate GDL aggregate object. (e.g. from a ¡tt¿:sequence (:size ..)¡/tt¿ ¡tt¿:object¡/tt¿ specification).) &optional (expression Expression using ¡tt¿the-element¡/tt¿. Similar to a ¡tt¿the-object¡/tt¿ reference, which should return a number.))

- **min-of-elements [Macro]**

  (type Number [Macro] intro Returns the minimum of ¡tt¿expression¡/tt¿ from each element of an aggregate, with an optional filter. arguments (aggregate GDL aggregate object. (e.g. from a ¡tt¿:sequence (:size ..)¡/tt¿ ¡tt¿:object¡/tt¿ specification).) &optional (expression Expression using ¡tt¿the-element¡/tt¿. Similar to a ¡tt¿the-object¡/tt¿ reference, which should return a number.))

- **most**

  (type List intro Returns the member of ¡b¿list¡/b¿ which returns the maximum numerical value when ¡b¿function¡/b¿ is applied to it. As second value is returned which is the actual maximum value (the return-value of ¡b¿function¡/b¿ as applied). This function comes from the Paul Graham book ¡u¿ANSI Common Lisp¡/u¿. arguments (function Function list List))

- **near-to?**

  (type Boolean intro Predicate to test if number is within tolerance of ¡b¿near-to¡/b¿. The default tolerance is the value of the parameter ¡tt¿*zero-epsilon*¡/tt¿. arguments (number Number near-to Number) &optional ((tolerance *zero-epsilon*) Number))

- **near-zero?**

  (type Boolean intro Returns non-NIL iff ¡b¿number¡/b¿ is greater than ¡b¿tolerance¡/b¿ different from zero. arguments (number Number) &optional ((tolerance *zero-epsilon*) Number) see-also ¡tt¿zerop¡/tt¿ (Common Lisp function))

- **never**

  (type NIL intro Always returns the value ¡tt¿NIL¡/tt¿ regardless of ¡b¿arg¡/b¿. arguments (arg Lisp object. Ignored))

- **number-format**

  (type String intro Returns a string displaying ¡b¿number¡/b¿ rounded to ¡b¿decimal-places¡/b¿ decimal places. arguments (number Number decimal-places Integer))

- **number-round**

  (type Number intro Returns ¡b¿number¡/b¿ rounded to ¡b¿decimal-places¡/b¿ decimal places. arguments (number Number decimal-places Integer))

- **plist-keys**

  (type List of keyword symbols intro Returns the keys from a plist. arguments (plist Plist))

- **plist-values**

  (type List of Lisp objects intro Returns the values from a plist. arguments (plist Plist))

- **print-messages [Macro]**

  (type [Macro] Void intro Prints the specified GDL object messages (i.e. slots) and their current values to standard output. arguments (vars unquoted symbols (&rest argument)))

- **print-variables [Macro]**

  (type [Macro] Void intro Prints the specified variables and current values to standard output. arguments (vars unquoted symbols (&rest argument)))

- **query-collect**

  (type List of row objects intro Returns all the rows from a query, or if a key is given, returns the values from the column named with the key. &optional (key Keyword symbol. If given, the values from this named column will be returned, rather than the entire row objects.))

- **read-safe-string**

  (type Lisp object intro Reads an item from string, protecting against lisp evaluation with the '#.' reader macro. Throws an error if evaluation would have occured. arguments (string string))

- **read-snapshot**

  (type GDL Instance intro Reads the snapshot data from stream, from the string, or from file indicated by filename. If no optional keyword ¡tt¿object¡/tt¿ argument is given, a new GDL instance based on the data in the snapshot file is returned. If an ¡tt¿object¡/tt¿ is given, the object should be compatible in type to that specified in the snapshot file, and this existing object will be modified to contain the set slot values and toplevel inputs as specified in the snapshot file. &key ((filename /tmp/snap.gdl) String or pathname. File to be read. If either string or stream is specified, this will not be used. (string nil) String of data. The actual snapshot contents, stored in a string. If stream is specified, this will not be used. (stream nil) Stream open for input. A stream from which the snapshot data can be read. (keep-bashed-values? nil) Boolean. Indicates whether to keep the currently bashed values in object before reading snap values into it. (object nil) GDL object. Existing object to be modified with restored values.))

- **remove-plist-entry**

  (type Plist intro Returns a new plist sans any key/value pairs where the plist key is eql to the given key. Optionally a different test than #'eql can be specified with the :test keyword argument. arguments (plist Plist. The source plist. key matching key, typically a keyword symbol. The key to target for removal.) &key ((test #'eql) predicate equality function taking two arguments. The function to use for matching.) examples (remove-plist-entry (list :a "a" :b :a) :a))

- **replace-substring**

  (type String intro Replaces all substring occurrences of ¡b¿old¡/b¿ with ¡b¿new¡/b¿ in ¡b¿string¡/b¿. Note: In a full GDL system, you will have glisp:replace-regexp, which is more powerful and probably more efficient than this. arguments (string String. The source string. old String. The substring to be replaced. new String. The substring to replace it with.) see-also ¡tt¿excl:replace-regexp¡/tt¿)

- **round-to-nearest**

  (type Number intro Rounds ¡b¿number¡/b¿ to the nearest ¡b¿interval¡/b¿. arguments (number Number interval Number))

- **safe-float**

  (type Double-float Number intro Coerces ¡b¿number¡/b¿ to a double-precision floating-point number if possible. If this is not possible, returns ¡tt¿0.0d0¡/tt¿ (i.e. zero in the form of a double-precision floating-point number). arguments (number Number))

- **safe-sort**

  (type List intro Nondestructive analog of the Common Lisp ¡tt¿sort¡/tt¿ function. Returns a freshly created list. arguments (list List. The list to be sorted.) &rest (args Argument list. Identical to the arguments for Common Lisp ¡tt¿sort¡/tt¿.))

- **set-format-slot [Macro]**

  (type Void [Macro] intro Sets the value of the given slot within the context of the current ¡tt¿with-format¡/tt¿ output format object. arguments (slot-name Symbol. value Lisp Value))

- **split**

  (type List of Strings intro Returns a list containing the elements of ¡b¿string¡/b¿ after having been split according to ¡b¿split-chars¡/b¿ as delimiting characters. arguments (string String) &optional ((split-chars (list

  )) List of characters) see-also ¡tt¿glisp:split-regexp¡/tt¿)

- **status-message**

  (type NIL intro Prints ¡b¿string¡/b¿, followed by a newline, to ¡tt¿*trace-output*¡/tt¿, which is generally the system console. arguments (string String))

- **string-append**

  (type String intro Returns a new string made up of concatenating the arguments. arguments (&rest (args strings)))

- **sum-elements [Macro]**

  (type Number [Macro] intro Returns the sum of ¡tt¿expression¡/tt¿ from each element of an aggregate, with an optional filter. arguments (aggregate GDL aggregate object. (e.g. from a ¡tt¿:sequence (:size ..)¡/tt¿ ¡tt¿:object¡/tt¿ specification).) &optional (expression Expression using ¡tt¿the-element¡/tt¿. Similar to a ¡tt¿the-object¡/tt¿ reference, which should return a number.))

- **the [Macro]**

  (type Lisp object intro Sends the ¡tt¿reference-chain¡/tt¿ to ¡tt¿self¡/tt¿, which typically means it is used within the context of a define-object where self is automatically lexically bound. arguments (reference-chain (&rest). A spliced-in list of symbols naming messages, which can be slots or objects starting from ¡tt¿self¡/tt¿. For referring to elements of a quantified set, or for passing arguments to GDL functions which take arguments, use parentheses around the message name and enclose the quantified element index or function arguments after the message name.) example This example sends the ¡tt¿length¡/tt¿ message to the "zeroth" element of the quantified set of arms contained in the body which is contained in the robot which is contained in self: ¡pre¿ (the robot body (arms 0) length) ¡/pre¿)

- **the-child [Macro]**

  (type similar to "the," but used to refer to the child part from within an :objects or :hidden-objects specification intro . This is often used for sending the ¡tt¿index¡/tt¿ message to an element of a quantified set. arguments (reference-chain (&rest). A spliced-in list of symbols naming messages relative to the child object.))

- **the-element [Macro]**

  (type Lisp Object [Macro] intro Acts similarly to ¡tt¿the-object¡/tt¿ for each element of an aggregate, within the context of a ¡tt¿list-elements¡/tt¿, ¡tt¿append-elements¡/tt¿, ¡tt¿max-of-elements¡/tt¿, ¡tt¿min-of-elements¡/tt¿, ¡tt¿sum-elements¡/tt¿, or a query operator (query operators are not yet documented). arguments (args (&rest). Standard reference chain applicable to the element.))

- **the-object [Macro]**

  (type Lisp object intro Sends the ¡tt¿reference-chain¡/tt¿ to ¡tt¿object¡/tt¿, which must be specified as a Lisp expression (e.g. a variable) which evaluates to a GDL object. arguments (reference-chain (&rest). A spliced-in list of symbols naming messages, which can be slots or objects starting from ¡tt¿object¡/tt¿. For referring to elements of a quantified set, or for passing arguments to GDL functions which take arguments, use parentheses around the message name and enclose the quantified element index or function arguments after the message name.) example This example sends the ¡tt¿length¡/tt¿ message to the "zeroth" element of the quantified set of arms contained in the body which is contained in the robot which is contained in ¡tt¿object¡/tt¿: ¡pre¿ (the-object object robot body (arms 0) length) ¡/pre¿)

- **twice**

  (type Number intro Returns the result of multiplying ¡b¿num¡/b¿ by the integer ¡tt¿2¡/tt¿. The type of the returned number will depend on the type of ¡b¿num¡/b¿. arguments (num Number))

- **undefine-object**

  (type NIL intro Clears all definitions associated with ¡b¿object-name¡/b¿ from the currently running GDL session. arguments (object-name Non-keyword Symbol naming a GDL object type))

- **universal-time-from-iso-8601**

  (type Integer representing Common Lisp Universal Time intro Returns the universal time from a date formatted as an iso-8601 date, optionally with time, e.g. 2012-07-08 or 2012-07-08T13:33 or 2012-07-08T13:33:00)

- **with-error-handling [Macro]**

(type [Macro] intro Wraps the ¡b¿body¡/b¿ of code with error-trapping and system timeout. A warning is given if an error condition occurs with ¡b¿body¡/b¿. &key ((timeout 2) Timeout in Seconds. timeout-body Body of code to evaluate if timeout occurs. Default is to print a warning and return nil.) &rest (body Body of code to be wrapped))

- **with-format [Macro]**

  (type Void [Macro] intro Used to establish an output format and a stream to which data is to be sent. This supports a full range of output options such as page dimensions, view transforms, view scales, etc. example ¡pre¿ (gdl::with-format (pdf ”/tmp/box.pdf” :view-transform (getf *standard-views* :trimetric)) (write-the-object (make-instance 'box :length 100 :width 100 :height 100) cad-output)) ¡/pre¿)

- **with-format-slots [Macro]**

  (type Void [Macro] intro Wrap this around a body of code which should have access to multiple slots from the context of the current ¡tt¿with-format¡/tt¿ output format object. arguments (slots List of Symbols.))

- **write-env [Macro]**

  (type Void [Macro] (usually used just for outputting) intro Within the context of a ¡tt¿with-format¡/tt¿, calls functions of the format object, optionally with arguments. Typically these functions will output data to the ¡tt¿stream¡/tt¿ established by the ¡tt¿with-format¡/tt¿. arguments (function-calls (&rest). Functions on the named output-format to be called.) example ¡pre¿ (with-format (base-format my-object) (write-env (:a ”Hello World, my object's length is: ”) (:a (the length)))) ¡/pre¿)

- **write-plist**

  (type Pretty-prints a plist to a file with standard I/O syntax intro . &key ((plist nil) List. The list to be printed to the file (output-path nil) Pathname of a file. The file to be created or superseded.))

- **write-the [Macro]**

  (type Lisp object [Macro] intro Typcially used only to send output, not for the return value. This macro is used within the body of a ¡tt¿with-format¡/tt¿. It sends the ¡tt¿reference-chain¡/tt¿ to ¡tt¿self¡/tt¿, which typically means it is used within the context of a define-object where self is automatically lexically bound. The reference-chain must terminate with an output-function defined for the combination of the output-format specified in the enclosing ¡tt¿with-format¡/tt¿, and the object identified by ¡tt¿self¡/tt¿. arguments (reference-chain (&rest). A spliced-in list of symbols naming messages, which can be slots or objects starting from ¡tt¿self¡/tt¿, terminating with the name of an output-function. For referring to elements of a quantified set, or for passing arguments to GDL functions which take arguments, use parentheses around the message name and enclose the quantified element index or function arguments after the message name.))

- **write-the-object [Macro]**

(type Lisp object [Macro] intro Typcially used only to send output, not for the return value. This macro is used within the body of a ¡tt¿with-format¡/tt¿. It sends the ¡tt¿reference-chain¡/tt¿ to ¡tt¿object¡/tt¿, which must be specified as a Lisp expression (e.g. a variable) which evaluates to a GDL object. The reference-chain must terminate with an output-function defined for the combination of the output-format specified in the enclosing ¡tt¿with-format¡/tt¿, and the object identified by ¡tt¿object¡/tt¿. arguments (reference-chain (&rest). A spliced-in list of symbols naming messages, which can be slots or objects starting from ¡tt¿object¡/tt¿, terminating with the name of an output-function. For referring to elements of a quantified set, or for passing arguments to GDL functions which take arguments, use parentheses around the message name and enclose the quantified element index or function arguments after the message name.))

- **hat-2**

  (type Number intro Return ¡b¿number¡/b¿ raised to the power two (2). arguments (number Number))

9.8.3 Variables and Constants

- ***allow-nil-list-of-numbers?***
- ***bias-to-double-float?***
- ***color-plist***
- ***color-table***
- ***color-table-decimal***
- ***colors-default***
- ***compile-circular-reference-detection?***
- ***compile-dependency-tracking?***
- ***compile-documentation-database?***
- ***compile-for-dgdl?***
- ***compile-source-code-database?***
- ***curve-chords***
- ***ensure-lists-when-bashing?***
- ***load-documentation-database?***
- ***load-source-code-database?***
- ***on-syntax-error***
- ***out-of-bounds-sequence-reference-action***

- ***remember-previous-slot-values?***
- ***root-checking-enabled?***
- ***run-with-circular-reference-detection?***
- ***run-with-dependency-tracking?***
- ***sort-children?***
- ***undeclared-parameters-enabled?***
- ***with-format-direction**
- ***with-format-element-type***
- ***with-format-external-format***
- ***with-format-if-does-not-exist***
- ***with-format-if-exists***
- ***zero-epsilon***
- **+phi+**
- **2pi**
- **pi/2**

9.9 gdl-user

9.10 gendl-doc

9.11 genworks-gdl (Genworks GDL)

9.12 geom-base (Wireframe Geometry)

9.12.1 Object Definitions

- **angular-dimension**

  Mixins: linear-dimension, vanilla-mixin

  Description This dimensional object produces a clear and concise arc dimensional annotation.

  Input slots (required):

  Arc-object *GDL object*
  The arc being measured.

```
(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))
```

Figure 9.1: Example Code for angular-dimension

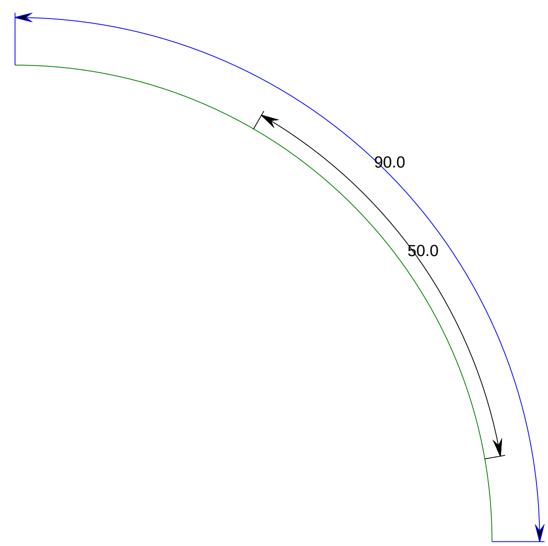

Figure 9.2: angular-dimension example

Input slots (optional):

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.

```
(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))
```

Figure 9.3: Example Code for arc

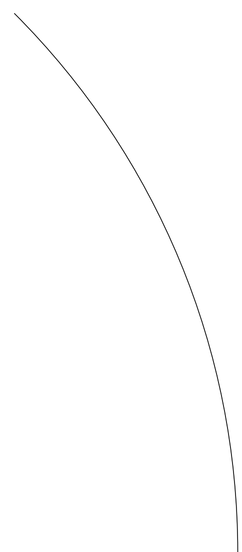

Figure 9.4: arc example

Computed slots:

Dim-value *Number*
2D distance relative to the base-plane-normal. Can be over-ridden in the subclass

- **arc**

Mixins: arcoid-mixin, base-object

Description A segment of a circle. The start point is at the 3 o'clock position, and positive angles are measured anti-clockwise.

Input slots (required):

Radius *Number*
Distance from center to any point on the arc.

Input slots (optional):

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.

Gdl 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.

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).

- **arcoid-mixin**

Mixins: vanilla-mixin

Description This object is a low level object used to define an arc like object. It is not recommended to be used directly by GDL common users. For developers it should be used as a mixin.

Input slots (required):

Radius *Number*
Distance from center to any point on the arc.

Input slots (optional):

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.

- **base-coordinate-system**

Mixins: base-object, vanilla-mixin

Description This provides a default 3D Cartesian coordinate system. It mixes in base-object and does not extend it in any way, so as with base-object, it provides an imaginary geometric reference box with a length, width, height, center, and orientation.

- **base-drawing**

Mixins: base-object

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.

Input slots (optional):

Height *Number*

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.

- **base-object**

Mixins: vanilla-mixin

```
(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)))

(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))
```

Figure 9.5: Example Code for base-drawing

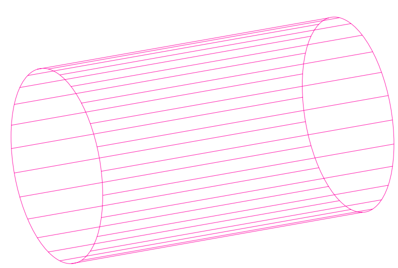

Figure 9.6: base-drawing example

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.

Input slots (optional):

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`.

Onclick-function *Lambda function of zero arguments, or nil*

If non-nil, this function gets invoked when the user clicks the object in graphics front-ends which support this functionality, e.g. SVG/Raphael and X3DOM.

Input slots (optional, defaulting):

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 wire-frame 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.

```
 (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))
 #tower @ #x750666f2
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):
|#
;;
;;
;;
```

Figure 9.7: Example Code for base-object

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 ıorthonormal 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-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.

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.

Gdl 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.

- **base-view**

Mixins: base-object

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.

Input slots (optional):

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.

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

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

  :objects
  ((drawing :type 'two-viewed-drawing
            :objects-to-draw (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-to-draw)

  :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-to-draw))

   (top-view :type 'base-view
             :projection-vector (getf *standard-views* :top)
             :length (* 0.30 (the length))
             :objects (the objects-to-draw))))

  (generate-sample-drawing :objects
   (the-object (make-object 'box-with-two-viewed-drawing) drawing top-view))
```

Figure 9.8: Example Code for base-view

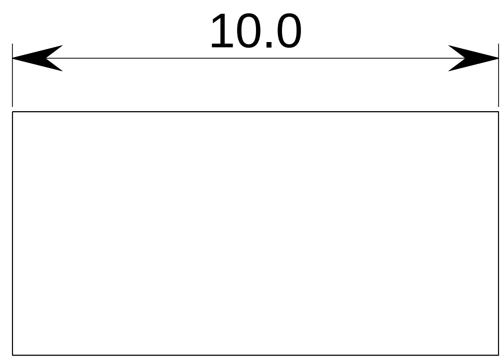

Figure 9.9: base-view example

Center *3D-point*

Center of the view box. Specify this or corner, not both.

Corner *3D-point*

Top left (i.e. rear left from top view) of the view box. Specify this or center, not both.

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.

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.

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].

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*`.

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.

Gdl 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.

- **bezier-curve**

Mixins: base-object

Description GDL currently supports third-degree Bezier curves, which are defined using four 3D ıcontrol-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.

Input slots (required):

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.

Gdl 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`.

