Genworks GDL: A User's Manual

Dave Cooper

July 2013

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 has computed something, it generally will not need to

[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 that data, 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

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*, so as 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 accordingly 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 whole 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 the 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 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 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 state modification 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 standard specification, 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."

- 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

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

that GDL is a superset of Common Lisp, you of course still have 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 as contrasted to 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 taken care of for you 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 too. Do not be concerned at this point if you are not fully conversant of 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 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;*
- 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.

[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 Lispalso covers the topic; see http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html.

1.13 What GDL is not

- A CAD system (although it may operate on and/or generate geometric entities);
- 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

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 want 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 very 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!)`

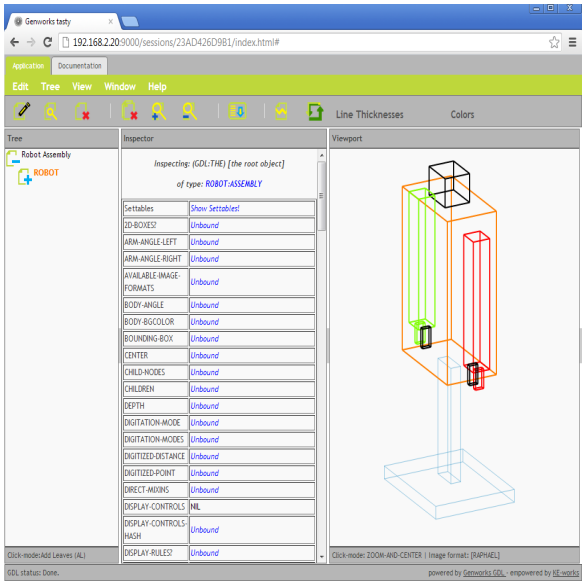


Figure 2.1: Robot displayed in Tasty

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).

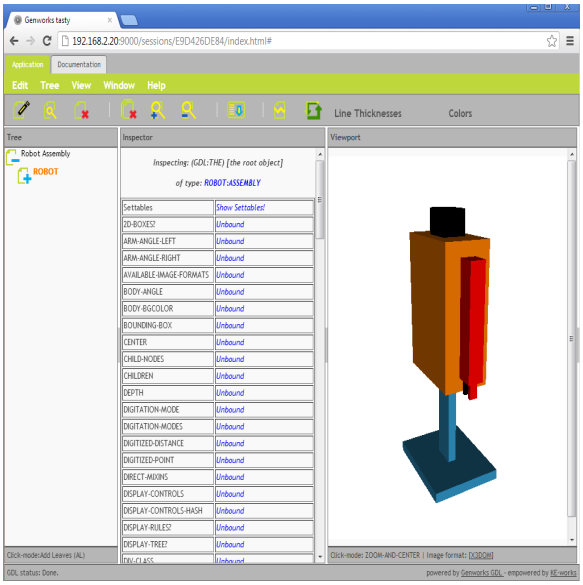


Figure 2.2: Robot x3dom

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 timeframe 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 go into particular depth about the additional features of the environment or language syntax in this section — this chapter is merely for getting familiar with and 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 constantly 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).

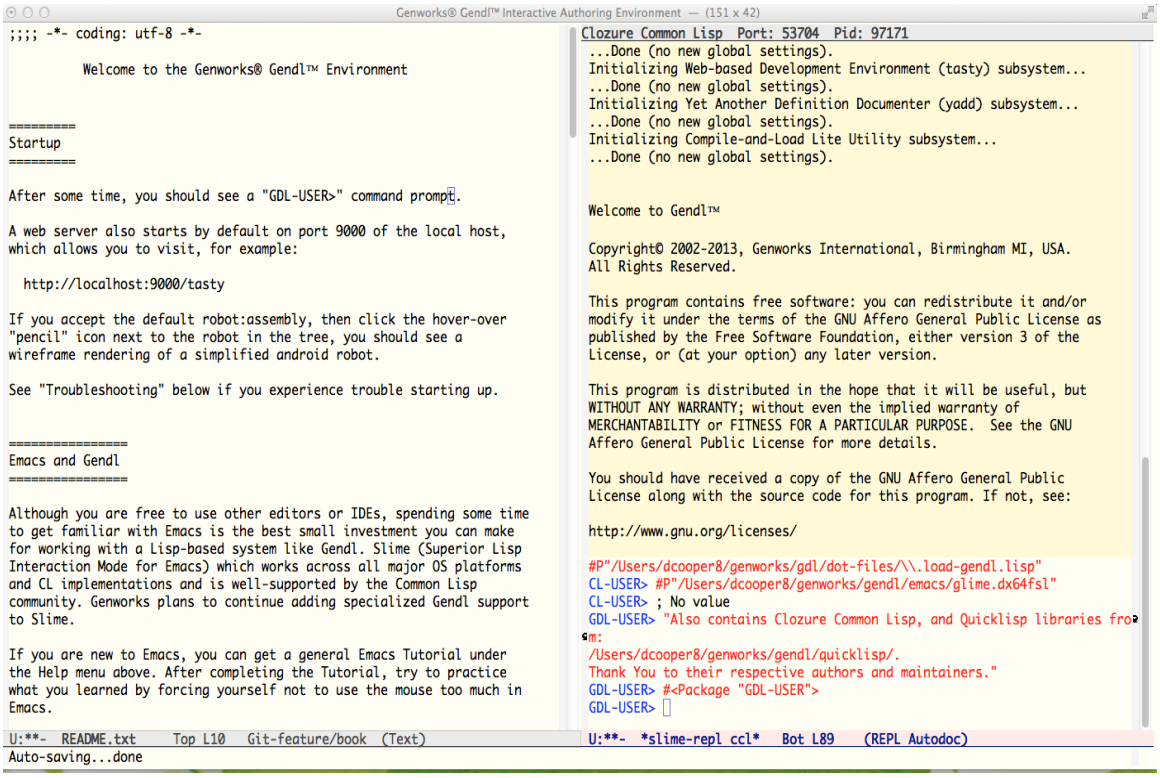


Figure 3.1: Startup of Emacs with GDL

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 give 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 on 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 looks like 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*)`

   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 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. For example:

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

2. `(ff:unload-foreign-library (merge-pathnames "smlib.dll" "sys:smlib;"))`
3. `(net.aserve:shutdown)`
4. `(setq excl:*restart-init-function* '(gdl:start-gdl :edition :trial))`
5. (to save an image named yoyodyne.dxl) Invoke the command

   ```
   (dumplisp :name "yoyodyne.dxl")
   ```

   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 `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. Unlike in a spreadsheet, however, 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.