```
(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))))
```

Figure 9.10: Example Code for bezier-curve

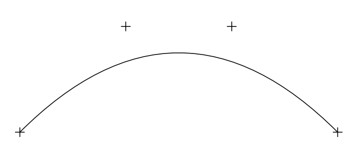

Figure 9.11: bezier-curve example

```
(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))
```

Figure 9.12: Example Code for box

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.

Point *3D Point*

Returns the point on this Bezier curve corresponding to the given `parameter`, which should be between 0 and 1.

- **box**

Mixins: base-object

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.

- **c-cylinder**

Mixins: cylinder

Description Provides a simple way to create a cylinder, by specifying a start point and an end point.

Input slots (required):

End *3D Point*

Center of the end cap.

Start *3D Point*

Center of the start cap.

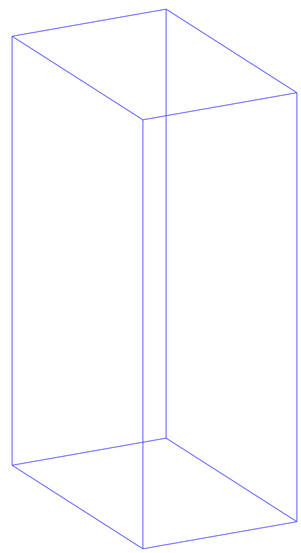

Figure 9.13: box example

```
(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))
```

Figure 9.14: Example Code for c-cylinder

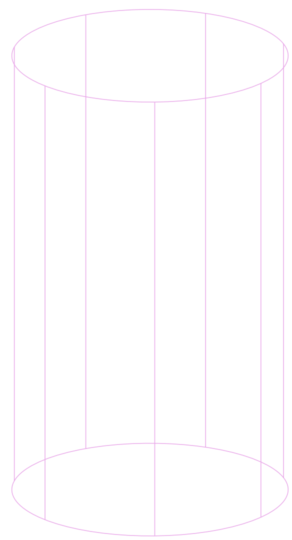

Figure 9.15: c-cylinder example

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.

- **center-line**

Mixins: outline-specialization-mixin, base-object

Description Creates a dashed single centerline or crosshair centerline on a circle.

Input slots (required):

Size *Number*
The length of the centerline.

Input slots (optional):

Circle? *Boolean*
Determines whether this will be a circle crosshair. Defaults to nil.

```
(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))
```

Figure 9.16: Example Code for center-line

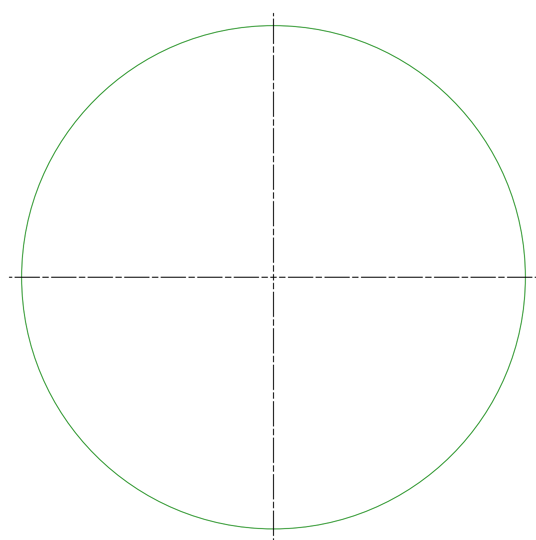

Figure 9.17: center-line example

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

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

(generate-sample-drawing :objects (make-object 'circle-sample))
```

Figure 9.18: Example Code for circle

Input slots (optional, defaulting):

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.

- **circle**

Mixins: arc

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.

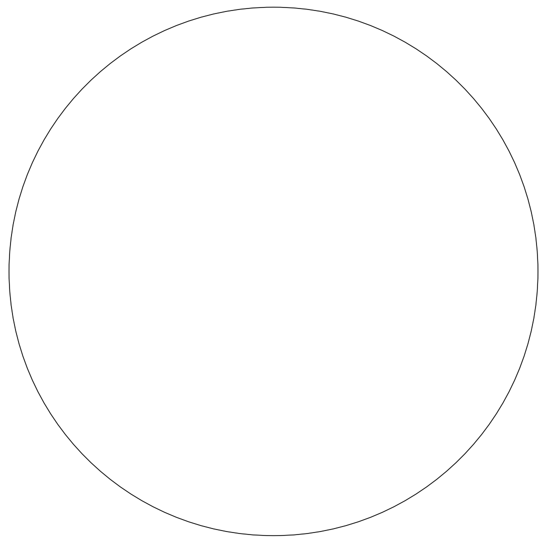

Figure 9.19: circle example

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.

- **cone**

Mixins: cylinder

Description A pyramid with a circular cross section, with its vertex above the center of its base. Partial cones and hollow cones are supported.

Input slots (optional):

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:

```
 (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)
```

Figure 9.20: Example Code for cone

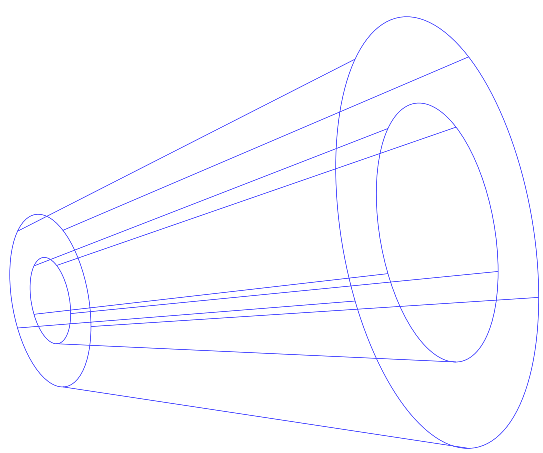

Figure 9.21: cone example

Height *Number*

Z-axis dimension of the reference box. Defaults to zero.

Width *Number*

X-axis dimension of the reference box. Defaults to zero.

- **constrained-arc**

Mixins: arc

Description This object is intended to simplify the process of constructing lines using various constraints. Currently supported are 2 through-points or 1 through-point and at-angle. Note the line-constraints must be an evaluatable s-expression as this is not processed as a macro

Computed slots:

Center *3D Point*

Indicates in global coordinates where the center of the reference box of this object should be located.

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 ıorthonormal matrix, meaning each row is a vector with a magnitude of one (1.0).

Radius *Number*

Distance from center to any point on the arc.

- **constrained-fillet**

Mixins: constrained-arc, vanilla-mixin

Description This object is the same as constrained-arc, but it is only meaningful for arc-constraints which contain two :tangent-to clauses, and it automatically trims the result to each point of tangency

Computed 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.

- **constrained-line**

Mixins: line

```
(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))
```

Figure 9.22: Example Code for cylinder

Description This object is intended to simplify the process of constructing lines using various constraints. Currently supported are 2 through-points or 1 through-point and at-angle. Note the line-constraints must be an evaluatable s-expression as this is not processed as a macro

Computed 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.

Gdl functions:

Tangent-point Icad Compat function

- **cylinder**

Mixins: ifs-output-mixin, arcoid-mixin, base-object

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.

Input slots (required):

Length *Number*

Distance from center of start cap to center of end cap.

Radius *Number*

Radius of the circular cross section of the cylinder.

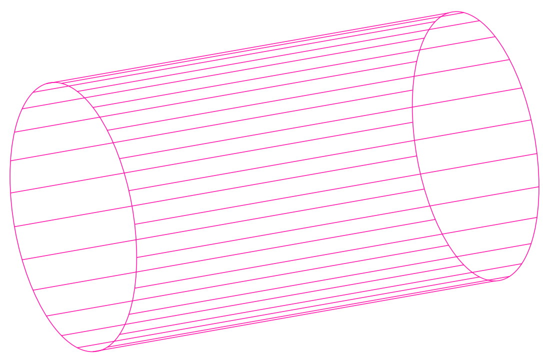

Figure 9.23: cylinder example

Input slots (optional):

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.

```
(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))
```

Figure 9.24: Example Code for ellipse

Hollow? *Boolean*

Indicates whether there is an inner-radius and thus the cylinder is hollow.

Start *3D Point*

The center of the start cap.

- **ellipse**

Mixins: arcoid-mixin, base-object

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.

Input slots (required):

Major-axis-length *Number*

Length of (generally) the longer ellipse axis

Minor-axis-length *Number*

Length of (generally) the shorter ellipse axis

Input slots (optional):

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.

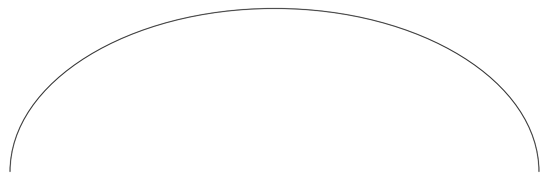

Figure 9.25: ellipse example

Length *Number*
: Y-axis dimension of the reference box. Defaults to zero.

Width *Number*
: X-axis dimension of the reference box. Defaults to zero.

- **general-note**

Mixins: outline-specialization-mixin, base-object

Description Creates a text note in the graphical view port and in a PDF DXF output file.

Input slots (optional):

Center *3D-point*
: Center of the text. Specify this or start, not both.

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

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

(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))
```

Figure 9.26: Example Code for general-note

CCD 2	Approved by	Jane Smith
ABC 2	Created by	David Brown

Figure 9.27: general-note example

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

- **global-filleted-polygon-projection**

Mixins: global-polygon-projection

Description Similar to a global-polygon-projection, but the polygon is filleted as with global-filleted-polygon.

Input slots (optional):

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.

- **global-filleted-polyline**

Mixins: global-filleted-polyline-mixin, vanilla-mixin

```
(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)
```

Figure 9.28: Example Code for global-filleted-polygon-projection

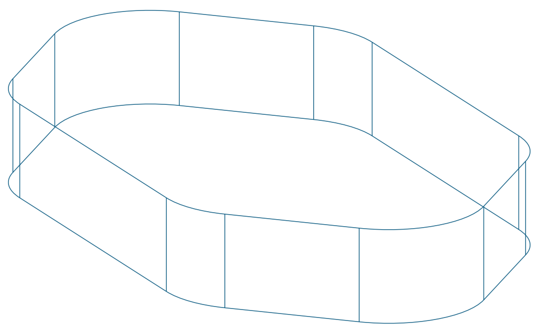

Figure 9.29: global-filleted-polygon-projection example

```
(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))
```

Figure 9.30: Example Code for global-filleted-polyline

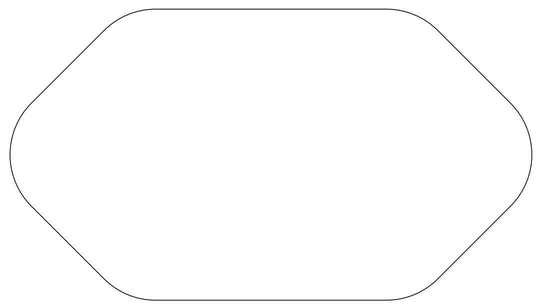

Figure 9.31: global-filleted-polyline example

<u>**Description**</u> 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.

- **global-filleted-polyline-mixin**

  <u>**Mixins:**</u> global-polyline-mixin

  <u>**Description**</u> Generates a polyline with the corners filleted according to default radius or the radius-list.

  <u>**Input slots (required):**</u>

  Vertex-list *List of 3D Points*

  The vertices ("corners") of the polyline.

  <u>**Input slots (optional):**</u>

  Closed? *Boolean*

  Controls whether the filleted-polyline should automatically be closed.

  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.

```
(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))
```

Figure 9.32: Example Code for global-polygon-projection

Computed slots:

Straights *List of pairs of 3D points*
Each pair represents the start and end of each straight segment of the filleted-polyline.

Hidden objects (sequence):

Fillets *Sequence of fillets*
Each fillet is essentially an arc representing the curved elbow of the filleted-polyline.

- **global-polygon-projection**

Mixins: base-object, ifs-output-mixin

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.

Input slots (required):

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.

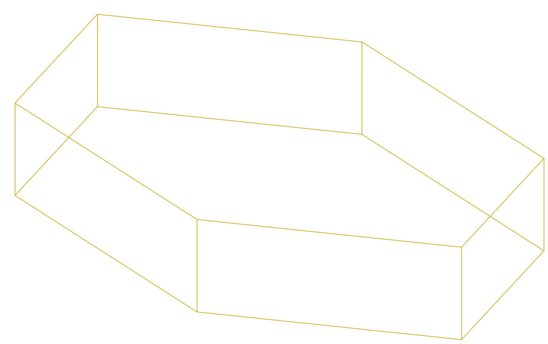

Figure 9.33: global-polygon-projection example

Input slots (optional):

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 ıand 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.

- **global-polyline**

Mixins: global-polyline-mixin, vanilla-mixin

```
(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))
```

Figure 9.34: Example Code for global-polyline

Description A sequence of points connected by straight line segments. Please see global-polyline-mixin for documentation on the messages.

- **global-polyline-mixin**

  Mixins: base-object

  Description Makes a connected polyline with vertices connected by straight line segments.

  Input slots (required):

  Vertex-list *List of 3D Points*
  The vertices ("corners") of the polyline.

  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.

  Lines *List of pairs of 3D points*
  Each pair represents the start and end of each line segment in the polyline.

- **horizontal-dimension**

  Mixins: linear-dimension, vanilla-mixin

  Description Creates a dimension annotation along the horizontal axis.

  Input slots (optional):

  Base-plane-normal Must be specified in the subclass except for angular

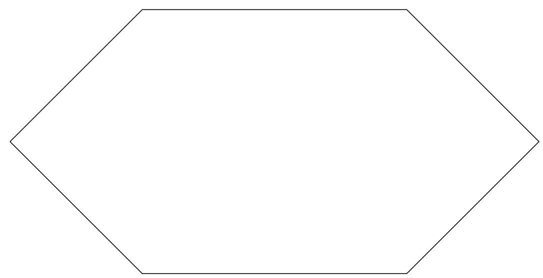

Figure 9.35: global-polyline example

```
(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))
```

Figure 9.36: Example Code for horizontal-dimension

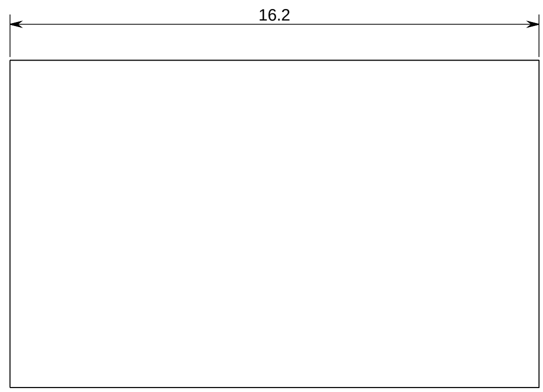

Figure 9.37: horizontal-dimension example

Dim-text-start *3D Point*

Determines where the text will start. Defaults to reasonable location for horizontal-dimension.

Computed slots:

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

- **label**

Mixins: outline-specialization-mixin, base-object

Description Produces a text label for graphical output

Input slots (required):

Leader-path *List of 3D Points*

List making up leader line, starting from where the arrowhead normally is.

Input slots (optional):

Arrowhead-length *Length (from tip to tail) of arrowhead glyph*

Defaults to twice the `arrowhead-width`

```
(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))
```

Figure 9.38: Example Code for label

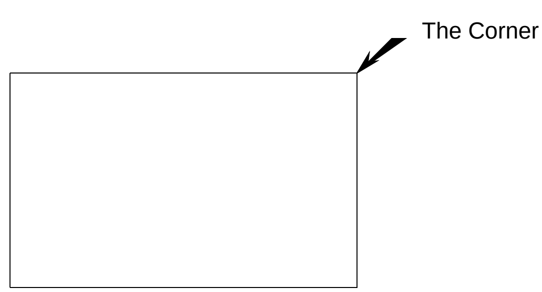

Figure 9.39: label example

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 `:left` or `:right`*

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.

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 ıorthonormal matrix, meaning each row is a vector with a magnitude of one (1.0).

- **leader-line**

Mixins: base-object

Description Creates a leader line with arrows on zero, one, or both ends

Input slots (required):

Path-points *List of 3D Points*

Leader-line is rendered as a polyline going through these points.

Input slots (optional):

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.

Computed slots:

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.

```
(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))
```

Figure 9.40: Example Code for line

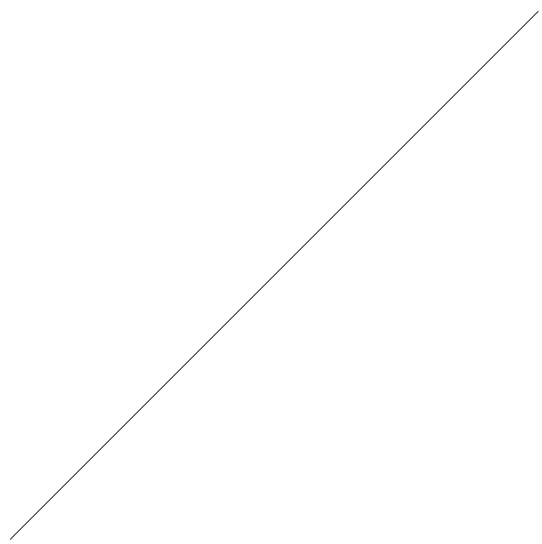

Figure 9.41: line example

: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.

- **line**

  Mixins: base-object

  Description Provides a simple way to create a line, by specifying a start point and an end point.

  Input slots (required):

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.

- **linear-dimension**

Mixins: outline-specialization-mixin, base-object

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.

Input slots (required):

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

Input slots (optional):

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 ıorthonormal 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

View-reference-object *GDL object or NIL*

View object which will use this dimension. Defaults to NIL.

```
(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))
```

Figure 9.42: Example Code for parallel-dimension

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

- **parallel-dimension**

  Mixins: linear-dimension

  Description Creates a dimension annotation along an axis from a start point to an end point.

  Input slots (optional):

Dim-text-start *3D Point*

Determines where the text will start. Defaults to reasonable location for horizontal-dimension.

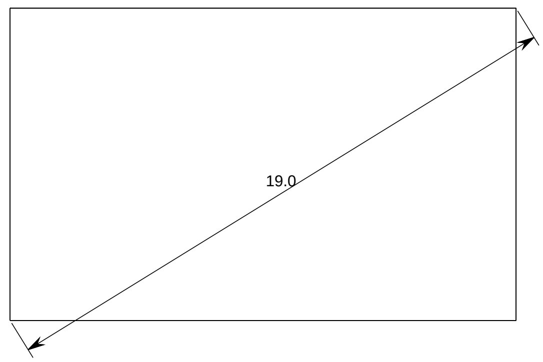

Figure 9.43: parallel-dimension example

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

- **pie-chart**

  Mixins: base-object

  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).

  Input slots (optional):

  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.

```
(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))
```

Figure 9.44: Example Code for pie-chart

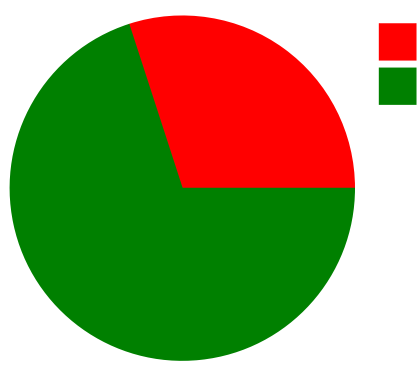

Figure 9.45: pie-chart example

```
(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))
```

Figure 9.46: Example Code for point

Line-color *Keyword symbol naming color from `*color-table*`*
. 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 `*color-table*`*
. 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.

- **point**

  Mixins: sphere

  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*).

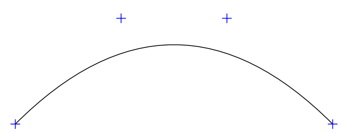

Figure 9.47: point example

Input slots (optional):

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.

- **points-display**

Mixins: outline-specialization-mixin

Description Product a list of hidden-children which are GDL point objects, which will be displayed in normal renderings using the `outline-specialization-mixin` mechanism.

Input slots (required):

Points *List of 3D points (i*
e. vectors). The points to be displayed.

- **renderer-mixin**

<u>**Mixins:**</u> vanilla-mixin

<u>**Description**</u> Object mixed into the base-view to compute required values to provide a rendered perspective view, as in VRML.

<u>**Input slots (required):**</u>

Object-roots *List of GDL Objects*
Roots of the leaf objects to be displayed in this renderer view.

Objects *List of GDL Objects*
Leaves of the objects to be displayed in this renderer view.

<u>**Input slots (optional):**</u>

3d-box *List of two 3D points*
The left-front-lower and right-rear-upper corners of the axis-aligned bounding box of the `object-roots` and `objects`.

3d-box-center *3D Point*
The effective view center for the scene contained in this view object. Defaults to the center of the bounding sphere of all the objects in the scene, consisting of the `object-roots` and the `objects`.

Bounding-sphere *Plist containing keys: `:center` and `:radius`*
This plist represents the tightest-fitting sphere around all the objects listed in the `object-roots` and the `objects`

Field-of-view-default *Number in angular degrees*
The maximum angle of the view frustrum for perspective views. Defaults to 0.1 (which results in a near parallel projection with virtually no perspective effect).

View-vectors *Plist*
Keys indicate view vector names (e.g. `:trimetric`), and values contain the 3D vectors. Defaults to the parameter `*standard-views*`, but with the key corresponding to current `(the view)` ordered first in the plist. This list of view-vectors is used to construct the default `viewpoints`.

Viewpoints *List of Plists*
Each plist contains, based on each entry in the `view-vectors`, keys:

- `:point` (camera location, defaults to the `3d-box-center` translated along the corresponding element of `view-vectors`) by the local camera distance. The camera distance is computed based on the field-of-view angle and the `bounding-sphere`
- `:orientation` (3d matrix indicating camera orientation)

```
(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))
```

Figure 9.48: Example Code for route-pipe

  - `field-of-view` Angle in degrees of the view frustrum (i.e. lens angle of the virtual camera).

- **route-pipe**

  Mixins: global-filleted-polyline-mixin, outline-specialization-mixin

  Description Defines an alternating set of cylinders and torus sections for the elbows

  Input slots (required):

  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)

  Input slots (optional):

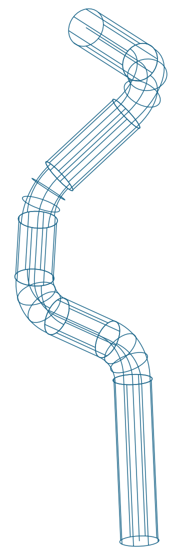

Figure 9.49: route-pipe example

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 ıorthonormal 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.

- **sample-drawing**

Mixins: base-drawing, vanilla-mixin

```
(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)
```

Figure 9.50: Example Code for sphere

Description Defines a simple drawing with a single view for displaying objects or object-roots.

Input slots (optional):

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.

- **sphere**

Mixins: ifs-output-mixin, arcoid-mixin, base-object

Description The set of points equidistant from a given center point.

Input slots (required):

Radius *Number*
Distance from center to any point on the sphere.

Input slots (optional):

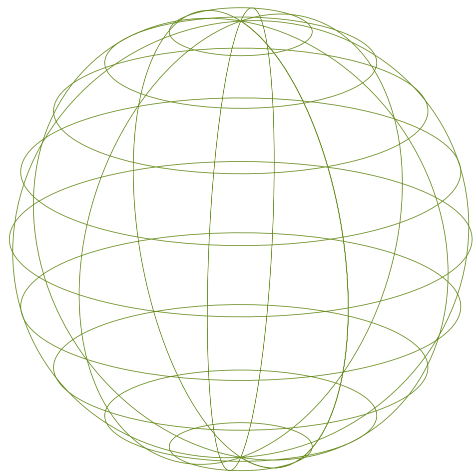

Figure 9.51: sphere example

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.

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.

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.

```
(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)
```

Figure 9.52: Example Code for spherical-cap

Width *Number*
X-axis dimension of the reference box. Defaults to zero.

- **spherical-cap**

Mixins: ifs-output-mixin, arcoid-mixin, base-object

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.

Input slots (required):

Axis-length *Number*
The distance from the center of the base to the center of the dome.

Base-radius *Number*
Radius of the base.

Input slots (optional):

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.

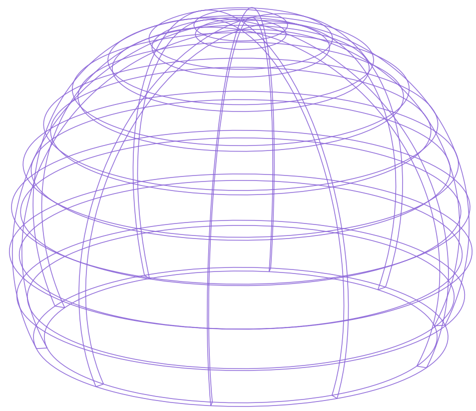

Figure 9.53: spherical-cap example

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.

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.

- **text-line**

  Mixins: base-object

  Description Outputs a single line of text for graphical display.

  Input slots (optional):

  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.

- **torus**

  Mixins: ifs-output-mixin, arcoid-mixin, base-object

  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.

  Input slots (required):

  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.

  Input slots (optional):

  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.

  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

```
(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)
```

Figure 9.54: Example Code for torus

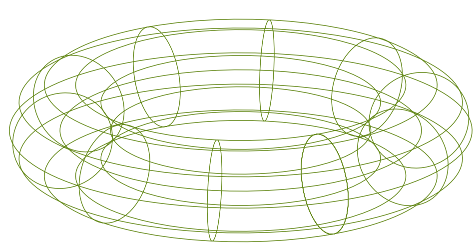

Figure 9.55: torus example

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.

Input slots (optional, defaulting):

Arc *Angle in Radians*

Indicates the end angle for the donut. Defaults to twice pi for a full-circle donut.

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.

- **typeset-block**

Mixins: base-object

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.

Input slots (optional):

Center *3D-point*

Center of the text. Specify this or start, not both.

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.

```
(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))
```

Figure 9.56: Example Code for vertical-dimension

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.

- **vertical-dimension**

Mixins: linear-dimension

Description Creates a dimension annotation along the vertical axis.

Input slots (optional):

Dim-text-start *3D Point*

Determines where the text will start. Defaults to reasonable location for horizontal-dimension.

Computed slots:

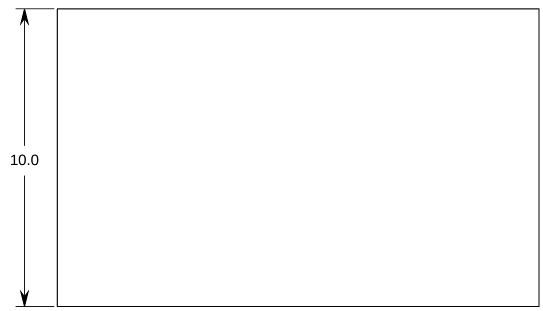

Figure 9.57: vertical-dimension example

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

9.12.2 Function and Macro Definitions

- **3d-distance**

  (type Number intro The three-dimensional distance from point-1 to point-2. arguments (point-1 3D point point-1 3D point))

- **3d-point-p**

  (type Boolean intro . FUNCTION 3d-point-p - predicate function to check if a make-point is 3D. That is, the point has 3 dimensions, representing a 3-dimensional point. USAGE 3d-point-p point DESCRIPTION A predicate function to check if a point is 3-dimensional. The function may also be accessed by calling the function 3d-point?. EXAMPLES (3d-point-p (make-point 1 2 3)) –¿ t (3d-point-p (make-point 1 2 3 4)) –¿ nil

)

- **3d-point?**

  (type Boolean intro A predicate function to check if a point is 3-dimensional.)

- **3d-vector-p**

(type Boolean intro . FUNCTION 3d-vector-p - predicate function to check if a vector is 3D. That is, the vector has 3 dimensions, representing a 3-dimensional vector. USAGE 3d-vector-p vector DESCRIPTION A predicate function to check if a vector is 3-dimensional. The function may also be accessed by calling the function 3d-vector?. EXAMPLES (3d-vector-p (make-vector 1 2 3)) −¿ t (3d-vector-p (make-vector 1 2 3 4)) −¿ nil

)

- **3d-vector-to-array**

  (type 3-by-1 Lisp array of double-floats intro Returns a 3-by-1 Lisp array of double-float numbers built from a 3D-Vector of double-floats. This can be useful for example for multiplying a GDL 3d-point (which is a 1-d vector) by a 3x3 matrix represented as a 2D Lisp array. arguments (vector 3D-Vector of double-floats (e.g. created with make-vector macro)))

- **3d-vector?**

  (type Boolean intro A predicate function to check if a vector is 3-dimensional.)

- **acosd**

  (type Number intro Returns the arc cosine of ¡b¿theta¡/b¿, converted into degrees. arguments (theta Number. An angle in radians))

- **add-matrices**

  (type Lisp Array intro Adds two matrices element-by-element. rest (matrics "Lisp Arrays of same dimensions"))

- **add-vectors**

  (type Vector intro Return a new vector, the result of affine vector addition. arguments (v1 2D, 3D, or 4D Vector v2 2D, 3D, or 4D Vector))

- **alignment**

  (type 3x3 Orthonormal Rotation Matrix intro Constructs a rotation matrix from the given axes and vectors. Up to three pairs of axis and vector can be given. If only one pair is given, then the orthogonal component of its vector with respect to the other two global axes is used. If a second pair is given, then the orthogonal component of its vector with respect to the first vector is used. A third pair is only required if a left-handed coordinate system is desired (right-handed is the default). The third vector will always be converted to the cross of the first two, unless it is given as the reverse of this, which will force a left-handed coordinate system. Axes are direction keywords which can be one of: ¡ul¿¡li¿¡tt¿:right¡/tt¿¡/li¿ ¡li¿¡tt¿:left¡/tt¿¡/li¿ ¡li¿¡tt¿:rear¡/tt¿¡/li¿ ¡li¿¡tt¿:front¡/tt¿¡/li¿ ¡li¿¡tt¿:top¡/tt¿¡/li¿ ¡li¿¡tt¿:bottom¡/tt¿¡/li¿¡/ul¿ The second axis keyword, if given, must be orthogonal to the first, and the third, if given, must be orthogonal to the first two. arguments (axis-1 Direction Keyword vector1 3D Vector) &optional (axis-2 Direction Keyword vector2 3D Vector axis-3 Direction Keyword vector3 3D Vector))

- **angle-between-vectors**

(type Number intro Returns the angle in radians between ¡b¿vector-1¡/b¿ and ¡b¿vector-2¡/b¿. If no ¡b¿reference-vector¡/b¿ given, the smallest possible angle is returned. If a ¡b¿reference-vector¡/b¿ is given, computes according to the right-hand rule. If ¡b¿-ve¡/b¿ is given, returns a negative number for angle if it really is negative according to the right-hand rule. arguments (vector-1 3D Vector vector-2 3D Vector) &optional ((reference-vector nil) 3D Vector) &key ((epsilon *zero-epsilon*) Number. Determines how small of an angle is considered to be zero. (-ve nil) Boolean))

- **angle-between-vectors-d**

  (type Number intro This function is identical to angle-between-vectors, but returns the angle in degrees. Refer to angle-between-vectors for more information. Technical note: the ¡b¿more¡/b¿ argument has been introduced to support both angle-between-vectors call conventions and the legacy signature: (vector-1 vector-2 &optional reference-vector negative?) Optionally, a deprecation warning is printed when code invokes this legacy pattern..)

- **apply-make-point**

  (type 2D, 3D, or 4D point intro This function takes a list of two, three, or four numbers rather than multiple arguments as with the make-point and make-vector macro. This is equivalent to calling the make-point or make-vector macro on the elements of this list. arguments (list List of 2, 3, or 4 numbers. The coordinates for the point.))

- **array-to-3d-vector**

  (type 3D Vector intro Returns a 3D-Vector of double-floats built from a 3-by-1 Lisp array of numbers. arguments (array 3-by-1 Lisp array of numbers))

- **array-to-list**

  (type List intro Converts ¡b¿array¡/b¿ to a list. arguments (array Lisp Array of Numbers) &optional ((decimal-places 2) Integer. Numbers will be rounded to this many decimal places.))

- **asind**

  (type Number intro Returns the arc sine of ¡b¿theta¡/b¿, converted into degrees. arguments (theta Number. An angle in radians))

- **atand**

  (type Number intro Returns the arc tangent of ¡b¿theta¡/b¿, converted into degrees. arguments (theta Number. An angle in radians))

- **coincident-point?**

  (type Boolean intro Returns non-NIL iff the distance between ¡b¿point-1¡/b¿ and ¡b¿point-2¡/b¿ is less than ¡b¿tolerance¡/b¿. arguments (point-1 3D Point point-2 3D Point) &key ((tolerance *zero-epsilon*) Number))

- **create-obliqueness**

  (type 3x3 Orthonormal Rotation Matrix intro Gives the transform required to be applied to the parent's orientation to achieve alignment indicated by the arguments. The direction keywords are the same as those used with the GDL ¡tt¿alignment¡/tt¿ function. arguments (vector-1 3D Vector direction-1 Direction Keyword vector-2 3D Vector direction-2 Direction Keyword self GDL object inheriting from ¡tt¿base-object¡/tt¿))

- **cross-vectors**

  (type 3D Vector intro Returns the cross product of vector-1 and vector-2. According to the definition of cross product, this resultant vector should be orthogonal to both ¡b¿vector-1¡/b¿ and ¡b¿vector-2¡/b¿. arguments (vector-1 3D Vector vector-2 3D Vector))

- **degree**

  (type Number intro Converts angle in degrees, minutes, and seconds into radians. arguments (degrees Number) &optional ((minutes 0) Number))

- **degrees-to-radians**

  (type Number intro Converts ¡b¿degrees¡/b¿ to radians. arguments (degrees Number))

- **distance-to-line**

  (type Number intro Returns shortest distance from point to line.)

- **dot-vectors**

  (type Number intro Returns the dot product of vector-1 and vector-2. arguments (vector-1 2D, 3D, or 4D Vector vector-2 2D, 3D, or 4D Vector))

- **equi-space-points**

  (type List of points intro Returns a list of equally spaced points between start and end.)

- **get-u**

  (type Double-float number intro Returns U component of 2D parameter value. arguments (point 2D point))

- **get-v**

  (type Double-float number intro Returns V component of 2D parameter value. n:arguments (point "2D point"))

- **get-w**

  (type Double-float number intro Returns W component of point or vector arguments (quaternion 4D point, Quaternion, or Axis-Angle style rotation spec))

- **get-x**

  (type Double-float number intro Returns X component of point or vector arguments (point 2D, 3D, or 4D point))

- **get-y**

  (type Double-float number intro Returns Y component of point or vector arguments (point 2D, 3D, or 4D point))

- **get-z**

  (type Double-float number intro Returns Z component of point or vector arguments (point 3D or 4D point))

- **inter-circle-sphere**

  (type 3D Point or NIL intro Returns point of intersection between the circle described by ¡b¿circle-center¡/b¿, ¡b¿circle-radius¡/b¿, and ¡b¿circle-plane-normal¡/b¿, and the sphere described by ¡b¿sphere-center¡/b¿ and ¡b¿sphere-radius¡/b¿. Iff the circle and sphere do not intersect at all, NIL is returned. arguments (circle-center 3D Point circle-radius Number circle-plane-normal 3D Vector sphere-center 3D Point sphere-radius Number positive-angle? Boolean. Controls which of two intersection points is returned) &key ((tolerance *zero-epsilon*) Controls how close the entities must come to touching to be considered as intersecting.))

- **inter-line-plane**

  (type 3D Point or NIL intro Returns one point of intersection between line described by point ¡b¿p-line¡/b¿ and direction-vector ¡b¿u-line¡/b¿, and plane described by ¡b¿p-plane¡/b¿ and ¡b¿u-plane¡/b¿. Iff the line and plane do not intersect at all (i.e. they are parallel), NIL is returned. arguments (p-line 3D Point. Any point on the line. u-line 3D Vector. Direction of the line. p-plane 3D Point. Any point on the plane. u-plane 3D Vector. Normal of the plane.))

- **inter-line-sphere**

  (type 3D Point or NIL intro Returns one point of intersection between line described by point ¡b¿p-line¡/b¿ and direction-vector ¡b¿u-line¡/b¿, and sphere described by ¡b¿center¡/b¿ and ¡b¿radius¡/b¿. Iff the line and sphere do not intersect at all, NIL is returned. arguments (p-line 3D Point. Any point on the line. u-line 3D Vector. Direction of the line. center 3D Point. Center of the sphere. radius Number. The radius of the sphere. side-vector 3D Vector. Controls which of two possible intersection points is returned.))

- **length-vector**

  (type Number intro Return the vector's magnitude arguments (vector 3D Vector))

- **make-point [Macro]**

  (type 3D Point intro (Internally this is the same as a 3D Vector) Returns a vector of double-floats from up to 4 numbers.)

- **make-transform**

  (type Lisp array intro Builds a matrix from ¡b¿list-of lists¡/b¿. arguments (list-of-lists List of lists of numbers))

- **make-vector [Macro]**

  (type 0D, 1D, 2D, 3D, or 4D Vector intro (Internally this is the same as a Point) Returns a vector of double-floats from up to 4 numbers.)

- **matrix*vector**

  (type Lisp array intro Multiplies matrix by column vector of compatible dimension. arguments (matrix Lisp Array of Numbers vector Vector))

- **matrix-to-quaternion**

  (type Quaternion represented as a 4D Vector intro Transforms rotation ¡b¿matrix¡/b¿ into the corresponding quaternion. arguments (matrix 3x3 Orthonormal Rotation Matrix (as a Lisp Array of Numbers)))

- **merge-display-controls [Macro]**

  (type Plist of display controls intro This macro "merges" the given display controls list with that coming as a trickle-down slot from the parent. It will replace any common keys and add any new keys. arguments (display-controls Plist. The new display controls to be merged with the defaults from the parent))

- **midpoint**

  (type 3D Point intro Returns the barycentric average (i.e. midpoint) of ¡b¿point1¡/b¿ and ¡b¿point2¡/b¿. arguments (point1 3D Point point2 3D Point))

- **multiply-matrices**

  (type Lisp Array intro Multiplies compatible-size matrices according to normal matrix math. arguments (matrix-1 Lisp Array of Numbers matrix-2 Lisp Array of Numbers))

- **orthogonal-component**

  (type 3D Unit Vector intro Returns the unit vector orthogonal to ¡b¿reference-vector¡/b¿ which is as close as possible to ¡b¿vector¡/b¿. arguments (vector 3D Vector reference-vector 3D Vector))

- **parallel-vectors?**

  (type Boolean intro Returns non-nil iff ¡b¿vector-1¡/b¿ and ¡b¿vector-2¡/b¿ are pointing in the same direction or opposite directions. arguments (vector-1 3D Vector vector-2 3D Vector) &key ((tolerance *zero-epsilon*) Number (directed? nil) Boolean. If :directed? is t, the function returns t if the vectors are both parallel and point in the same direction. The default is nil, meaning that the function will return t regardless of which way the vectors point, as long as they are parallel.))

- **point-on-plane?**

  (type Boolean intro Determines whether or not the ¡tt¿3d-point¡/tt¿ lies on the plane specified by ¡tt¿plane-point¡/tt¿ and ¡tt¿plane-normal¡/tt¿, within ¡tt¿tolerance¡/tt¿. arguments (3d-point Point in question plane-point point on the known plane plane-normal normal to the known plane) &key ((tolerance *zero-epsilon*) Tolerance for points to be considered coincident.))

- **point-on-vector?**

(type Boolean intro Determines whether or not the ¡tt¿unknown-point¡/tt¿ lies on the ray specified by the vector pointing from ¡tt¿first-point¡/tt¿ to ¡tt¿second-point¡/tt¿, within ¡tt¿tolerance¡/tt¿. arguments (first-point first point of vector second-point second point of vector unknown-point point in question) &key ((tolerance *zero-epsilon*) Tolerance for vectors to be considered same-direction.))

- **proj-point-on-line**

  (type 3D-Point intro Drops ¡b¿3d-point¡/b¿ onto line containing ¡b¿line-point¡/b¿ and whose direction-vector is ¡b¿vector¡/b¿.

  arguments (3D-point 3D Point Line-point 3D Point vector 3D Unit Vector))

- **projected-vector**

  (type 3D Vector intro Returns result of projecting ¡b¿vector¡/b¿ onto the plane whose normal is ¡b¿plane-normal¡/b¿. arguments (vector 3D Vector plane-normal 3D Vector))

- **pythagorize**

  (type Number intro Returns the square root of the sum of the squares of ¡i¿numbers¡/i¿. &rest (numbers List of Numbers))

- **quaternion-to-matrix**

  (type 3x3 Orthonormal Rotation Matrix intro Transforms ¡b¿quaternion¡/b¿ into a 3x3 rotation matrix. arguments (quaternion Quaternion, represented as a 4D Vector))

- **quaternion-to-rotation**

  (type Euler rotation represented as a 4D Vector intro Transforms ¡b¿quaternion¡/b¿ into a Euler angle rotation consisting of an arbitrary axis and an angle of rotation about that axis. arguments (quaternion Quaternion, represented as a 4D Vector))

- **radians-to-degrees**

  (type Number intro Converts angle in radians to degrees. arguments (radians Number))

- **radians-to-grads**

  (type Number intro Converts angle in radians to grads. arguments (radians Number))

- **reverse-vector**

  (type Vector intro Return the vector pointing in the opposite direction. arguments (vector 2D, 3D, or 4D Vector))

- **roll [Macro]**

  (type [macro] Transformation matrix intro In the context of a GDL object definition (i.e. in a ¡tt¿define-object¡/tt¿), returns a transformation matrix based on rotation about ¡b¿axis¡/b¿ by some ¡b¿angle¡/b¿. ¡b¿Axis¡/b¿ is a keyword symbol, one of: ¡ul¿¡li¿¡tt¿:lateral¡/tt¿¡/li¿ ¡li¿¡tt¿:longitudinal¡/tt¿¡/li¿ ¡li¿¡tt¿:vertical¡/tt¿¡/li¿¡/ul¿ ¡b¿Angle¡/b¿ is specified in radians. Any number of axis-angle pairs can be specified. arguments (axis Keyword Symbol angle Number) &rest (other-axes-and-angles Plist made from axis keyword symbols and numbers))

- **rotate-point**

  (type 3D Point intro Returns the 3D Point resulting from rotating ¡b¿point¡/b¿ about ¡b¿center¡/b¿ in the plane defined by ¡b¿normal¡/b¿. The rotation can specified either by an arc length (¡b¿arc-length¡/b¿) or an angle in radians (¡b¿angle¡/b¿). A second value is returned, which is the resulting angle of rotation in radians (this is of possible use if ¡b¿arc-length¡/b¿ is used to specify the rotation). arguments (point 3D Point center 3D Point normal 3D Vector) &key ((arc-length nil) Number (angle nil) Number))

- **rotate-point-d**

  (type 3D Point intro Returns the 3D Point resulting from rotating ¡b¿point¡/b¿ about ¡b¿center¡/b¿ in the plane defined by ¡b¿normal¡/b¿. The rotation can specified either by an arc length (¡b¿arc-length¡/b¿) or an angle in degrees (¡b¿angle¡/b¿). A second value is returned, which is the resulting angle of rotation in degrees (this is of possible use if ¡b¿arc-length¡/b¿ is used to specify the rotation). arguments (point 3D Point center 3D Point normal 3D Vector) &key ((arc-length nil) Number (angle nil) Number))

- **rotate-vector**

  (type Number intro Rotates ¡b¿vector¡/b¿ around ¡b¿normal¡/b¿ by an amount of rotation specified by ¡b¿angle¡/b¿, which is an angle measured in radians. arguments (vector 3D Vector angle Number normal 3D Vector))

- **rotate-vector-d**

  (type Number intro Rotates ¡b¿vector¡/b¿ around ¡b¿normal¡/b¿ by an amount of rotation specified by ¡b¿degrees¡/b¿. arguments (vector 3D Vector degrees Number normal 3D Vector))

- **rotation**

  (type 3x3 orthonormal rotation matrix (as a Lisp Array of Numbers) intro . Returns a transformation matrix based on a rotation by ¡b¿angle¡/b¿, specified in radians, about an arbitrary ¡b¿vector¡/b¿. arguments (vector 3D Vector angle Number))

- **same-direction-vectors?**

  (type Boolean intro Returns non-NIL iff ¡b¿vector-1¡/b¿ and ¡b¿vector-2¡/b¿ are pointing in the same direction. arguments (vector-1 3D Vector vector-2 3D Vector) &key ((tolerance *zero-epsilon*) Number))

- **scalar*matrix**

  (type Lisp Array intro Returns result of multiplying the scalar number by the matrix. arguments (scalar Number matrix Lisp Array of Numbers))

- **scalar*vector**

  (type Vector intro Returns result of multiplying the scalar number by the vector arguments (scalar Number vector 2D, 3D, or 4D Vector))

- **sort-points-along-vector**

  (type List of points intro Returns points in order along given vector.)

- **subtract-vectors**

  (type Vector intro Return a new vector, the result of affine vector subtraction. arguments (v1 2D, 3D, or 4D Vector v2 2D, 3D, or 4D Vector))

- **transform-and-translate-point**

  (type 3D-Point intro Returns the product of ¡b¿vector¡/b¿ and ¡b¿transform¡/b¿, translated by (i.e. added to) ¡i¿trans-vector¡/i¿. arguments (vector 3D Vector transform 3x3 Rotation Matrix trans-vector 3D Vector) examples ¡pre¿ (let ((transform (make-transform '((0.0 0.0 1.0) (0.0 1.0 0.0) (1.0 0.0 0.0)))) (v (make-vector 1.0 2.0 3.0)) (t-v (make-vector 3.0 0.0 0.0))) (transform-and-translate-point v transform t-v)) —¿ #(6.0 2.0 1.0) ¡/pre¿)

- **transform-numeric-point**

  (type 3D-Point intro Returns the product of ¡b¿vector¡/b¿ and ¡i¿transform¡/i¿. arguments (vector 3D Vector transform 3x3 Rotation Matrix) examples ¡pre¿ (let ((transform (make-transform '((0.0 0.0 1.0) (1.0 0.0 0.0) (0.0 1.0 0.0)))) (v (make-vector 1.0 2.0 3.0))) (transform-numeric-point v transform)) —¿ #(2.0 3.0 1.0) ¡/pre¿)

- **translate [Macro]**

  (type [Macro] 3D Point intro Within the context of a GDL object definition (i.e. a ¡tt¿define-object¡/tt¿), translate ¡b¿origin¡/b¿ by any number of ¡b¿offsets¡/b¿. arguments (origin 3D Point) &rest (offsets Plist consisting of direction keywords and numbers. A direction keyword can be one of: ¡ul¿¡li¿¡tt¿:top¡/tt¿ (or ¡tt¿:up¡/tt¿)¡/li¿ ¡li¿¡tt¿:bottom¡/tt¿ (or ¡tt¿:down¡/tt¿)¡/li¿ ¡li¿¡tt¿:left¡/tt¿¡/li¿ ¡li¿¡tt¿:right¡/tt¿¡/li¿ ¡li¿¡tt¿:front¡/tt¿¡/li¿ ¡li¿¡tt¿:rear¡/tt¿ (or ¡tt¿:back¡/tt¿)¡/li¿¡/ul¿))

- **translate-along-vector**

  (type 3D Point intro Returns a new point which is ¡b¿point¡/b¿ translated along ¡b¿vector¡/b¿ by ¡b¿distance¡/b¿ arguments (point 3D Point vector 3D Vector distance Number))

- **transpose-matrix**

  (type Lisp array intro Transposes rows and columns of ¡b¿matrix¡/b¿. arguments (matrix Lisp Array))

- **unitize-vector**

  (type Unit Vector intro Returns the normalized unit-length vector corresponding to ¡b¿vector¡/b¿. arguments (vector 3D Vector) &key ((espsilon *zero-epsilon*) Number. How close vector should be to 1.0 to be considered unit-length.))

- **zero-vector?**

  (type Boolean intro Returns non-NIL iff the vector has zero length according to Common Lisp ¡tt¿zerop¡/tt¿ function. arguments (vector 3D Vector))

9.12.3 Variables and Constants

- ***break-leaders?***

- ***gs-graphics-alpha-bits***

- ***gs-text-alpha-bits***
- ***hash-transforms?***
- ***zero-vector-checking?***
- **+postnet-bits+**

9.13 glm

9.14 gwl (Generative Web Language (GWL))

9.14.1 Object Definitions

- **application-mixin**

  Mixins: layout-mixin, vanilla-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`

- **base-ajax-graphics-sheet**

  Mixins: base-ajax-sheet, base-html-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.

  Input slots (optional):

  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.

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).]

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.

Input slots (optional, defaulting):

Respondent *GDL Object*

Object to respond to the form submission. Defaults to self.

Computed slots (settable):

Dropped-height-width *Plist with :width and :height*

The dimensions of the bounding-box of the dragged and/or dropped element.

Dropped-object *List representing GDL root-path*

This is the root path of the dragged and/or dropped object. This is not tested to see if it is part of the same object tree as current self.

Dropped-x-y *3D point*

This is the upper-right corner of the bounding box of the dragged and/or dropped element.

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.

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.

X3dom-graphics *String of valid HTML*

This can be used to include the x3dom tag content for 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.

View-object *GDL web-drawing object*

This must be overridden in the specialized class.

Gdl 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`.

- **base-ajax-sheet**

  Mixins: base-html-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.

  Input slots (optional):

  Body-class *String or nil*

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

  Body-onload *String of Javascript or nil*

  This Javascript will go into the :onload event of the body. Default is nil.

  Body-onpageshow *String of Javascript or nil*

  This Javascript will go into the :onpageshow event of the body. Default is nil.

  Doctype-string *String or nil*

  Contains the string for the doctype at the top of the document. Default is the standard doctype for HTML5 and later.

  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.

  Input slots (optional, settable):

  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.