[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.

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 shows, `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 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 occur in an expression. As we have seen, if a *symbol* appears first in a list expression, as with the `+` in `(+ 2 2)`, the symbol is evaluated for its function slot. If the first element of an expression indeed has 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 subexpressions.

4.2.4 List Basics

Lisp takes 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?

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 cover 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`.

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 see how one may alter the value of a variable such as `my-slides`.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 really 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 `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. Much as with any language, practice (that is, usage) makes perfect. Following chapters 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 — 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

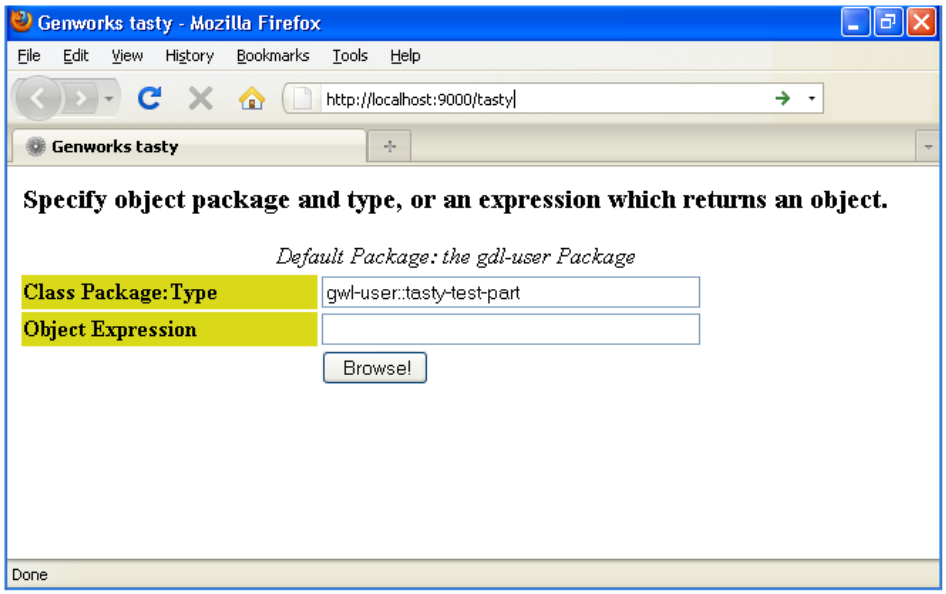


Figure 6.2: Tasty start-up

`tower1` definition in Figure ??. This definition is in the `:chapter-5` package. Consequently, the specification will be `chapter-5: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 (see figure 6.3). 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 device) 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.

Figure 6.3: Tasty Interface

- 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. For example for the tower assembly this will be as depicted in figure **??**

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.

[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 different in your installation.

Alternatively, you can select the "rapid" button from third toolbar which is symbolized by a 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 look ahead to Chapter **??** if you are interested in more details on this GDL syntax). We will also pass 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 (see Figure **??**). 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 (see Figure **??**).

Chapter 7

Working with Geometry in GDL

Although Genworks GDL is a powerful framework for all kinds of general-purpose computing, 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`.

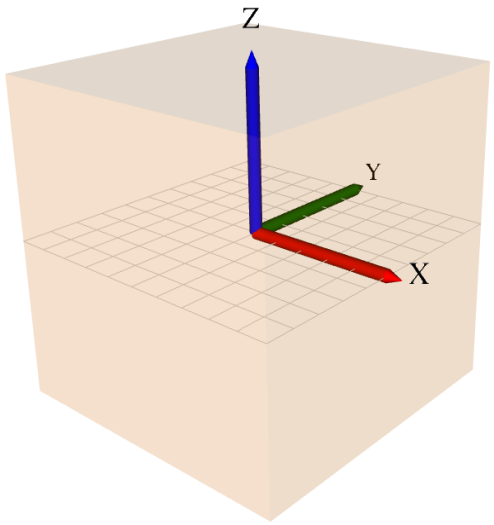


Figure 7.1: Coordinate System in Trimetric View

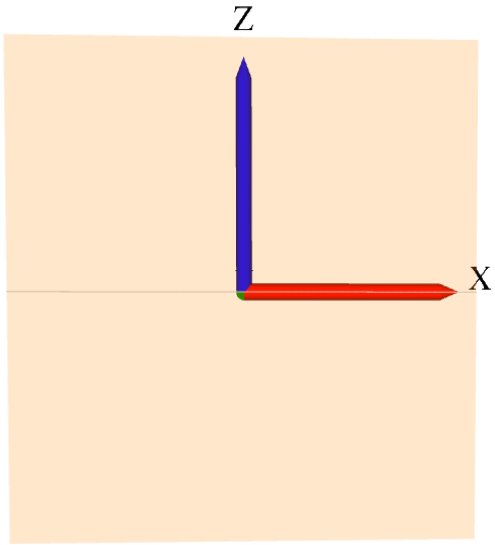


Figure 7.2: Coordinate System in Front View

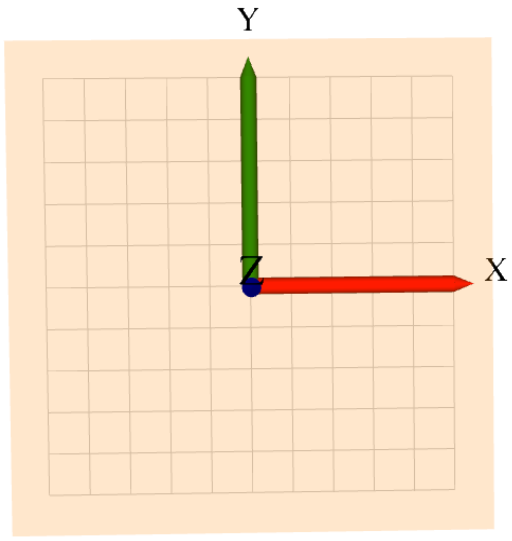


Figure 7.3: Coordinate System in Top View

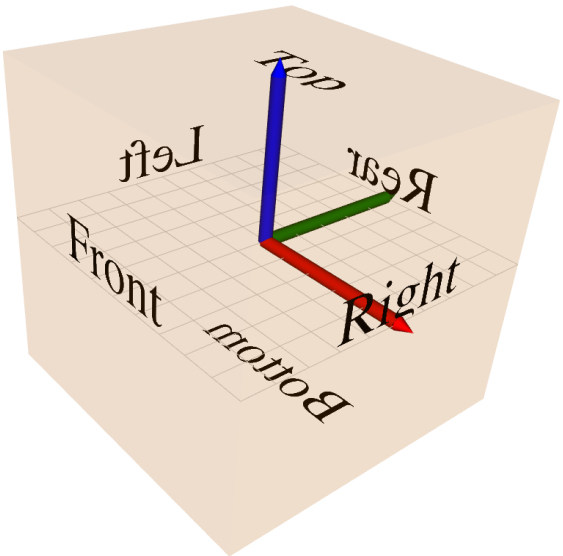


Figure 7.4: Coordinate System with Symbolically Labeled Faces

```
(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 shows 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.

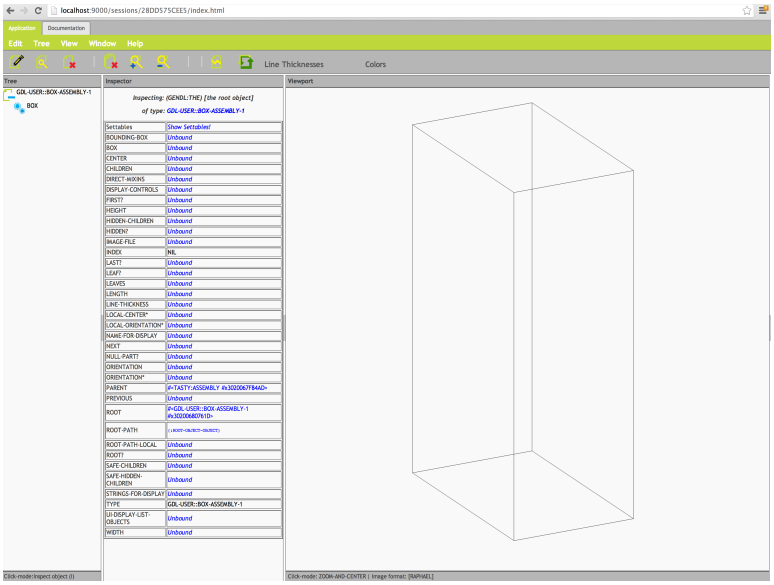


Figure 7.6: Simple box displayed in tasty

```
(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

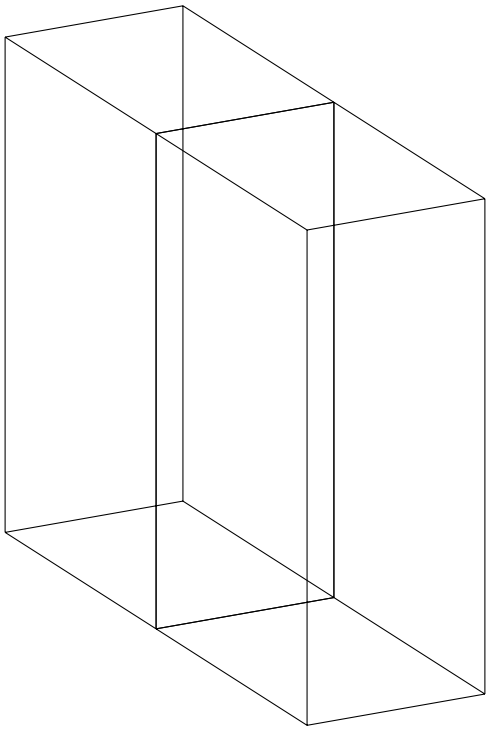


Figure 7.8: Positioned Boxes

```
(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 next 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 7.10 and 7.10.

7.2.3 Relative positioning using the translate operator

It is usually wise 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 fine 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 falls apart.

[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).

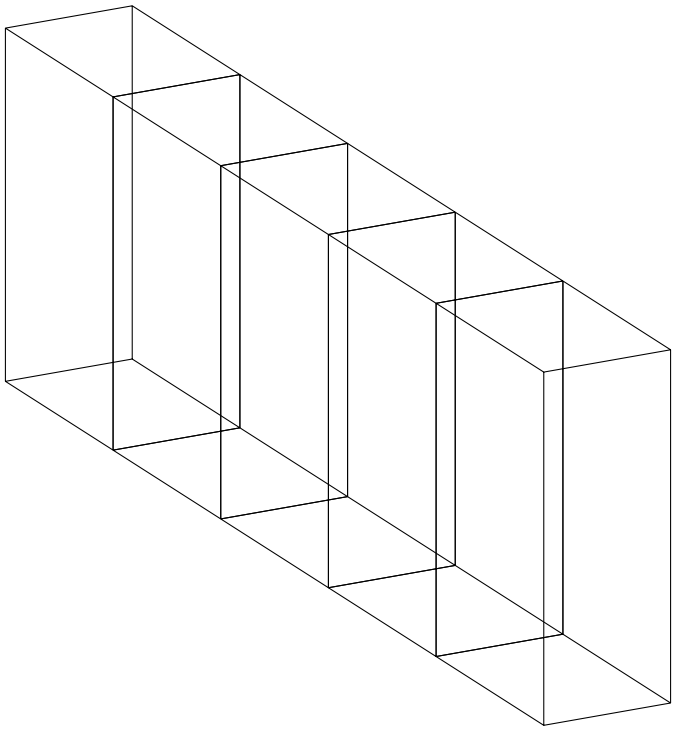


Figure 7.10: Positioned by Index

```
(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, chosen 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 certain default display characteristics[4] for objects in GDL, such as:

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

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

[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`. Lenses will be covered in more detail in Chapter **??**.