```
(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"))))
```

Additional-header-js-content *valid javascript*

This javascript is added to the head of the page, just before the body.

Ui-specific-layout-js *Absolute URI in the browser*

. This is additional JavaScript that needs to be loaded in order to initiate the layout of a user interface. Defaults to nil.

Use-jquery? *Boolean*

Include jquery javascript libraries in the page header? Default nil.

Computed slots:

Development-links *String of HTML*

Provides the developer control links for current sheet.

Gdl functions:

Custom-snap-restore! *Void*

This is a hook function which applications can use to restore automatically from a saved snapshot file.

- **base-form-control**

Mixins: skeleton-form-control, vanilla-mixin

Author Dave Cooper, Genworks

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.

Input slots (optional):

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.

Ajax-submit-on-change? *Boolean*

If set to non-nil, this field's value will be sent to server upon change. Default is nil.

Ajax-submit-on-enter? *Boolean*

If set to non-nil, this field's value will be sent to server upon enter. 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.

```
(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
```

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),

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, unless ajax-submit-on-change? is non-nil, in which case it calls ajax to set current form value.

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.

Onenter *String or nil*

Maps to HTML form control attribute of the same name. Default is nil, unless ajax-submit-on-enter? is non-nil, in which case it calls ajax to set current form value.

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.

Placeholder *String*

Text to place in the field by default, overwritten as soon as the field is selected. Works only in HTML5. 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).

Computed slots (settable):

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.

Gdl functions:

Restore-defaults! *Void*

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

- **base-html-graphics-sheet**

Mixins: base-html-sheet, geometry-view-mixin, base-object

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.

Input slots (optional):

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

Input slots (optional, settable):

Digitation-mode *Keyword symbol, one of `:zoom-and-center`, `:report-point`, or `:measure-distance`*

.

- 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.

Gdl 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.

- **base-html-sheet**

Mixins: sheet-section, vanilla-mixin

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.

Input slots (optional):

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? *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.

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.

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).

Input slots (optional, defaulting):

Respondent *GDL Object*

Object to respond to the form submission. Defaults to self.

Computed slots (settable):

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.

Gdl functions:

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.

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.

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.

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.

- **checkbox-form-control**

Mixins: base-form-control, vanilla-mixin

Author Dave Cooper, Genworks

Description This represents a INPUT of TYPE CHECKBOX

Input slots (optional):

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.

- **color-map**

Mixins: base-html-sheet

Description Shows a list of the default colors. This is published as the URI "/color-map" of the running GWL webserver.

Gdl functions:

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:

- **geometry-view-mixin**

Mixins: vanilla-mixin

Description Internal mixin for use inside e.g. base-html-graphics-sheet.

Input slots (optional):

Length *Number*

Length ("height" of screen window) of the graphics viewport. Default is 300.

View-object *GDL web-drawing object*

This must be overridden in the specialized class.

Width *Number*

Width of the graphics viewport. Default is 300.

- **grid-form-control**

Mixins: skeleton-form-control, vanilla-mixin

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.

Input slots (optional):

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:

Form-controls *List of GDL objects*

All the children or hidden-children of type base-form-control.

- **gwl-rule-object**

  Mixins: base-html-graphics-sheet, base-rule-object

  Description Used to display a rule as a GWL web page. Mixes together `base-html-sheet` and `base-rule-object`.

- **layout-mixin**

  Mixins: base-html-graphics-sheet

  Description This is mixed into both `node-mixin` and `application-mixin`. It contains the common messages for nodes in a GWL application tree. For any `node-mixin` or `application-mixin`, you may override the default (empty) `model-inputs` output-function of the corresponding html-format view to make specific model-inputs for that node.

Input slots (optional):

Available-image-formats *List of keyword symbols*

Determines which formats are available in the Preferences. Defaults to :png, :jpeg, and :vrml.

Body-bgcolor *Keyword symbol*

Color keyword from `*color-table*` for the body background. Defaults to `:blue-sky`.

Height *Number*

Z-axis dimension of the reference box. Defaults to zero.

Image-format *Keyword symbol*

Determines the default image format. Defaults to :png

Inputs-bgcolor *Keyword symbol*

Color keyword from `*color-table*` for the model-inputs area background. Defaults to `:aquamarine`.

Inputs-title *String*

Title for the model-inputs section. Defaults to "Model Inputs".

Length *Number*

Length ("height" of screen window) of the graphics viewport. Default is 300.

Multipart-form? *Boolean*

Determines whether the embedded form will support multipart MIME parts. Defaults to NIL.

Other-rules *List of GDL objects of type `base-rule-object` or (preferably) `gwl-base-rule-object`*

. Links to these will be displayed in the other-rules section. Default to the collection of all objects of type `base-rule-object` from this node in the tree down to the leaves, whose `violated?` message evaluates to NIL.

Other-rules-bgcolor *Keyword symbol*

Color keyword from `*color-table*` for the other-rules area background. Defaults to `:aquamarine`.

Other-rules-title *String*

Title for the other-rules section. Defaults to "Other Rules".

Page-title *String*

The title to display on the page and in the tree. Defaults to `(the strings-for-display)`.

Show-title? *Boolean*

Indicates whether to display the title at the top of the page. Defaults to T.

Tree-bgcolor *Keyword symbol*

Color keyword from `*color-table*` for the tree area background. Defaults to `:aquamarine`.

Tree-title *String*

Title for the Tree section. Defaults to "Assembly Tree" if the tree-root is only a subclass of `application-mixin`, and "Assembly Tree" if the tree-root is an actual node with child applications.

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 objects*

This should be overridden with a list of objects of your choice. The leaves of these objects will be scaled to fit and displayed in the graphics area. Defaults to NIL.

Violated-rules *List of GDL objects of type `base-rule-object` or (preferably) `gwl-base-rule-object`*

. Links to these will be displayed in the other-rules section. Default to the collection of all objects of type `base-rule-object` from this node in the tree down to the leaves, whose `violated?` message evaluates to non-NIL.

Violated-rules-bgcolor *Keyword symbol*

Color keyword from `*color-table*` for the violated-rules area background. Defaults to `:aquamarine`.

Violated-rules-title *String*

Title for the violated-rules section. Defaults to "Violated Rules".

Width *Number*

Width of the graphics viewport. Default is 300.

Input slots (optional, defaulting):

Display-rules? *Boolean*

Indicates whether the Rules panel should be displayed. Defaults to T.

Display-tree? *Boolean*

Indicates whether the Tree area should be displayed. Defaults to T.

Graphics-height *Integer*

Height (top to bottom on screen) in pixels of the graphics area. Defaults to 500.

Graphics-width *Integer*

Height (left to right on screen) in pixels of the graphics area. Defaults to 500.

Use-standard-saved-slots? *Boolean*

Determines whether the standard-saved-slots are automatically used by default for the saved-slots. This is a trickle-down slot so its value will be passed to descendent objects automatically. The default value is NIL.

Computed slots:

Saved-slots *List of keyword symbols or lists*

. The first of this list should be the unique name for this tree node for the purposes of saving slots. The rest of this list is made up of either keyword symbols or lists. A keyword symbol indicates the name of a slot to be saved in the current object. These slot names should correspond to `:settable` slots of this object. A list indicates slots to be saved in a child object, specified as follows: the first of the list is the name of the child part, and the rest is made up of keywords naming the slots in the child part to be saved. These should correspond to `:settable` slots in the child object. The default value is the `standard-saved-slots` if the `use-standard-saved-slots?` is non-NIL, NIL otherwise.

Standard-saved-slots *List of keyword symbols*

The first of this list is the `name-for-display` of this object. The rest of the list are all the keyword symbols representing the settable computed-slots and input-slots which have a default value. Required input-slots (i.e. input-slots without a default value) are not included in this list. If you wish to include required inputs with the saved-slots, you should explicitly append them to this list when specifying the `saved-slots`.

Hidden objects:

View-object *GDL web-drawing object*

This must be overridden in the specialized class.

Gdl functions:

Read-saved-slots *Void*

Reads the slots data from `filename`, restores the corresponding slots in this object and matching descendant objects, and calls the `restore!` function on each object.

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-saved-slots *Void*

Writes the unique application name names and values of all saved-slots in this and all descendants which are of type node-mixin or application-mixin.

- **menu-form-control**

Mixins: base-form-control, vanilla-mixin

Author Dave Cooper, Genworks

Description This represents a SELECT form control tag wrapping some OPTION tags. OPTIONGROUP is not yet implemented, but will be.

Input slots (optional):

Choice-list *List*

Display values, also used as return values, for selection list. Specify this or choice-plist, not both.