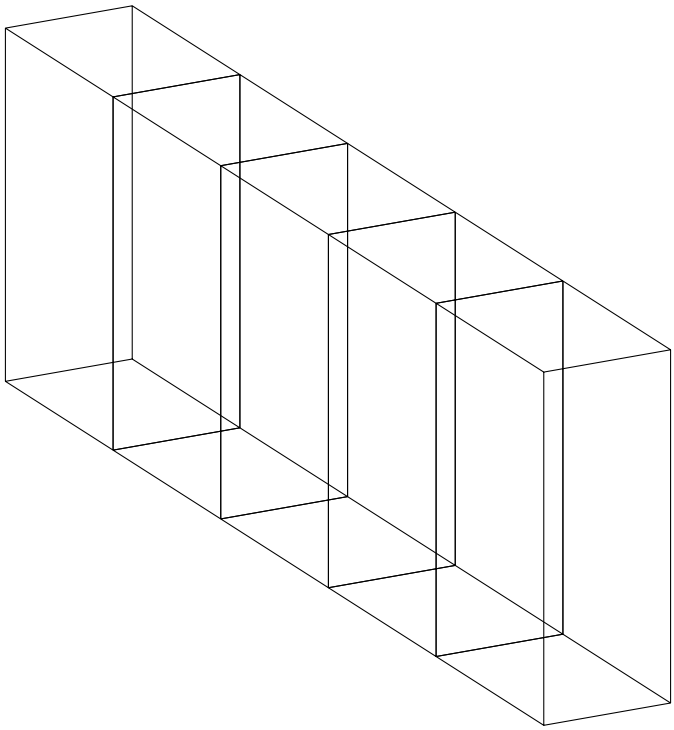
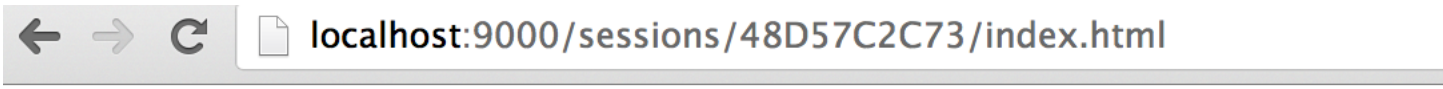


Figure 7.12: Translated by Index



Color	Hex	Decimal
:GREEN-PALE	"#8fbc8f"	0.5607843S0, 0.7372549S0
:BLUE-NEON	"#4d4dff"	0.3019608S0, 0.3019608S0,
:GREY	"#c0c0c0"	0.7529412S0, 0.7529412S0,
:GREEN-COPPER	"#527f76"	0.3215686S0, 0.4980392S
:TEAL	"#008080"	0.0S0, 0.5019608S0, 0.5019
:BLUE-STEEL-LIGHT	"#8f8fbd"	0.5607843S0, 0.5607843S
:GREEN-OLIVE-DARK	"#4f4f2f"	0.3098039S0, 0.3098039S
:PURPLE	"#800080"	0.5019608S0, 0.0S0, 0.5019
:BLUE-SLATE	"#007fff"	0.0S0, 0.4980392S0, 1.0S0
:GREEN-LIME	"#32cd32"	0.1960784S0, 0.8039216S0
:SCARLET	"#8c1717"	0.5490196S0, 0.0901961S
:WHITE	"#ffffff"	1.0S0, 1.0S0, 1.0S0
:GREEN-SPRING	"#00ff7f"	0.0S0, 1.0S0, 0.4980392S0
:RED-VIOLET	"#cc3299"	0.8S0, 0.1960784S0, 0.6S0
:WOOD-DARK	"#855e42"	0.5215686S0, 0.3686275S
:BLUE-MEDIUM	"#3232cd"	0.1960784S0, 0.1960784S
: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

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 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 demonstrates the use of the `:color` keyword in the `display-controls` for our positioned boxes example.

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 takes 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))
```

Usually, 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 use 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.