```
...

:objects
((menu-1 :type 'menu-form-control
         :choice-plist (list 1 "one" 2 "two")))

...
```

Figure 9.60: Example Code for gwl:menu-form-control

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.

- **node-mixin**

Mixins: layout-mixin, vanilla-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.

Input slots (optional):

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`.

- **radio-form-control**

Mixins: menu-form-control, vanilla-mixin

Description Produces a standard radio-button form control.

Input slots (optional):

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.

- **session-control-mixin**

Mixins: vanilla-mixin

Author Brian Sorg, Liberating Insight LLC (revised Dave Cooper, Genworks)

Description Mixin to the root object of the part which you wish to have session control over

Input slots (optional):

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.

Input slots (optional, settable):

Expires-at Universal time after which the session should expire

Gdl functions:

Clear-expired-session This is the function called to check for and handle session control

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.

Set-expires-at Method which will set the expires-at slot to the current time + the session-duration

- **sheet-section**

  Mixins: skeleton-ui-element, vanilla-mixin

  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.

- **skeleton-form-control**

  Mixins: skeleton-ui-element, vanilla-mixin

```
FLAG -- fill in!!!
```

Figure 9.61: Example Code for gwl:sheet-section

Author Dave Cooper, Genworks

Description Computes standard values for base-form-control and similar container objects, e.g. grid-form-control.

Does not perform the actual bashing and computation of result value, should be mixed in to something which does this.

Input slots (optional):

Class *String*

You can use this to specify a user-defined class for the form-control. Defaults to nil, which means no class attribute will be generated.

Field-name *Keyword symbol*

The name of this field. Computed from the object name within the tree.

Id *Keyword symbol*

The ID attribute for this tag. Defaults to (the field-name).

Primary? *Boolean*

Set this to t if the form-control should always occur first in an outputted snapshot file. Defaults to nil.

Computed slots:

Form-control *String of valid HTML*

This is the default HTML which can be included in a form in a web page to display this form control. Previously known as form-control-string. Default is the form-control-string.

Form-control-string *String of valid HTML*

Also known as simply form-control. This is the default HTML which can be included in a form in a web page to display this form control. Default is the output from form-control method of the lens for html-format and the specific type of this object, returned as a string.

Form-controls *List of GDL objects*

All the children or hidden-children of type base-form-control.

Html-string *String of valid HTML*

This is the default HTML which can be included in a form in a web page to display this form control, wrapped with labels and table cells.

```
FLAG -- Fill in!!!
```

Figure 9.62: Example Code for gwl:skeleton-ui-element

- **skeleton-ui-element**

  Mixins: vanilla-mixin

  Description 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.

  Input slots (optional):

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.

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.

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.

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.

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.

Input slots (optional, defaulting):

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.

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.

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.

Gdl 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.

- **text-form-control**

  Mixins: base-form-control, vanilla-mixin

  Author Dave Cooper, Genworks

  Description This represents a INPUT TYPE=TEXT or TEXTAREA form control tag.

  Input slots (optional):

  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.

- **web-drawing**

  Mixins: renderer-mixin, base-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.

  Input slots (optional):

  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).

```
(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 ")))
```

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.

9.14.2 Function and Macro Definitions

- **base64-decode-list**

  (type List intro Decodes a base64 string into a Lisp list. arguments (string string))

- **base64-decode-safe**

  (type String intro Decodes a base64 string without need for trailing = signs into a decoded string. arguments (string string))

- **base64-encode-list**

  (type String intro Encodes a list into base64 without the trailing = signs. arguments (list list))

- **base64-encode-safe**

  (type String intro Encodes a string into base64 without the trailing = signs. arguments (string string))

- **clear-all-instances**

  (type Void intro Clears all instances from GWL's master table of root-level instances. The instance IDs are the numbers you see in published GWL URIs, and are available as the "instance-id" message within each GWL object which inherit from base-html-sheet. Clearing all the instances makes available for garbage collection all memory used by the object hierarchies rooted at the instances, as well as all associated published URIs. example ¡pre¿ (clear-all-instance) ¡/pre¿)

- **clear-instance**

  (type Void intro Clears the specified instance from GWL's master table of root-level instances. The instance ID is the same number you see in published GWL URIs, and is available as the "instance-id" message within all GWL objects which inherit from base-html-sheet. Clearing the specified instance makes available for garbage collection all memory used by the object hierarchy rooted at the instance, as well as all associated published URIs. arguments (id Integer or Keyword Symbol. The key whose entry you wish to clear from the *instance-hash-table*.) example ¡pre¿ (clear-instance 639) ¡/pre¿)

- **clear-old-timers**

  (type Void intro This is a lighter-weight alternative to the session-object-mixin for timing out instances in a web application. &key ((idle-time-required 600) Time in seconds. The maximum age of a session for timeout.))

- **crawl**

  (type Void intro Instantiates and "Crawls" a given object instance and creates static HTML pages reflecting the instance tree. This means it will recursively follow all the links for the object. By default the files are written into "/tmp/sites/". arguments (part String. Names a package-qualified part which should mix in ¡tt¿base-html-sheet¡/tt¿.) &key ((host localhost) String. Host on which the server is running. (port 9000) Integer. Port on which the server is running. (output-root /tmp/sites/[non-package-qualified-part-name]/) String or pathname. Directory where filfes will be written make-part-args Plist. Other make-instance arguments to use to initialize the object.) example ¡pre¿ (gwl:crawl "yadd:assembly") ¡/pre¿)

- **gwl-make-object**

  (type Void intro Used within the context of the body of a :function argument to Allegroserve's publish function, makes an instance of the specified part and responds to the request with a redirect to a URI representing the instance. arguments (req Allegroserve request object, as used in the function of a publish ent Allegroserve entity object, as used in the function of a publish package-and-part String. Should name the colon- (or double-colon)-separated package-qualified object name) &key ((make-object-args nil) Plist of keys and values. These are passed to the object upon instantiation. (share? nil) Boolean. If non-nil, the instance ID will be the constant string "share" rather than a real instance id.) example ¡pre¿ (publish :path "/calendar" :function #'(lambda(req ent) (gwl-make-object req ent "calendar:assembly"))) ¡/pre¿)

- **publish-gwl-app**

  (type Void intro Publishes an application, optionally with some initial arguments to be passed in as input-slots. arguments (path String. The URL pathname component to be published. string-or-symbol String or symbol. The object type to insantiate.) &key (make-object-args Plist. Extra arguments to pass to make-object.))

- **publish-shared**

  (type Void intro Used to publish a site which is to have a shared toplevel instance tree, and no URI rewriting (i.e. no "/sessions/XXX/" at the beginning of the path). So, this site will appear to be a normal non-dynamic site even though the pages are being generated dynamically. &key ((path nil) String. The URI path to be published. (object-type nil) Symbol. The type of the toplevel object to be instantiated. (host nil) hostname for the URI to be published. (server *wserver*) Allegroserve server object. If you have additional servers other than the default ¡tt¿*wserver*¡/tt¿ (e.g. an SSL server) you may want to call this function for each server.) example ¡pre¿ (publish-shared :path "/" :object-type 'site:assembly :host (list "www.genworks.com" "ww2.genworks.com" "mccarthy.genworks.com")) ¡/pre¿)

- **publish-string-content**

  (type String (representing a url path) intro Publishes given url to respond with text content as specified by given string. arguments (url String. The url to be published. string String. The content to be emitted when the url is requested.) &rest (publish-args plist. Arguments to be passed on to publish function, e.g. :content-type.))

- **relativize-pathname**

  (type nil intro Return a relative pathname for TARGET-PATHNAME that can be reached from the directory that TARGET-PATHNAME refers to.)

- **session-control-auto-refresh**

  (type Adding this javascript function into the header of a web page will cause the page to timeout and reload repeatedly intro This is intended to be used such that when an instance is open in an active browser the page will automatically update the expires-at function even if the operator takes an extended break from the application. It works by checking if any forms exist on this page. If they do it will submit the first form on the page when the timeout value is reached. This is done to avoid the Post Data confirmation warning that most browser present. If no forms are found it will use the reload(true) function to reload the page. arguments (timeout Time in seconds between page reloads) &optional (html-stream Stream which the output should be sent to. Default is *html-stream*))

- **session-report**

  (type Returns list of instances in a runtime environment intro Those that are of type session-control-mixin, it provides more detailed information, that can be useful in tracking the session life. Currently, this is intended to run from the lisp command prompt.)

- **with-cl-who [Macro]**

  (type Form intro Sets up body to be evaluated with cl-who and output the resulting string to the default *stream* Note that the args are spliced into cl-who:with-html-output after *stream* nil, so for example you can do ¡pre¿ (with-cl-who (:indent t) ...) ¡/pre¿ and it will expand into: ¡pre¿ (with-html-output (*stream* nil :indent t) ...) ¡/pre¿ .)

- **with-cl-who-string [Macro]**

  (type Form intro Sets up body to be evaluated with our with-cl-who return the resulting string instead of side-effecting anything at all to the default *stream*.)

- **with-html-form [Macro]**

  (type Enclose a body of code with a form intro . FLAG – fill in.)

9.14.3 Variables and Constants

- ***break-on-set-self?***
- ***bypass-security-check?***
- ***developing?***
- ***ent***
- ***failed-request-url***
- ***instance-finalizers***
- ***instance-hash-table***
- ***jump-to-toplevel-on-set-self?***
- ***max-id-value***
- ***publishers***
- ***query***
- ***reap-expired-sessions?***
- ***recovery-url-default***
- ***req***

9.15 iq

9.16 jquery

9.17 raphael

9.18 robot (Simplified Android Robot example)

9.19 smlib (Interface to SMLib Geometry Engine)

9.20 surf (NURBS Surface and Solids Geometry Primitives)

9.20.1 Object Definitions

- **3d-approximated**

  Mixins: approximated-curve, vanilla-mixin

  Description Given a curve in uv parameter space of a surface, produce a corresponding 3d curve with brute-force fitting techniques.

Input slots (required):

Surface *GDL Surface object*
The surface corresponding to the given uv-curve.

Uv-curve *GDL Curve object*
Curve whose points are understood to be 2D u, v parameter values on the surface.

Input slots (optional):

Number-of-samples *Integer*
How many point samples to take for fitting purposes. Default is 42.

Tolerance-factor *Number*
The tolerance for the final approximation, to be multiplied by the 3d curve length. Default is 10e-5.

Computed slots:

Curve-in *GDL Curve object*
The curve to be approximated with this curve.

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.

- **3d-curve**

Mixins: b-spline-curve, vanilla-mixin

Description Given a uv on-surface curve and its surface, produce the corresponding 3d curve. Note this must be a true on-surface curve, not just an arbitrary curve whose domain happens to be in the parameter space of the surface.

Input slots (required):

Surface *GDL Surface object*
The surface corresponding to the given uv-curve.

Uv-curve *GDL Curve object*
Curve which is a proper on-surface curve (e.g. produced by a project or drop operation onto the surface).

Computed slots:

Control-points *List of 3D Points*
The control points.

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.

- **approximated-curve**

Mixins: 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.

NOTE: This object is currently non-functional and is expected to be re-activated in GDL 1585.

Input slots (required):

Curve-in *GDL Curve object*

The curve to be approximated with this curve.

Input slots (optional):

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

```
(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)
 #(-7.219456797793601 10.0 0.0687181850892115)
 #(-7.21363748676987 10.0 0.0693049508093313)
 #(-7.203721331045569 10.0 0.0702991235190865)
```

Figure 9.65: approximated-curve example

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..

- **arc-curve**

  Mixins: curve, arc

  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.

- **b-spline-curve**

  Mixins: 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.

  Input slots (required):

  Control-points *List of 3D Points*

  The control points.

  Input slots (optional):

```
 (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))
```

Figure 9.66: Example Code for arc-curve

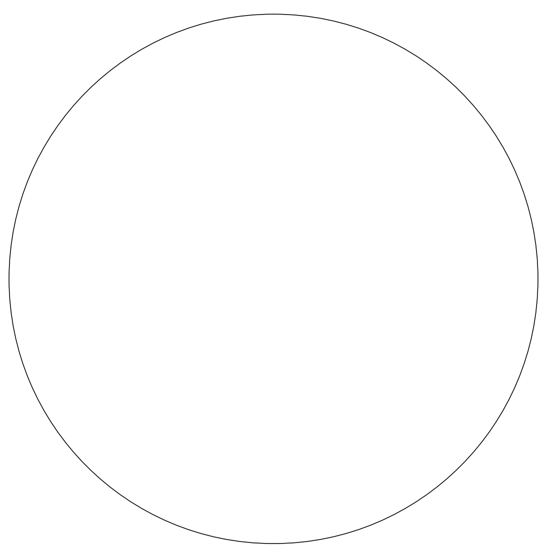

Figure 9.67: arc-curve example

```
(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))
```

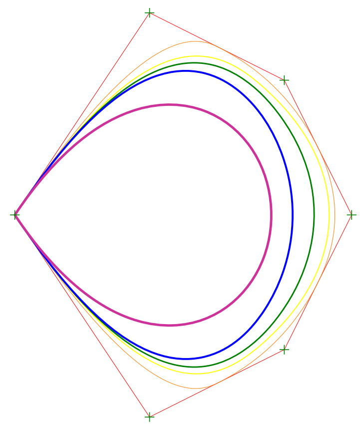

Figure 9.69: b-spline-curve example

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.

Local? *Boolean*

Indicates whether the inputted control-points should be considered in local coordinate system of this object. Default is nil.

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.

- **b-spline-surface**

Mixins: surface

Description A general b-spline surface specified with control points, weights, knots, and degree.

Input slots (required):

Control-points *List of lists of 3D Points*

The control net.

Input slots (optional):

```
(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)
```

Figure 9.70: Example Code for b-spline-surface

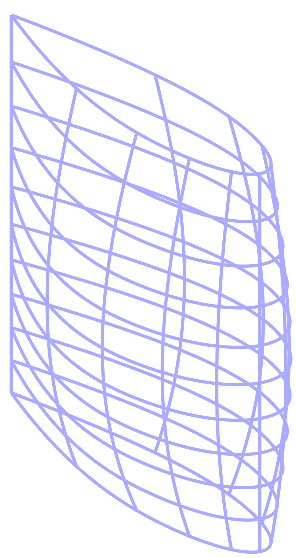

Figure 9.71: b-spline-surface example

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.

- **basic-surface**

Mixins: surface

Description This routine constructs a 3D surface by interpolating a set of four boundary 3D curves.

Input slots (required):

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.

- **blended-solid**

Mixins: brep

Description This primitive attempts to fillet one or more edges of a brep solid.

Input slots (required):

Brep *GDL Brep object*

This is the original solid, whose edges you want to be filleted.

```
(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)
```

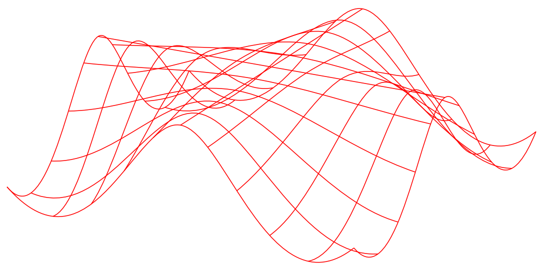

Figure 9.73: basic-surface example

```
(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))
```

Figure 9.74: Example Code for blended-solid

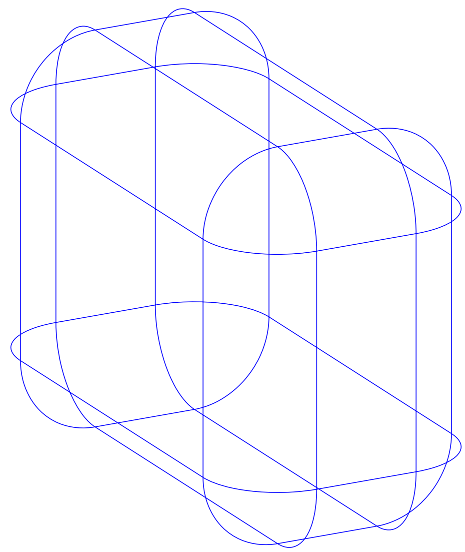

Figure 9.75: blended-solid example

Default-radius *Number*
This will be used as the fillet radius.

Input slots (optional):

Specs *Plist with key :edges*
This specifies which edges are to be filleted. The default (nil) means that all edges should be filleted.

- **boolean-merge**

  Mixins: boolean-tolerance-mixin, brep

  Description Generalized Merge container for doing boolean operations. This is not to be used directly, but is mixed into subtracted-solid, united-solid, intersected-solid, and separated-solid. The specific operation is specified in the respective subclass.

  Input slots (required):

  Other-brep GDL Brep object or object containing a brep, or list of brep objects or object containing a brep. Second (or rest) of the breps to be merged into the given `brep`

  Input slots (optional):

  Allow-multiple-regions? *Boolean*
  If set to non-nil, throw warning but not error if any of the input breps has more than one non-infinite region. Defaults to *boolean-allow-multiple-regions?*, which itself defaults to nil.

Angle-tolerance *Number*

Defaults to *angle-tolerance-radians-default*.

Brep *GDL Brep object or object containing a brep*

First of the breps to be merged

Error-on-invalid? *Boolean*

If set to non-nil, we throw an error instead of a warning if the resulting brep does not pass the built-in validation test. If nil, we throw a warning but continue to return the resulting brep. Defaults to *boolean-error-on-invalid-brep?* which itself defaults to t.

Sew-and-orient? *Boolean*

Indicates whether we should try to sew and orient the resulting brep. This defaults to t for merge operation and nil otherwise.

- **boolean-tolerance-mixin**

Mixins: vanilla-mixin

Description Implements adaptive tolerancing for certain brep objects.

Input slots (optional):

Approximation-tolerance *Number*

Defaults to *3d-approximation-tolerance-default* if non-nil. If this value is nil, then this defaults to the approximation-tolerance-adaptive.

Approximation-tolerance-adaptive *Number*

Defaults to the minimum of the adaptive-tolerance of any of the input breps, multiplied by the approximation-tolerance-factor, rounded to nearest multiple of tenths (e.g. it will be 0.01, 0.001, 0.001), however if this evaluates as zerop, *3d-approximation-tolerance-default* will be used instead.

Approximation-tolerance-factor *Number*

Defaults to *approximation-tolerance-factor*. This is multiplied by the minimum of the adaptive-tolerance of any of the input breps to produce the approximation-tolerance-adaptive.

- **box-solid**

Mixins: brep, box

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.

- **boxed-curve**

Mixins: outline-specialization-mixin, b-spline-curve

```
(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))
```

Figure 9.76: Example Code for box-solid

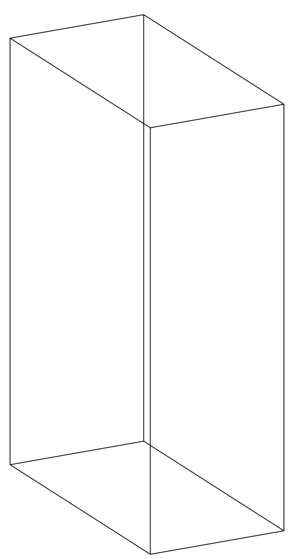

Figure 9.77: box-solid example

```
(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))))
```

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.

Input slots (required):

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+

Input slots (optional):

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.

- **boxed-surface**

  Mixins: b-spline-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.

  Input slots (required):

  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+

  Input slots (optional):

  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.

```
(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))))
```

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.

- **brep**

Mixins: geometry-kernel-object-mixin, ifs-output-mixin, base-object

Author Dave Cooper, Genworks

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.

Input slots (optional):

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*)
```

Input slots (optional, settable):

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.

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.

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.

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

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

Hidden objects (sequence):

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.

Gdl functions:

Area *Number*
Area covered by the faces of the brep.

Area-moments-of-inertia *3D Vector (i*
e. 3D Point). Returns the Area Moments of Inertia of the brep.

Area-products-of-inertia *3D Vector (i*
e. 3D Point). Returns the Area Products of Inertia of the brep.

Area-second-moment-about-coordinate-axii *3D Vector (i*
e. 3D Point). Returns the Area Second Moment About Coordinate Axii of the brep.

Area-static-moments *3D Vector (i*
e. 3D Point). Returns the Area Static Moments of the brep.

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.

Center-of-gravity *3D Point*
Center of gravity of the mass of the brep.

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.

Moments *Plist*
Returns the moments of the brep. The plist contains keys: :area-static-moments, :area-moments-of-inertia,

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,

Precise-properties-plist *Plist with keys: :area, :volume, :mass, :moments*
Returns the area, volume, mass, and moments for the brep. The moments are labeled as: :area-static-moments, :area-moments-of-inertia,

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.

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).

Volume *Number*

Volume enclosed by the brep.

Volume-moments-of-inertia *3D Vector (i*

e. 3D Point). Returns the Volume Moments of Inertia of the brep.

Volume-products-of-inertia *3D Vector (i*

e. 3D Point). Returns the Volume Products of Inertia of the brep.

Volume-second-moment-about-coordinate-axii *3D Vector (i*

e. 3D Point). Returns the Volume Second Moment about Coordinate Axii of the brep.

Volume-static-moments *3D Vector (i*

e. 3D Point). Returns the Volume Static Moments of the brep.

- **brep-intersect**

Mixins: base-object

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.

Input slots (required):

Brep *Object of type brep*

The first brep for intersecting its faces.

Other-brep *Object of type brep*

The other brep for intersecting faces.

Input slots (optional):

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.

- **cad-assembly**

  Mixins: base-object

  Description Represents a hierarchical CAD assembly for import into GDL.

  Input slots (optional):

  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 wire-frame 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.

  Imported-assembly *GDL smlib:assembly-import object*

  Normally this should be left to its default value. Defaults to the assembly-import produced from the file-name.

  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 ıorthonormal matrix, meaning each row is a vector with a magnitude of one (1.0).

  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.

- **cardinal-spline**

  Mixins: curve

  Author Dave Cooper, Genworks International

  Description This object makes a Cardinal Spline, which defaults to a Catmull-Rom Spline for nil tension-params (which means they all default to 0.0).

  Input slots (required):

```
(in-package :surf)

(define-object test-circular-curve (circular-curve)
  :computed-slots
  ((center (make-point 0 0 0)) (radius 10)))

(generate-sample-drawing :objects (make-object 'test-circular-curve))
```

Figure 9.80: Example Code for circular-curve

Through-points *List of 3D Points*
The points through which the curve will pass.

Input slots (optional):

Parameterization *Keyword symbol*
`uniform`, `:chordal` (or `:chord-length`), or `centripetal`. Default is `:uniform`.

Periodic? *Boolean*
Indicates whether the curve should close back to its start point. Default is `nil`.

- **circular-curve**

Mixins: arc-curve

Description An full circule represented exactly as a quadratic NURBS curve. Inputs and messages are the same as for arc-curve.

- **circular-surface**

Mixins: trimmed-surface, vanilla-mixin

Description Represents a planar, full-circular surface controlled by a radius, and the normal center and orientation for base-object.

Input slots (required):

Radius *Number*
The radius of the resulting trimmed-surface.

Hidden objects:

Basis-surface *GDL NURBS Surface*
The underlying surface to be trimmed.

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.

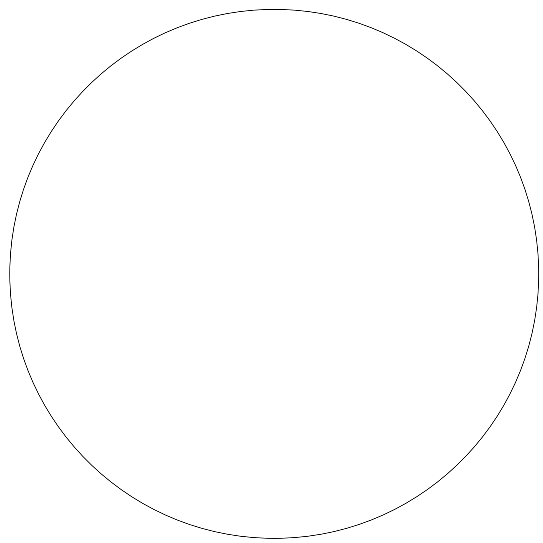

Figure 9.81: circular-curve example

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

(define-object circular-surface-test (surf::circular-surface)
 :computed-slots ((radius 108)))

(generate-sample-drawing :objects (make-object 'circular-surface-test))
```

Figure 9.82: Example Code for circular-surface

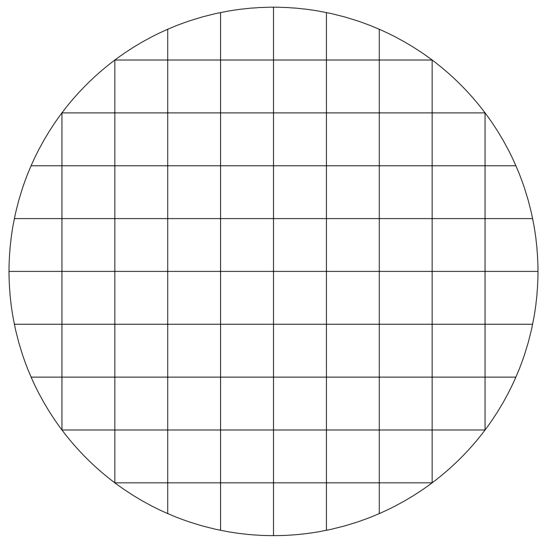

Figure 9.83: circular-surface example

- **compatible-curves**

  Mixins: base-object

  Author Dave Cooper, Genworks

  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.

  Input slots (optional):

  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.

  Objects (sequence):

  Curves *Sequence of GDL Curve objects*
  These are the resultant curves which are supposed to be compatible.

- **compatible-surfaces**

  Mixins: base-object

  Description This routine makes compatible a list of GDL surface, minimum 2 surfaces is required.

```
(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)))
```

Figure 9.84: Example Code for compatible-curves

Figure 9.85: compatible-curves example

```
(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 'compatible-surfaces-test)
                                           make-compatible-A-and-B)))
```

Figure 9.86: Example Code for compatible-surfaces

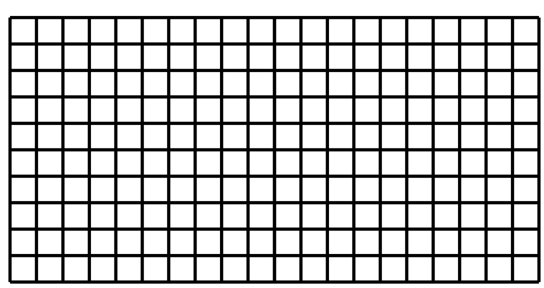

Figure 9.87: compatible-surfaces example

Input slots (required):

Surface-list *List*
A list of Gdl surface objects.

- **composed-curve**

Mixins: 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.

Input slots (required):

Curves *List of GDL curve objects*
These are the curves to be composed into a single curve.

Input slots (optional):

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.

- **composed-curves**

Mixins: base-object

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.

Input slots (required):

Curves-in *List of GDL curve objects*
These are the curves to be composed into a single curve.

Input slots (optional):

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.

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

(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))))))

(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))
```

Figure 9.88: Example Code for composed-curve

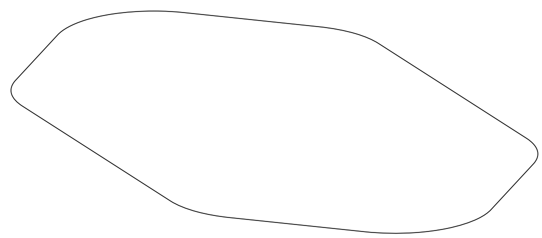

Figure 9.89: composed-curve example

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.

Objects (sequence):

Curves *Sequence of GDL Curve Objects*
The curves resulting from composition.

- **cone-solid**

  Mixins: cylinder-solid, cone

  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.

- **conic-curve**

  Mixins: curve

  Description A curve constructed from conic inputs. Inputs are two endpoints, two end tangents, and a point on the curve.

  Input slots (required):

  End-point *3D Point*
  End Point

```
(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))
```

Figure 9.90: Example Code for cone-solid

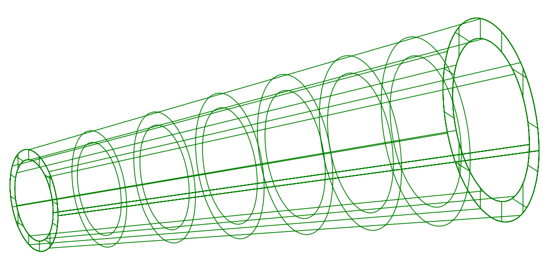

Figure 9.91: cone-solid example

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

(define-object conic-curve-test (conic-curve)
   :computed-slots
   ((start-point (make-point -10 0 0))
    (start-tangent (make-point 0.1 1 0))
    (end-point (make-point 10 0 0))
    (end-tangent (make-point -0.1 1 0))
    (point-on-curve (make-point 0 12 0))))

 (generate-sample-drawing :objects (make-object 'conic-curve-test))
```

Figure 9.92: Example Code for conic-curve

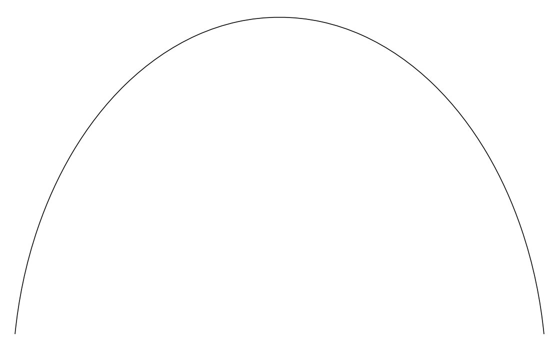

Figure 9.93: conic-curve example

```
(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)
```

Figure 9.94: Example Code for curve

End-tangent *3D Vector*
End Tangent

Point-on-curve *3D Point*
A point on the curve.

Start-point *3D Point*
Start Point

Start-tangent *3D Vector*
Start Tangent

- **curve**

  Mixins: geometry-kernel-object-mixin, base-object

  Description A generalized NURBS curve. Usually used as a mixin in more specific curves.

  Input slots (optional):

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)

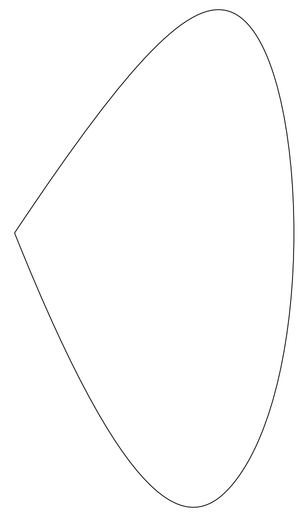

Figure 9.95: curve example

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.

Input slots (optional, settable):

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.

Gdl 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 `:distance, angle, :length`*

.

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.

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.

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.

Equi-spaced-parameters *List of Numbers*

Returns the specified number of parameters equally spaced along the curve.

Equi-spaced-points *List of 3D Points*

Returns the specified number of points equally spaced along the curve.

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.

Local-bounding-box 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.

Surface-intersection-points *List of Surface points*

Returns the point(s) of intersection between this curve and the surface given as an argument.

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.

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.

```
(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))
```

Figure 9.96: Example Code for cylinder-solid

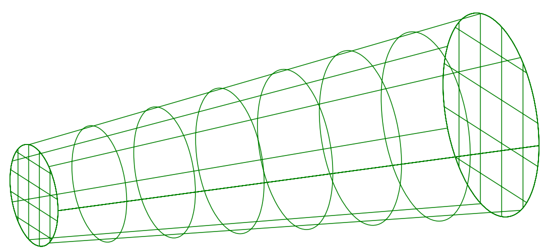

Figure 9.97: cylinder-solid example

Total-length *Number*

The total length of the curve from given start-parameter to given end-parameter.

Trim *GDL Curve object*

Returns a curve which is trimmed from parameter-1 to parameter-2.

- **cylinder-solid**

  Mixins: brep, cylinder

  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

- **decomposed-curves**

Mixins: base-object

Description Given an input curve, creates a sequence of curve segments that do not contain knots with degree-fold mutiplicity.

Input slots (required):

Curve-in *GDL Curve*
Curve (presumably multi-segment) to be decomposed.

Objects (sequence):

Curves *Sequence of GDL curve objects*
The resulting segment curves after decomposition.

- **dropped-curve**

Mixins: 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.

Input slots (required):

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.

- **dual-blend-surface**

Mixins: general-dual-blend-surface, vanilla-mixin

```
(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))))))


(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)))

(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))
```

Figure 9.98: Example Code for decomposed-curves

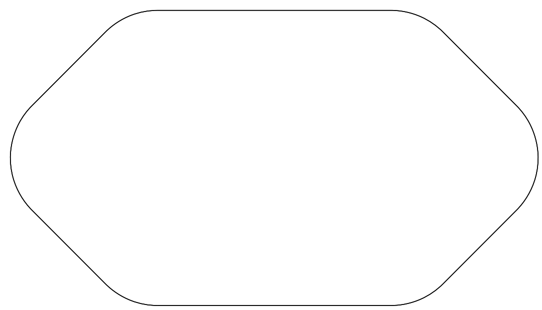

Figure 9.99: decomposed-curves example

Description Creates a smooth blend-surface between curve-1, lying on surface-1 and curve-2 on surface-2. The local start and end directions of this surface along curve-1 and curve-2 are determined from the cross-product between the tangent of these curves 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 perpendicular to the input-curves. Takes the same inputs as general-dual-blend-surface, except for f-tangent-1 and f-tangent-2.

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.

- **edge**

Mixins: curve

Description Represents one edge in a brep. This object is a type of curve and answers all curve messages.

```
(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)
```

Figure 9.100: Example Code for dropped-curve

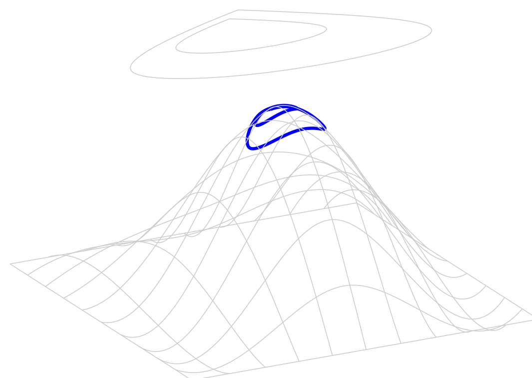

Figure 9.101: dropped-curve example

Computed slots:

Faces *List of GDL Face objects*
The faces connected to this edge.

Gdl 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.

- **edge-blend-surface**

Mixins: general-dual-blend-surface, vanilla-mixin

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

```
(in-package :surf)

(define-object test-d-b-s (base-object)

 :objects
 ((d-b-s :type 'dual-blend-surface
         :display-controls (list :color :green)
         :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-d-b-s))
                         :projection-direction (getf *standard-views* :top))
```

Figure 9.102: Example Code for surf::dual-blend-surface

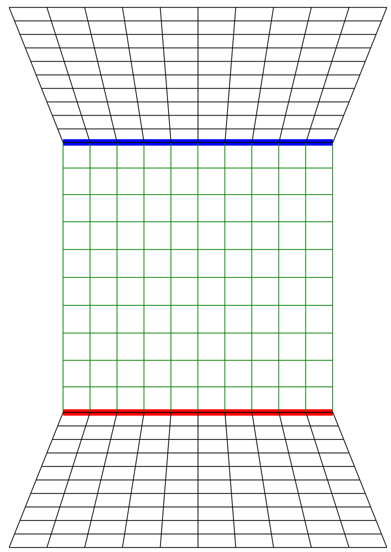

Figure 9.103: surf::dual-blend-surface example

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.

Input slots (optional):

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.

```
(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))
```

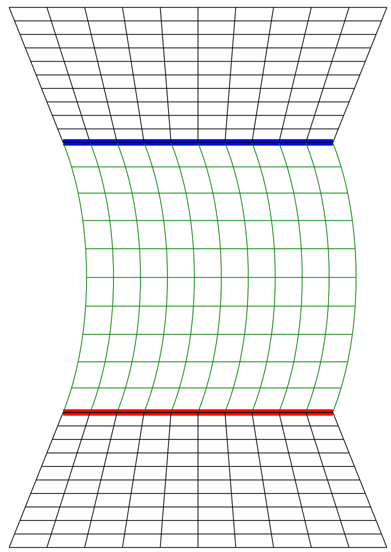

Figure 9.105: edge-blend-surface example

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.

- **elliptical-curve**

Mixins: curve, ellipse

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.

- **extended-curve**

Mixins: curve

Description Creates an extended curve extending its start or end (`u1` and `u2`).

Input slots (required):

Curve-in *GDL Curve Object*

The underlying curve from which to build this curve.

Input slots (optional):

Continuity *Keyword*

```
(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))
```

Figure 9.106: Example Code for elliptical-curve

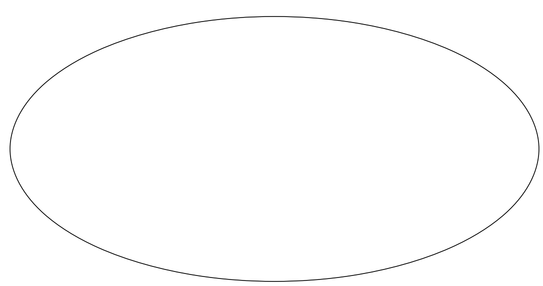

Figure 9.107: elliptical-curve example

```
(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))
```

Figure 9.108: Example Code for extended-curve

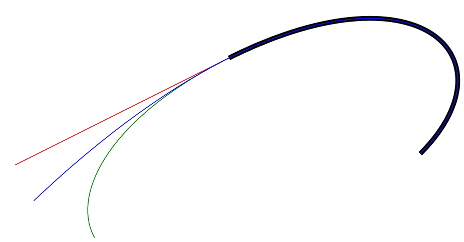

Figure 9.109: extended-curve example

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`.

- **extended-surface**

Mixins: 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.