```
(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

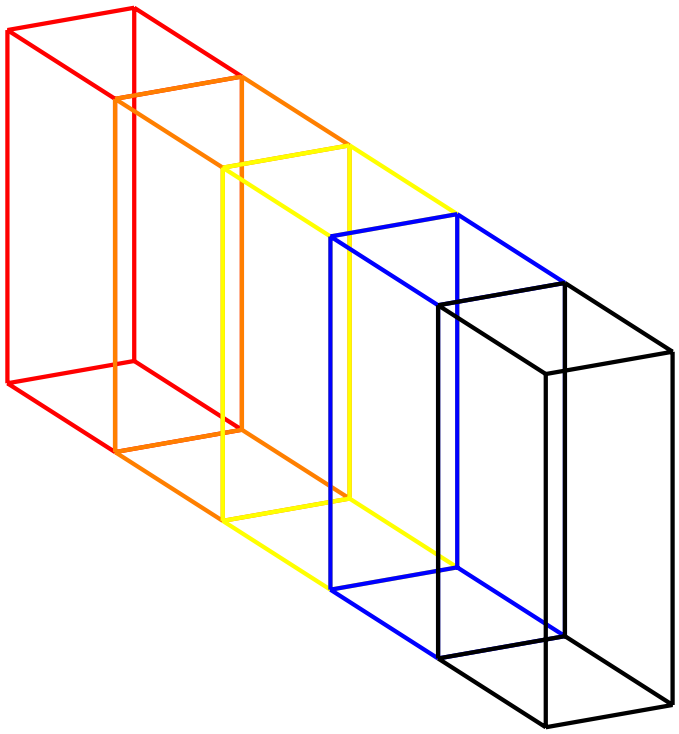


Figure 7.15: Color controlled by display-controls

```
(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

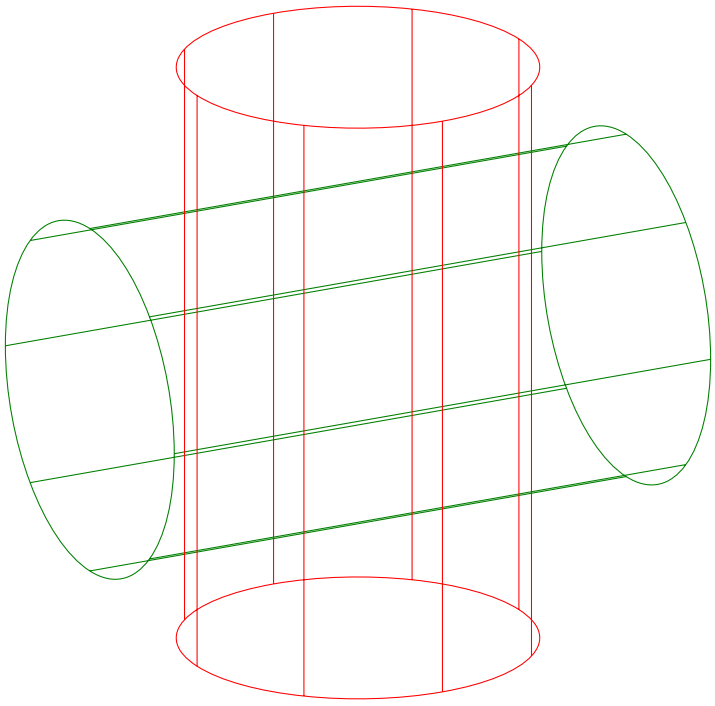


Figure 7.17: Cylinder aligned vertically