```
(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)
```

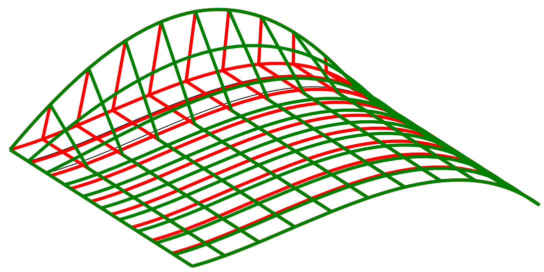

Figure 9.111: extended-surface example

Input slots (required):

Curve *GDL curve object*

The curve to which the surface should be extended.

Surface *GDL surface object*

The surface to be extended.

Input slots (optional):

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.

- **extruded-solid**

  Mixins: brep

  Description Generates a brep by extruding a profile along an axis vector.

  Input slots (required):

  Profile *GDL Curve object*
  The profile to be extruded into a solid.

  Input slots (optional):

  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).

- **face**

  Mixins: surface

  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.

  Input slots (required):

  Brep *GDL Brep object*
  This is the brep object which contains this face object.

  Input slots (optional):

  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.

Hidden objects (sequence):

Edges *Sequence of GDL Edge Objects*

The Edges contained within this brep.

Gdl functions:

Area *Number*

Returns the area of the face.

Area-moments-of-inertia *3D Vector (ie 3D Point)*

Returns the Area Moments of Inertia of the face.

Area-products-of-inertia *3D Vector (ie 3D Point)*

Returns the Area Products of Inertia of the face.

Area-second-moment-about-coordinate-axii *3D Vector (ie 3D Point)*

Returns the Area Second Moment About Coordinate Axii of the face.

Area-static-moments *3D Vector (ie 3D Point)*

Returns the Area Static Moments of the face.

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.

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.

Volume *Number*

Returns the volume of the face.

Volume-moments-of-inertia *3D Vector (ie 3D Point)*

Returns the Volume Moments of Inertia of the face.

Volume-products-of-inertia *3D Vector (ie 3D Point)*

Returns the Volume Products of Inertia of the face.

```
(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)))))
```

Figure 9.112: Example Code for fitted-curve

Volume-second-moment-about-coordinate-axii *3D Vector (ie 3D Point)*
Returns the Volume Second Moment about Coordinate Axii of the face.

Volume-static-moments *3D Vector (ie 3D Point)*
Returns the Volume Static Moments of the face.

- **fitted-curve**

Mixins: curve

Description Fits a curve through a set of points with given degree and parameterization.

Input slots (required):

Points *List of 3D Points*
The points for fitting.

Input slots (optional):

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 .

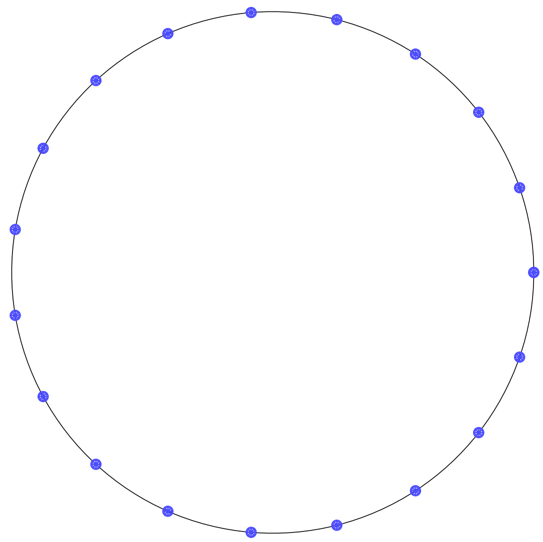

Figure 9.113: fitted-curve example

Local? *Boolean*

Indicates whether the inputted control-points should be considered in local coordinate system of this object. Default is nil.

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 `:tangents`, `:normals`, or `:first-last`*

`: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.

- **fitted-surface**

<u>**Mixins:**</u> 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.

Input slots (required):

Points *List of lists of 3D Points*

The points for fitting, with inner lists representing U direction and outer lists V direction.

Input slots (optional):

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.

```
(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))))))


(#+allegro excl:without-package-locks #-allegro progn
 (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))
```

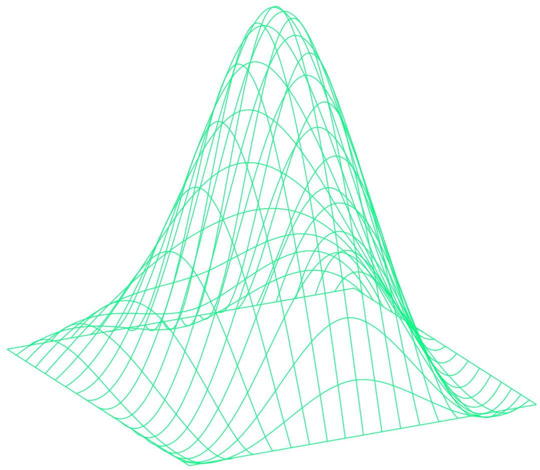

Figure 9.115: fitted-surface example

- **general-dual-blend-surface**

  Mixins: b-spline-surface

  Description Creates a smooth blend-surface between curve-1, lying on surface-1 and curve-2 on surface-2. The local start and end directions of this surface along curve-1 and curve-2 are specified by the user with parametric functions.

  Input slots (required):

Curve-1 *GDL curve*

Curve lying on surface-1 that forms the starting-edge of the blend-surface

Curve-2 *GDL curve*

Curve lying on surface-2 that forms the ending-edge of the blend-surface

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.

Surface-1 *GDL surface*

Surface that is to be connected to surface-2 by the blend-surface

```
(in-package :surf)

(define-object test-g-d-b-s (base-object)

 :computed-slots
 ((f-tangent-1 #'(lambda(param)
                   (rotate-vector-d
                    (the (face-normal-vector :top))
                    -10
                    (the curve-1 (tangent param)))))


  (f-tangent-2 #'(lambda(param)
                   (rotate-vector-d
                    (the (face-normal-vector :bottom))
                    10
                    (the curve-2 (tangent param)))))

  (f-ratio-1 #'(lambda(param)
                 (declare (ignore param))
                 1/3))

  (f-ratio-2 #'(lambda(param)
                 (declare (ignore param))
                 1/3)))

 :objects
 ((g-d-b-s :type 'general-dual-blend-surface
           :display-controls (list :color :green)
           :pass-down (f-tangent-1
                        f-tangent-2 f-ratio-1 f-ratio-2
                        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))
```

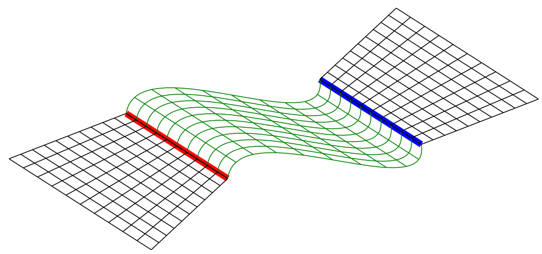

Figure 9.117: surf::general-dual-blend-surface example

Surface-2 *GDL surface*

Surface that is to be connected to surface-1 by the blend-surface

Input slots (optional):

F-ratio-1 *Input-function*

Parametric function defined from 0 to 1 that outputs the blend-surface's local blend-ratio along curve-1. The input value of 0 corresponds to the start of curve-1, 1 to the end of curve-1. The output local blend-ratio can be any number greater than 0, but the resulting blend surface can become erratic for values greater than 1. Defaults to a constant value of 1/3.

F-ratio-2 *Input-function*

Parametric function defined from 0 to 1 that outputs the blend-surface's local blend-ratio along curve-2. The input value of 0 corresponds to the start of curve-2, 1 to the end of curve-2. The output local blend-ratio can be any number greater than 0, but the resulting blend surface can become erratic for values greater than 1. Defaults to a constant value of 1/3.

N-segments *Number*

Number of b-spline-segments of the resulting blend-surface in the parametric direction along curve-1 and curve-2. Defaults to 50.

Computed slots:

Control-points *List of lists of 3D Points*

The control net.

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.

- **geometry-kernel-object-mixin**

Mixins: vanilla-mixin

Description This mixin provides messages common to all NURBS-based objects whose underlying implementation comes from a modular geometry kernel such as SMLib.

Input slots (optional):

Color-decimal *Vector of three real numbers*

The RGB color of this object as imported from an external format (e.g. IGES) or as specified in :display-controls. Defaults to the foreground color specified in `*colors-default*`. This message should not normally be overridden in user application code.

Iges-level *Integer*

Synonym for the layer.

Layer *Integer*

The primary IGES-compatible level (layer) on which this object resides. Defaults to the first of the levels. This slot can be overridden in user code to specify a new layer which will be written out when this object is exported with the IGES output-format.

Levels *List of integers*

The IGES-compatible levels (layers) on which this object resides. GDL does not currently support writing out multiple levels (layers) through the IGES writer ; only the first of these will be output if the object is exported with the IGES output-format (please contact Genworks if you need all levels (layers) to be written out).

- **global-filleted-polyline-curve**

Mixins: global-filleted-polyline-curves, composed-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.

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

(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))))))

(generate-sample-drawing :objects (the-object (make-object 'test-global-filleted-polyline-curves))
                         :projection-direction (getf *standard-views* :trimetric))
```

Figure 9.118: Example Code for global-filleted-polyline-curves

- **global-filleted-polyline-curves**

  Mixins: global-filleted-polyline

  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.

  Hidden objects (sequence):

  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.

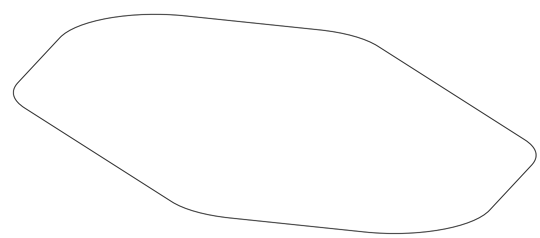

Figure 9.119: global-filleted-polyline-curves example

- **iges-reader**

  Mixins: base-object

  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.

  Input slots (required):

  File-name *String or pathname*

  The location of the IGES file to be read.

  Input slots (optional):

  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?).

Objects (sequence):

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.

- **intersected-solid**

Mixins: boolean-merge, vanilla-mixin

Description Given two brep solids, performs the intersect Boolean between the brep and the other-brep

- **iso-curve**

Mixins: curve

Description Represents an exact iso curve on the given surface in given direction at given parameter value.

Input slots (required):

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.

Input slots (optional):

```
(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))
```

Figure 9.120: Example Code for iso-curve

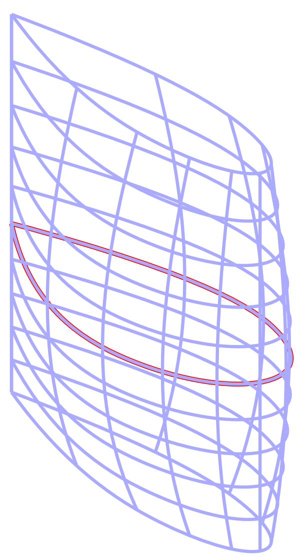

Figure 9.121: iso-curve example

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.

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

- **joined-surfaces**

Mixins: surface

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

Input slots (required):

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.

Input slots (optional):

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*

- **linear-curve**

Mixins: %linear-curve%, vanilla-mixin

```
(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)))
```

Figure 9.122: Example Code for joined-surfaces

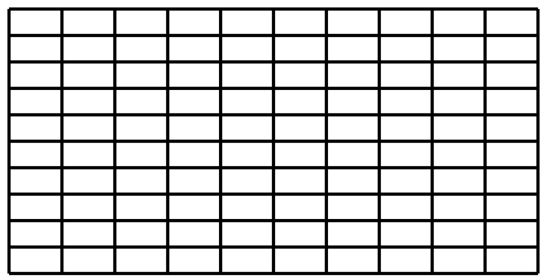

Figure 9.123: joined-surfaces example

```
(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))
```

Figure 9.124: Example Code for linear-curve

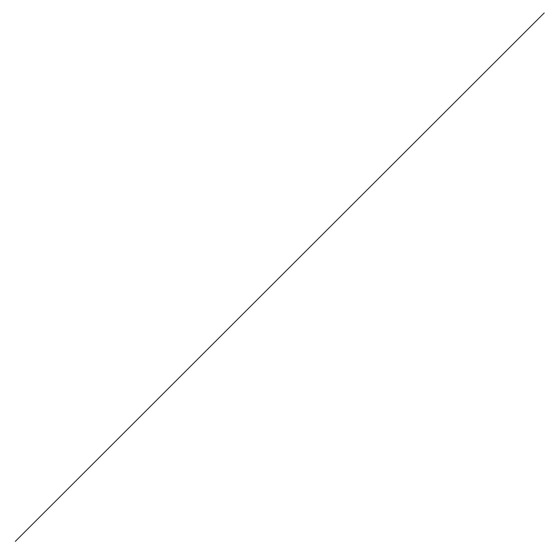

Figure 9.125: linear-curve example

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).

Input slots (optional):

End *3D Point*
The end point of the line, in global coordinates.

Start *3D Point*
The start point of the line, in global coordinates.

- **lofted-surface**

Mixins: surface

Description Loft of a surface through a list of curves with various controls.

Input slots (required):

Curves *List of GDL Curve objects*
The curves through which the surface will be lofted.

Input slots (optional):

Rail-1 *GDL Curve object*
Guide rail corresponding to minimum U parameter of resulting surface. Defaults to nil.

```
(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)
```

Figure 9.126: Example Code for lofted-surface

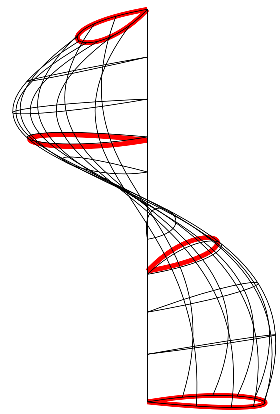

Figure 9.127: lofted-surface example

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.

- **manifold-solid**

  Mixins: brep

  Description Attempts to take in a non-manifold brep and yield a manifold version.

  Input slots (required):

  Brep *GDL brep object*

  The brep to be represented as a manifold brep in this instance.

  Input slots (optional):

  Keep-internal-faces? *Boolean*

  Indicates whether faces between two non-void regions should be kept. Defaults to nil.

- **merged-solid**

  Mixins: boolean-merge, vanilla-mixin

  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.

  Input slots (optional):

  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.

- **native-reader**

  Mixins: base-object

  Description This object will reflect the contents of an iwp file containing curves, surfaces, breps, and brep trees as sequences of GDL objects.

  Input slots (optional):

  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.

Objects (sequence):

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.

- **non-rational-curve**

Mixins: curve

Author Dave Cooper, Genworks International

Description This object accepts a rational curve and approximates it with a non-rational curve.

Input slots (required):

Curve-in *GDL Curve object*

Presumably this is a Rational curve (else this object will do nothing).

Input slots (optional):

Maintain-end-tangents? *Boolean*

Determines whether to try to maintain tangents at the ends. Defaults to t.

Non-rational-degree *Integer*

Determines the degree of the non-rational curve. Defaults to 3.

Parameterization *Keyword symbol, one of :uniform, :chord-length, :centripetal, or :inherited*

The default is :inherited.

Tolerance *Number*

The tolerance to use for non-rational approximation of a rational curve-in. Defaults to the curve-in's total length divided by the tolerance-divisor.

Tolerance-divisor *Number*

The amount by which to divide the total-length of the curve-in to compute the default tolerance. Default is 1000.

- **normalized-curve**

  Mixins: 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.

  Input slots (optional):

  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

- **offset-solid**

  Mixins: brep

  Description Offsets the faces of a brep by a certain distance.

  Input slots (required):

  Brep *GDL Brep object*
  The brep to be offset.

  Distance *Number*
  The distance to offset. Can be negative.

  Input slots (optional):

  Tolerance *Number*
  The tolerance to use for the shelling operation. Defaults to (the adaptive-tolerance) of the input brep.

- **offset-surface**

  Mixins: surface

  Description This primitive generates a new surface which is offset from input surface in all normal directions from input surface.

  Input slots (required):

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.

Input slots (optional):

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.

- **planar-offset-curve**

Mixins: curve

Description Creates a curve which is the result of offsetting a curve by its normals along a plane.

Input slots (required):

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

Input slots (optional):

Tolerance *Number*

The tolerance for approximating the resulting offset curve. Defaults to *3d-approximation-tolerance-default*.

```
(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))
```

Figure 9.128: Example Code for planar-offset-curve

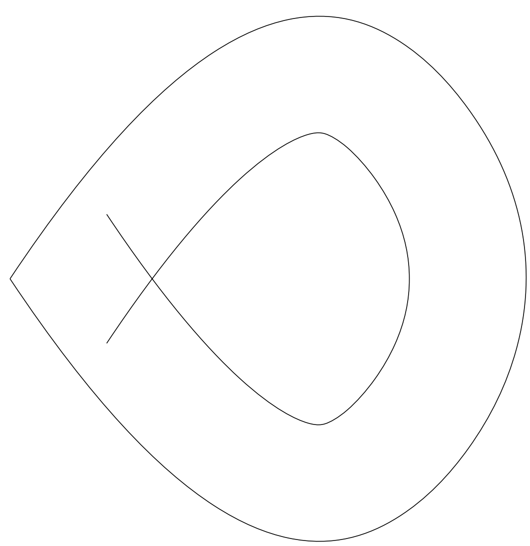

Figure 9.129: planar-offset-curve example

```
(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))
```

Figure 9.130: Example Code for planar-section-curve

- **planar-section-curve**

  Mixins: curve

  Description Produces a single curve by sectioning a surface with a plane. If multiple results are expected, use planar-section-curves instead.

  Input slots (required):

  Surface *GDL Surface, face, or trimmed surface*
  The surface to be sectioned with a plane.

  Input slots (optional):

  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:

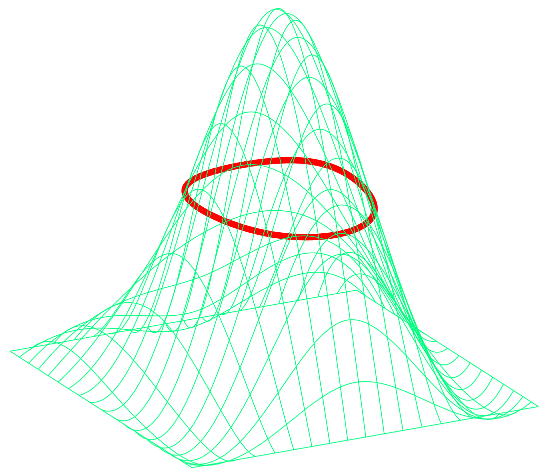

Figure 9.131: planar-section-curve example

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.

Gdl functions:

Uv-curve *GDL Curve object*

The UV curve for this curve in the context of the surface.

- **planar-section-curves**

Mixins: base-object

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.

Input slots (optional):

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*.

```
(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))
```

Figure 9.132: Example Code for planar-section-curves

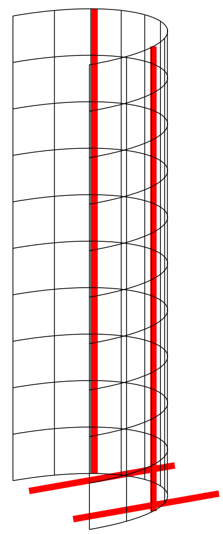

Figure 9.133: planar-section-curves example

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.

Objects (sequence):

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.

```
(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)
```

Figure 9.134: Example Code for planar-surface

- **planar-surface**

  Mixins: surface

  Description Creates a flat quadrilateral surface specified by its four corner points.

  Input slots (required):

  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.

- **poly-brep**

  Mixins: geometry-kernel-object-mixin, base-object

  Description A polygonal representation for a boundary representation geometric entity.

  Computed slots:

  Mesh-data *Plist*
  Contains mesh data from the poly brep.

- **projected-curve**

  Mixins: curve

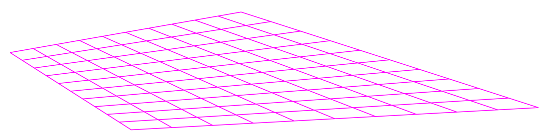

Figure 9.135: planar-surface example

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.

Input slots (required):

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.

Input slots (optional):

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.

```
(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)
```

Figure 9.136: Example Code for projected-curve

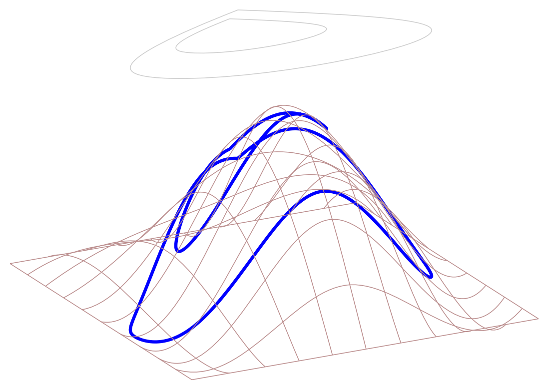

Figure 9.137: projected-curve example

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.

- **rectangular-surface**

Mixins: planar-surface

Description Creates a flat rectangular surface specified by the same inputs as box or base-object.

Input slots (required):

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.

```
(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)
```

Figure 9.138: Example Code for rectangular-surface

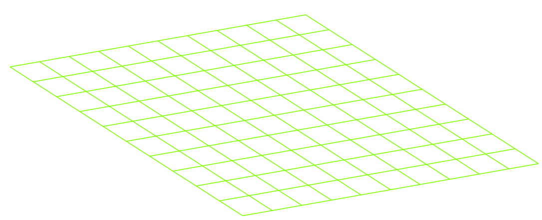

Figure 9.139: rectangular-surface example

Input slots (optional, defaulting):

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.

- **redirected-surface**

Mixins: b-spline-surface

Description Generates a new surface with U and V parameters swapped and/or reversed in their direction.

Input slots (required):

Surface-in GDL surface object

Input slots (optional):

Reverse-u? *Boolean*
Specifies whether resulting U parameter space should be reversed. Defaults to nil.

Reverse-v? *Boolean*
Specifies whether resulting V parameter space should be reversed. Defaults to nil.

Swap-uv? *Boolean*
Specifies whether existing U and V should be swapped before possibly reversing. Defaults to nil.

Computed slots:

Control-points *List of lists of 3D Points*
The control net.

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.

- **regioned-solid**

Mixins: base-object

Description Given a brep solid that contains multiple regions, splits the regions into separate breps

Input slots (required):

Brep *GDL Brep object or object containing a brep*

The multi-region brep to be split.

Input slots (optional):

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.

- **revolved-surface**

Mixins: brep, surface

Description Creates a surface of revolution based on an arbitrary NURBS curve revolved by some angle about a central axis and axis point.

Input slots (required):

```
(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)
```

Figure 9.140: Example Code for revolved-surface

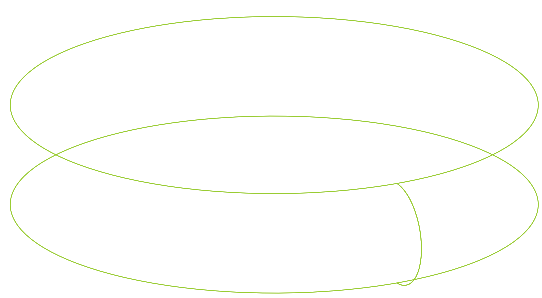

Figure 9.141: revolved-surface example

Curve *GDL Curve object*
The profile to be revolved.

Input slots (optional):

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.

Input slots (optional, defaulting):

Center *3D Point*
Indicates in global coordinates where the center of the reference box of this object should be located.

- **revolved-surfaces**

Mixins: base-object

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.

Input slots (required):

Curves *List of GDL Curve objects*
The profiles to be revolved.

Input slots (optional):

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.

Objects (sequence):

Surfaces *Sequence of GDL Surfaces*
The resultant revolved surfaces.

- **ruled-surface**

Mixins: surface

```
(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)
```

Figure 9.142: Example Code for revolved-surfaces

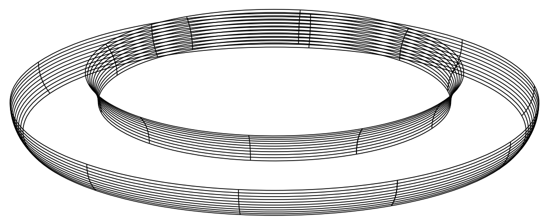

Figure 9.143: revolved-surfaces example

```
(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)
```

Figure 9.144: Example Code for ruled-surface

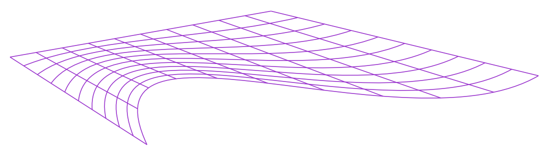

Figure 9.145: ruled-surface example

Description Creates a surface between two NURBS curves.:

Input slots (required):

Curve-1 *GDL Curve object*
First boundary of the ruled surface.

Curve-2 *GDL Curve object*
Second boundary of the ruled surface.

Input slots (optional):

Direction *Keyword symbol, either :u or :v*
The direction of parameterization of the surface between the two curves.

- **separated-solid**

Mixins: boolean-merge, vanilla-mixin

Description Given two brep solids or a brep solid and an open face represented as a brep, performs a split operation

Input slots (optional):

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.

Objects (sequence):

Breps *Sequence of GDL brep objects*
The resulting breps yielded from the separate operation. These are colored using section-colors.

- **shelled-solid**

Mixins: brep

Description Offsets the faces of a brep by a certain thickness.

Input slots (required):

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.

Input slots (optional):

Tolerance *Number*
The tolerance to use for the shelling operation. Defaults to (the adaptive-tolerance) of the input brep.

- **spherical-surface**

Mixins: surface, sphere

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.

Input slots (optional, defaulting):

```
(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)
```

Figure 9.146: Example Code for spherical-surface

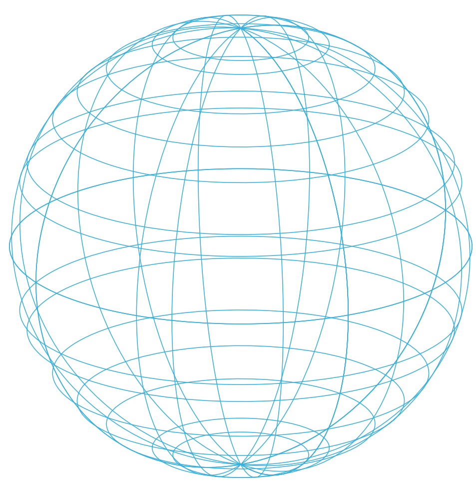

Figure 9.147: spherical-surface example

Center *3D Point*

Indicates in global coordinates where the center of the reference box of this object should be located.

- **spiral-curve**

Mixins: curve

Author Dave Cooper, Genworks International

Description This object approximates a spiral around the Z axis using a cubic NURBS (b-spline-curve).

Input slots (required):

Height *Number*

The height of the spiral.

Number-of-turns *Number*

The number of turns (1 = 360 degrees) in the spiral.

Radius-1 *Number*

Initial radius at Z=0.

Radius-2 *Number*

Final radius at z=height.

Input slots (optional):

Right-or-left *Keyword Symbol, :right or :left*

Defaults to :right.

Tolerance *Number*

The tolerance to use for non-rational approximation of a rational curve-in. Defaults to the height divided by the tolerance-divisor.

Tolerance-divisor *Number*

The amount by which to divide the height of the curve-in to compute the default tolerance. Default is 1000.

- **split-surface**

Mixins: 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.

```
(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))
```

Figure 9.148: Example Code for split-surface

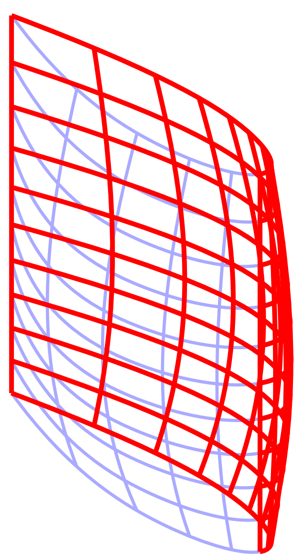

Figure 9.149: split-surface example

Input slots (optional):

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.

Projected-points *List of Surface points*

Returns the given point projected along the given vector intersected with the surface.

- **step-reader**

Mixins: base-object

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.

Input slots (required):

File-name *String or pathname*
The location of the STEP file to be read.

Input slots (optional):

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?).

Objects (sequence):

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.

- **stitched-solid**

  Mixins: brep

  Description Attempts to stitch together a list of surfaces into an open shell or possibly a manifold solid. This primitive is not widely tested and may be deprecated. Consider use of merged-solid instead. Please let Genworks know if you are using this so we can consider documenting/testing it better.

  Input slots (required):

Faces-in *List of GDL Surface or Face objects*

These will be stitched together into an open shell or possibly a Solid

Input slots (optional):

Tolerance *Number*

The tolerance to use for creating the brep and for sewing. Larger number is looser tolerance and more likely to lead to success. Default is 0.0 which uses the SMLib defaults.

- **subtracted-solid**

  Mixins: boolean-merge, vanilla-mixin

  Description Given two brep solids, performs the subtract Boolean of the other-brep from the brep

- **surface**

  Mixins: geometry-kernel-object-mixin, outline-specialization-mixin, base-object

  Description A generalized NURBS surface. Usually used as a mixin in more specific surfaces.

  Input slots (optional):

  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 the brep of this surface. 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.

Input slots (optional, defaulting):

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).

Hidden objects (sequence):

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.

Gdl functions:

Area *Number*

Returns the area of the surface.

B-spline-data 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.

Curve-intersection-points *List of Surface points*

Returns the point(s) of intersection between this surface and the curve given as an argument.

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.

Dropped-points *List of Surface points*

Returns the given 3D point dropped normally to this surface.

Local-bounding-box 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.

- **surface-knot-reduction**

  Mixins: surface

  Description This routine removes all removable knots from a GDL surface.

  Input slots (optional):

  Direction *Keyword symbol, one of :u, :v or :uv*
  Default is :uv.

  Surface *Gdl surface object*
  .

  Tolerance *Number*
  .

- **swept-solid**

  Mixins: brep

  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.

  Input slots (required):

  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.

- **transformed-solid**

  Mixins: brep

  Description This primitive Translates, Orients, and optionally Scales a brep solid into another brep solid.

  Input slots (required):

  Brep *GDL Brep Object*
  Source Brep to be copied and transformed.

  Input slots (optional):

  From-location *3D Point*
  Reference location from which to translate. Defaults to the from-object center.

```
(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))))
```

```
(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))
```

Figure 9.151: Example Code for swept-solid

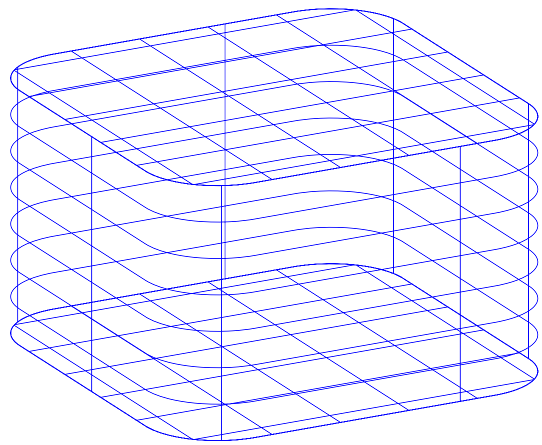

Figure 9.152: swept-solid example

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.

- **trimmed-curve**

Mixins: curve

Description Creates a curve based on an existing curve but possibly with new start and end parameters (`u1` and `u2`).

Input slots (required):

```
(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))
```

Figure 9.153: Example Code for trimmed-curve

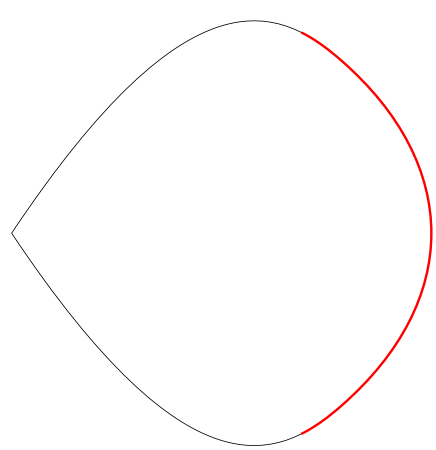

Figure 9.154: trimmed-curve example

Built-from *GDL Curve Object*

The underlying curve from which to build this curve.

Input slots (optional):

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.`

- **trimmed-surface**

Mixins: face

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.

Input slots (optional):

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.

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.

Reverse-holes? *Boolean*

Specify this as non-NIL if the holes are given in counter-clockwise orientation. Default is NIL.

```
(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)
```

Figure 9.155: Example Code for trimmed-surface

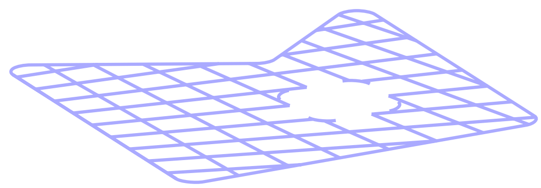

Figure 9.156: trimmed-surface example

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.

- **united-solid**

Mixins: boolean-merge, vanilla-mixin

Description Given two brep solids, performs the union Boolean between the brep and the other-brep

- **uv-iso-curve**

  Mixins: vanilla-mixin

  Description Convenience object to return UV parameter values on surface.

  Gdl functions:

  Point *2D point*

  The UV surface representation at the given parameter value.

9.20.2 Function and Macro Definitions

- **get-point-on-curve**

  (type 3D point intro Returns the point found from a minimum or maximum distance computation involving a curve.)

- **get-point-on-other-curve**

  (type 3D point intro Returns the other point found from a minimum or maximum distance computation involving a curve.)

- **get-point-on-surface**

  (type 3D point intro Returns the point found from a minimum or maximum distance computation involving a surface.)

- **with-pinned-values [Macro]**

  (type Macro intro Takes a list of bindings and body similar to let, but each binding contains a variable, which must already exist in this context, followed by a min and a max value to which its inner binding will be pinned. If the value of the outer binding falls outside the range, a warning will be issued.)

9.20.3 Variables and Constants

- ***3d-approximation-tolerance-default***
- ***3d-tolerance-default***
- ***angle-tolerance-radians-default***
- ***approximation-tolerance-factor***
- ***boolean-allow-multiple-regions?***
- ***boolean-error-on-invalid-brep?***
- ***boolean-operation-tolerance-default***

- ***brep-isos-default***
- ***brep-tolerance-default***
- ***chain-beziers-for-display?***
- ***crease-angle-default***
- ***display-tolerance***
- ***finalize-lofted-surfaces?***
- ***include-vrml-normals?***
- ***isos-default***
- ***output-units-default***
- ***separate-brep-faces-for-vrml?***
- ***tess-max-3d-edge-factor***
- ***tess-max-angle-degrees***
- ***tess-max-aspect-ratio***
- ***tess-max-chord-height***
- ***tess-min-3d-edge***
- ***tess-min-edge-ratio-uv***
- ***tess-min-number-of-segments***
- ***tess-min-parametric-ratio***

9.21 tasty (Web-based Development Environment (tasty))

9.21.1 Variables and Constants

- ***suppress-$$-messages?***
- ***suppress-%%-messages?***

9.22 tree (Tree component used by Tasty and potentially as a UI component on its own)

9.22.1 Object Definitions

- **newertree**

  Mixins: sheet-section

Description Implements an interactive graphical tree from a nested list using HTML list element and CSS.

Input slots (optional):

Onclick-function *Function of one argument*
This function takes a node in the tree as an argument, and should return a plist with keys :function and :arguments, which is a function in the bashee which will be called with the given arguments when the given node in the tree is clicked.

Respondent *GDL Object*
Object to respond to the form submission. Defaults to self.

Computed slots:

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).]

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

- **tree**

Mixins: sheet-section, tree-node-mixin

Description Implements an interactive grapical tree using HTML table elements.

Input slots (optional):

Button-color

Onclick-function *Function of one argument*
This function takes a node in the tree as an argument, and should return a plist with keys :function and :arguments, which is a function in the bashee which will be called with the given arguments when the given node in the tree is clicked.

Respondent *GDL Object*
Object to respond to the form submission. Defaults to self.

Tree-color String

9.23 yadd (Yet Another Definition Documenter (yadd))

9.23.1 Object Definitions

- **assembly**

Mixins: base-yadd-sheet

Author Dave Cooper (Genworks)

Description "Yet Another Definition Documenter." Generates documentation for all the relevant packages in the current Lisp session. Presents a standard :write-html-sheet method which can also be crawled with a call to

```
(gwl:crawl "yadd:assembly")
```

The packages to be documented, and whether the green/red supported messages flags show up, can be controlled with optional-inputs.

Input slots (optional):

External-only? *Boolean*

This defaults to nil, if it is set to t, only exported symbols will be considered for documentation.

Packages-to-ignore *List of keyword symbols*

These packages will be ignored. This list defaults to standard internal and test packages

Computed slots:

Title *String*

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

Objects:

Master-index *index*

Master index of all symbols (objects, functions, parameters, variables, constants)

Objects (sequence):

Package-dokumentations *package-dokumentation*

Quantified, one for each :package-to-document

Gdl functions:

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.

- **base-yadd-sheet**

  Mixins: base-ajax-sheet

Author Dave Cooper (Genworks)

Description Base mixin for a yadd sheet

Computed 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 *String of valid HTML*

Contains standard jQuery files to include in the header for additional search funcionality. This computed-slot contains javascript files, found in the *gdl-install-dir* and used throughout the yadd pages for the generation of automatic search forms (like the master-index). The javascript loaded is jquery.

Default-header-content *String of valid HTML*

Contains default header contents for yadd html files. This computed-slot is available in all children of this object. It contains links to default header content of a HTML generated yadd page. This contains a link to the favicon.ico and a link to a default CSS sheet. All these elements can be found in the *gdl-install-dir*/static/gwl/ directories.

- **master-index**

Mixins: base-yadd-sheet

Author Dave Cooper (Genworks)

Description Prints bullet list of symbols as links to their documentation pages.

Input slots (required):

Symbols-for-index *List of lists*

Each list contains the page object for the symbol's documentation and the symbol's print-name. The list should be sorted based on the symbols' print-names.

Computed slots:

Additional-header-js-content *valid javascript*

This javascript is added to the head of the page, just before the body.

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.

Use-jquery? *Boolean*

Include jquery javascript libraries in the page header? Default nil.

- **package-dokumentation**

  Mixins: base-yadd-sheet

  Author Dave Cooper

  Description Prepares documentation for all relevant symbols in a given Lisp package.

  Input slots (optional):

  External-only? *Boolean*

  Determines whether to consider all symbols in the package or just the exported ones.

  Package *String or keyword symbol*

  Names the package, or a nickname of the package, to be documented.

  Show-supported-flag *boolean*

  Determines whether to show red/green flag on each message indicating whether it is a supported message.

  Computed slots:

  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.

  Title *String*

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

  Objects:

  Function-docs *function-doc*

  Container for set of all Function documentation sheets.

  Object-docs *object-doc*

  Container for set of all Object documentation sheets.

  Variable-docs *variable-doc*

  Container for set of all Parameter/Variable/Constant documentation sheets.

  Hidden objects:

  Package-form *package-form*

  Allows user to modify toplevel optional-inputs.

  Gdl functions:

  Dom-section *List in GDL dom authoring format*

  Suitable for filling in a section of output document.

Write-html-sheet *Void*

Prints to *html-stream* a bulleted list for each of the three categories of docs in the package.

- **package-form**

Mixins: base-yadd-sheet

Author Dave Cooper (Genworks)

Description Presents a form to the user to be able to modify the Package, supported-flag, and external flag.

Gdl functions:

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:

Bibliography

[1] G. LaRocca *Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design* Advanced Engineering Informatics 26 (2012) 159-179, Elsevier.

Index