```
(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

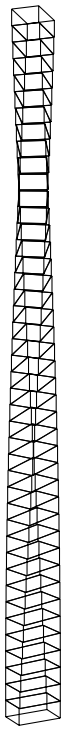


Figure 7.19: Twisty Tower

```
(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

7.2.7 Assemblies

Objects which you define can be used just like the built-in primitives. This illustrates why it is important for the positioning and orientation passed into a child object to be *relative* to that 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 **??** 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 mainly 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 113

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

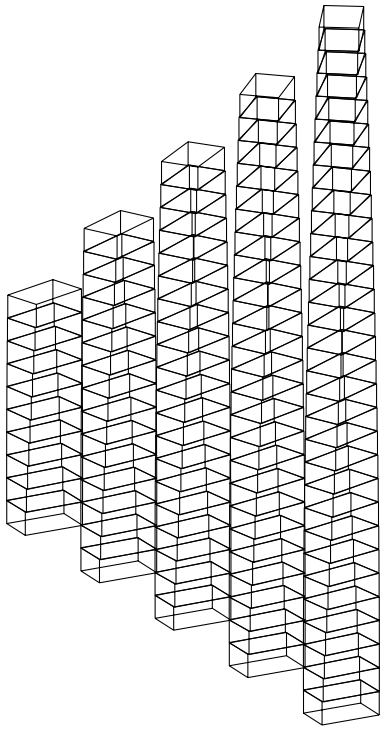


Figure 7.21: Tower Assembly

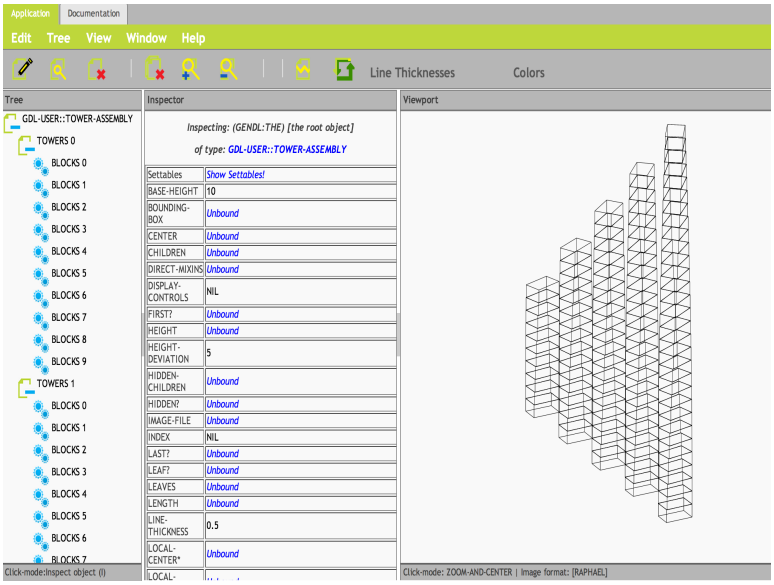


Figure 7.22: Tower Assembly as displayed in Tasty

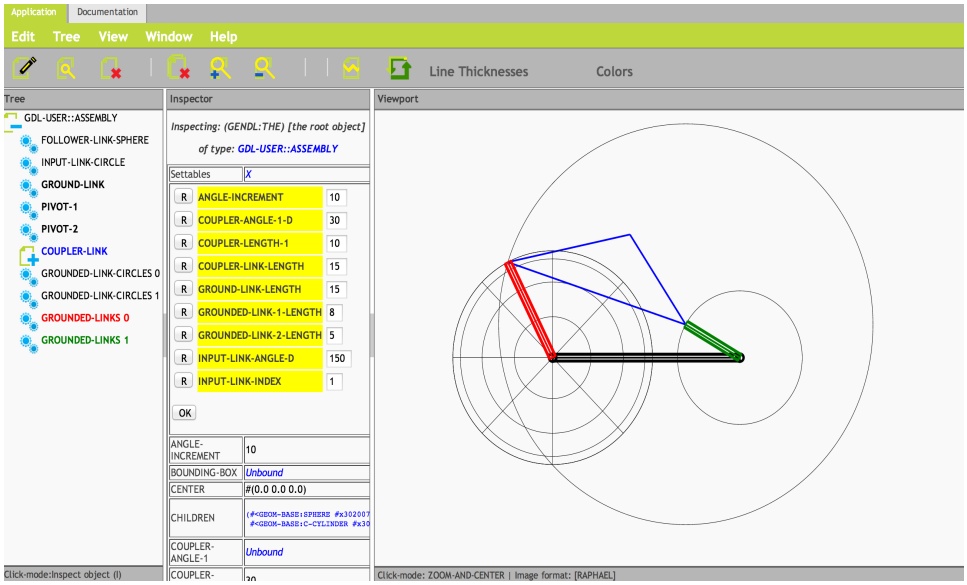


Figure 7.23: Four-bar Link Mechanism

- `line` described on page 133
- `arc` described on page 96
- `ellipse` described on page 117
- `bezier-curve`[6] described on page 107

Basic 3D primitives include:

- `sphere` described on page 145
- `cylinder` described on page 116
- `cone` described on page 114
- `global-polyline` described on page 127
- `global-polygon-projection` described on page 125
- `global-filleted-polyline` described on page 127
- `torus` described on page 149
- `route-pipe` described on page 143

[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 with the SMLib geometry kernel. These are covered in chapter **??**

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 control* elements such as type-in fields, checkboxes, and choice lists [using this capability does require a basic working knowledge of the HTML language].[2].

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

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

8.1 Package and Environment for Web Development

Similarly to `gdl:define-package`, you can use `gwl:define-package` in order to create a working package which has access to the symbols you will need for building a web application (in addition to 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 acronym "GWL" stands for Generative Web Language, which is not a separate language from GDL itself, but rather is a set of primitive objects and functions available within GDL for building web applications. The YADD reference documentation for package "Generative Web Language" provides detailed specifications for all the primitive objects and functions.

8.2 Traditional Web Pages and Applications

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

1. Mix `base-html-sheet` into the object definition.

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

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

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

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

Figure 8.1: Simple Static Page Example

2. define a GDL function called `main-sheet` within the object definition.

The `main-sheet` function should return valid HTML for the page. The easiest way to produce HTML is with the use of an HTML generating library, such as CL-WHO[3] or HTMLGen[4], both of which are built into GDL.

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

8.2.1 A Simple Static Page Example

In Figure 8.1, GWL convenience macro `with-cl-who` is used; this sets up a standard default environment for outputting HTML within a GWL application.

The code in Figure 8.1 produces HTML output as shown in Figure 8.2 which looks similar to Figure 8.3 in a web browser.

Several important concepts are lumped into this example. Note the following:

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

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

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

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

Figure 8.2: Simple Static Page Example

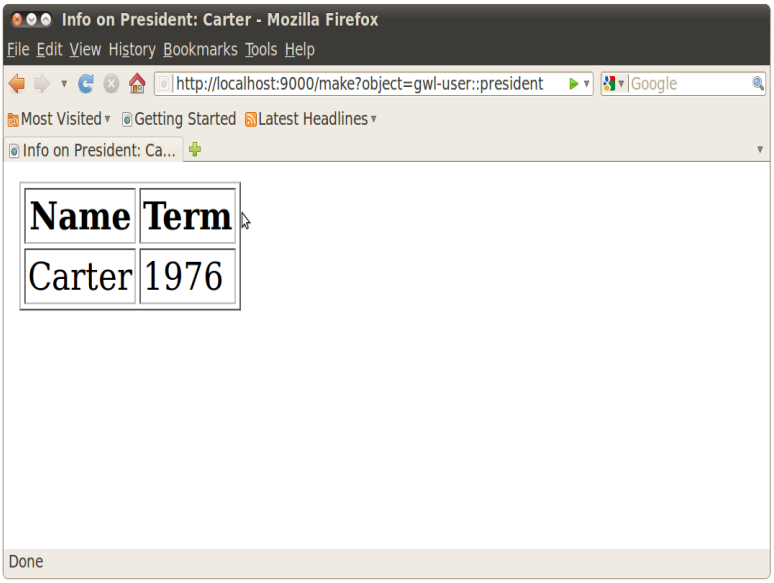


Figure 8.3: Simple Static Page Example

- We use the keyword argument `:indent t` in order to pretty-print the generated HTML. This does not affect the browser display but can make the generated HTML easier to read and debug. This option should be left as nil (the default) for production deployments.
- The `fmt` symbol has special meaning within the cl-who environment and works the same as a Common Lisp `(format nil ...)`, in order to evaluate a format string together with matching arguments, and produce a string at runtime.
- The `str` symbol has special meaning within the cl-who environment and works by evaluating an expression at runtime to return a string or other printable object, which is then included at that point in the HTML output.
- Expressions within the `body` of an HTML tag have to be evaluated, usually by use of the `fmt` or `str` in cl-who. There are three examples of this in the above sample: one `fmt` and two `str`.
- Expressions within a *tag attribute* are always evaluated automatically, and so do **not** require a `str` or other special symbol to force evaluation at runtime. Tag attributes in HTML (or XML) are represented as a plist spliced in after a tag name, wrapped in extra parentheses around the tag name. In the sample above, the `:border (the table-border)` is an example of a tag attribute on the `:table` tag. Notice that the expression `(the table-border)` does not need `str` in order to be evaluated - it gets evaluated automatically.
- In cl-who, if a tag attribute evaluates to `nil`, then that tag attribute will be left out of the output completely. For example if `(the table-border)` evaluates to nil, then the `:table` tag will be outputted without any attributes at all. This is a convenient way to conditionalize tag attributes.
- The URL `http://localhost:9000/make?object=gwl-user::president` is published automatically based on the package and name of the object definition. When you visit this URL, the response is redirected to a unique URL identified by a *session ID*. This ensures that each user to your application site will see their own specific instance of the page object. The session ID is constructed from a combination of the current date and time, along with a pseudo-random number.

8.2.2 A Simple Dynamic Page which Mixes HTML and Common Lisp/GDL

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

The example in Figure 8.4 uses this technique to output an HTML table row for each "row" of data in a list of lists. The output looks similar to Figure 8.5 in a web browser.

Note the following from this example:

- `title` is a `let` variable, so we use `(str title)` to evaluate it as a string. We do not use `(str (the title))` because `title` is a local variable and not a message (i.e. slot) in the object.
- Inside the `dolist`, we "drop back into" HTML mode using the `htm` operator.

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

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

   (table-border 1))

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

Figure 8.4: Mixing Static HTML and Dynamic Content

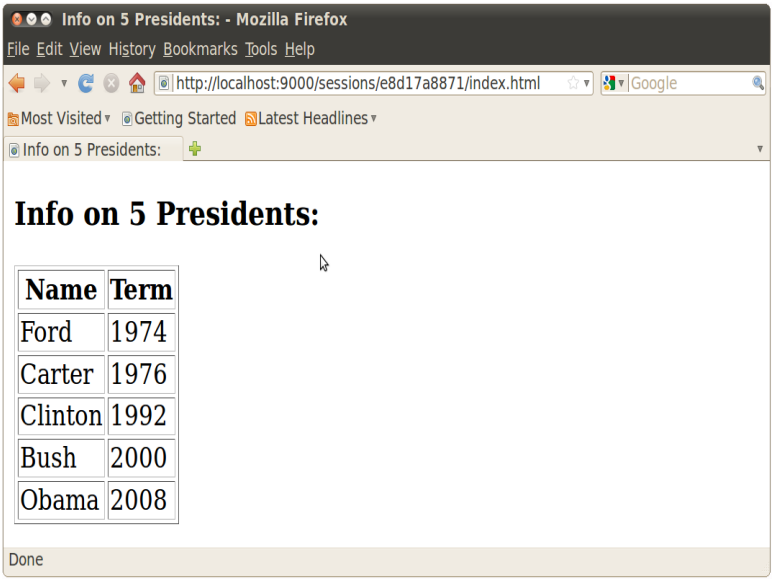


Figure 8.5: Mixing Static HTML and Dynamic Content

8.2.3 Linking to Multiple Pages

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

In Figures 8.6 and 8.7, we provide links from a parent page into a child page with detailed information on each president. The output looks similar to Figure 8.8 in a web browser.

Note the following from this example:

- The `write-self-link` message is a function which can take a keyword argument of `:display-string`. This string is used for the actual hyperlink text.
- There is a `write-back-link` message which similarly can take a keyword argument of `:display-string`. This generates a link back to `(the return-object)` which, by default in base-html-sheet, is `(the parent)`.

8.2.4 Form Controls and Fillout-Forms

Form Controls

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

The form-controls provided by GDL are documented in YADD accessible with

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

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

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

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

   (table-border 1))


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


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

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

Figure 8.6: Linking to Multiple Pages

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

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

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

Figure 8.7: Linking to Multiple Pages

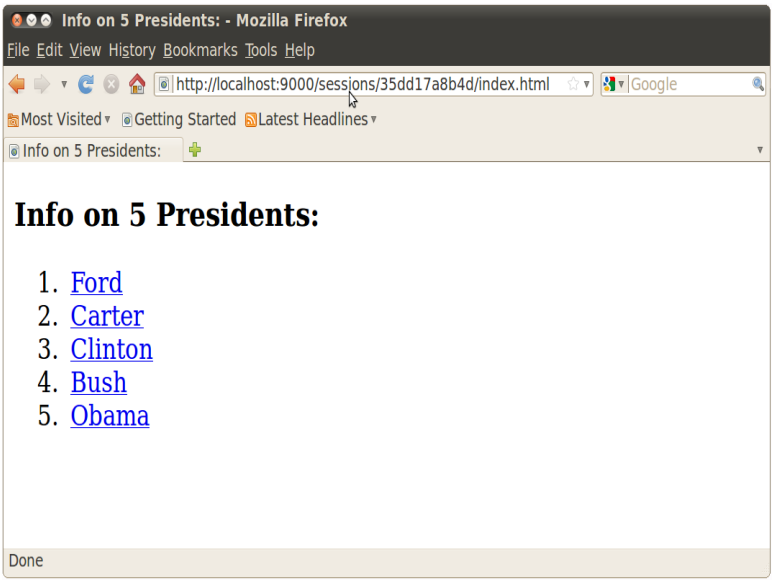


Figure 8.8: Linking to Multiple Pages

`http://localhost:9000/yadd`

and in Chapter **??** 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 is often not necessary). New form-controls such as for numbers, dates, etc will soon be added to correspond to latest HTML standards.

Fillout Forms

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

In Figure 8.9 is an example which allows the user to enter a year, and the application will respond with the revenue amount for that year. Additional form controls are also provided to adjust the table border and cell padding.

This example, when instantiated in a web browser, might look as shown in Figure 8.10.

8.3 Partial Page Updates with gdlAjax

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

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

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

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

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

  :input-slots

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

  :functions

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

  :objects

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

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

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


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


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

Figure 8.9: Form Controls and Fillout Forms

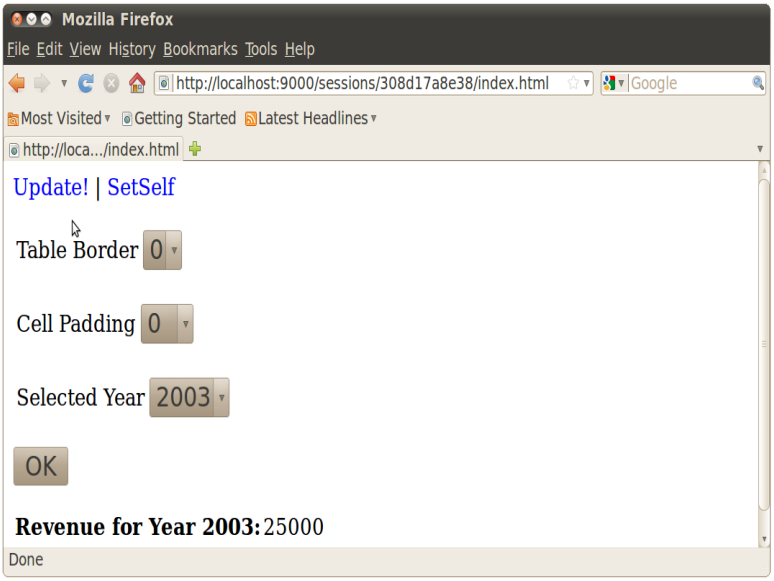


Figure 8.10: Form Controls and Fillout Forms

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 present in some other Ajax frameworks.

8.3.1 Steps to Create a gdlAjax Application

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

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

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

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

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

Note the following from the example in Figure 8.11:

- We mix in `base-ajax-sheet` and specify a `main-sheet-body` slot, which uses `with-cl-who-string` to compute a string of HTML. This approach is also easier to debug, since the `main-sheet-body` string can be evaluated in the tasty inspector or at the command-line.
- We use `str` to include the string for the main page section (called `main-section` in this example) into the `main-sheet-body`.
- In the `main-section`, we also use `str` to include the html-string for each of three form-controls. We have provided a form control for the table border, the table padding, and the revenue year to look up.
- The only page section in this example is `(the main-section)`. This is defined as a child object, and has its `inner-html` computed in the parent and passed in as an input. The `sheet-section` will automatically compute a `main-div` message based on the `inner-html` that we are passing in. The `main-div` is simply the `inner-html`, wrapped with an HTML DIV (i.e. "division") tag which contains a unique identifier for this section, derived from the root-path to the GDL object in the in tree which represents the sheet section.
- We introduce the CL function `gwl:publish-gwl-app`, which makes available a simplified URL for visiting an instance of this object in the web browser. In this case, we can access the instance using `http://localhost:9000/revenue-lookup`

Notice also the use of `:ajax-submit-on-change? ...` in each of the form-controls. This directs the gdlAjax system to "scrape" the values of these form controls and "bash" them into the `value` slot of the corresponding object on the server, whenever they are changed in the browser. No "Submit" button press is necessary.

It is also possible programmatically to send form-control values, and/or call a GDL Function, on the server, by using the `gdl-ajax-call` GDL function. This function will emit the necessary JavaScript code to use as an event handler, e.g. for an "onclick" event. For example, you could have the following snippet somewhere in your page:

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

This will produce a piece of text "Press Me," which, when pressed, will have the effect of calling a function named `restore-defaults!` in the page's object on the server. If the function `restore-defaults!` is not defined, an error will result. The `gdl-ajax-call` GDL function can also send arbitrary form-control values to the server by using the `:form-controls` keyword argument, and listing the relevant form-control objects. The `gdl-ajax-call` GDL function is fully documented in YADD and the reference appendix.

If for some reason you want to do more than one `gdl-ajax-call` sequentially, then it is best to use `gdl-sjax-call` instead. This variant will cause the browser to wait until each call completes, before making the next call. To achieve this, you would want to append the strings together, e.g:

```
((:span :onclick (string-append (the (gdl-sjax-call ...))
                                (the (gdl-sjax-call ...))
                                (the (gdl-sjax-call ...))) ... ))
```

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

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

  :input-slots

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

  :computed-slots

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

  :objects

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

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

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

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

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

Figure 8.11: Partial Page Updates with GdlAjax

With that said, it is rarely necessary to do these calls sequentially like this, because you can use :form-controls and :function-key simultaneously. As long as your logic works correctly when the form-controls are set before the function is called, then you can group the functions together into a "wrapper-function," and do the entire processing with a single Ajax (or Sjax) call. Normally this would be be the recommended approach whenever possible.

8.3.2 Including Graphics

The fundamental mixin or child type to make a graphics viewport is `base-ajax-graphics-sheet`. This object definition takes several optional input-slots, but the most essential 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 8.12 contains a simple box with two graphics viewports and ability to modify the length, height, and and with of the box:

This will produce a web browser output similar to what is shown in Figure 8.13.

Note the following from this example:

- The `(:use-raphael? t)` enables raphael for SVG or VML output.
- The `:raphael` image-format generates SVG or VML, depending on the browser.
- We conditionally include development-links for full Update and SetSelf! actions.
- We include two viewports in the `main-sheet-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).

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

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

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

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

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

   (inputs-section :type 'inputs-section)

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



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

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

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


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

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

Figure 8.12: Including Graphics in a Web Page

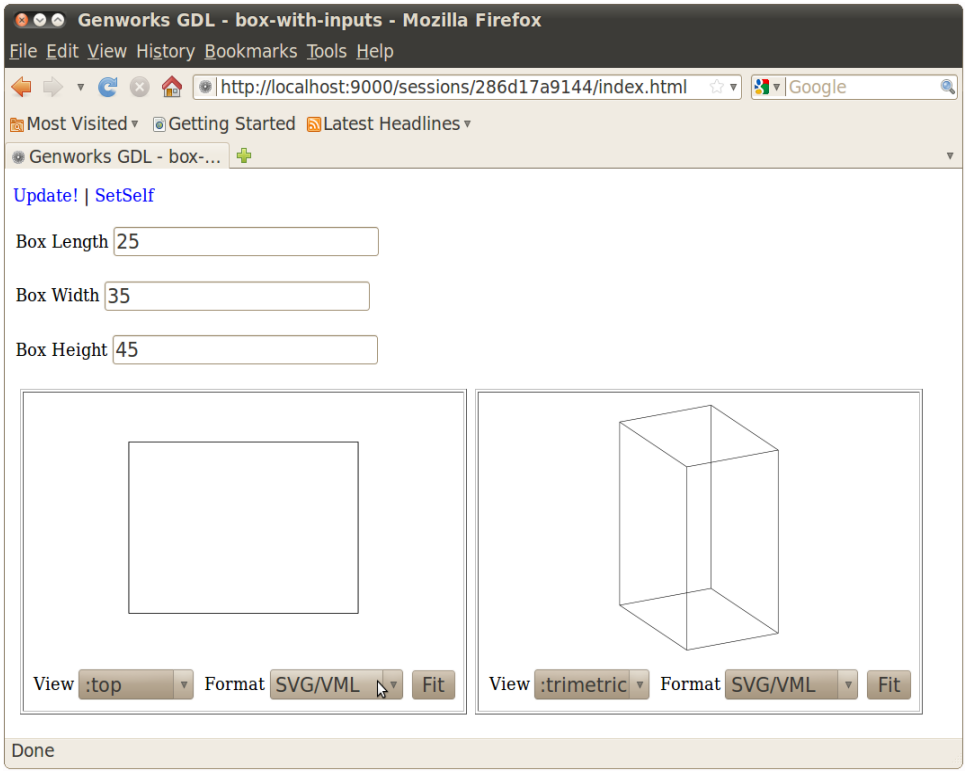


Figure 8.13: Including Graphics

Chapter 9

More Common Lisp for GDL

Chapter 10

Advanced GDL

Upgrade Notes

GDL 1580 marked the end of a major branch of GDL development, and 1581 was an upgraded new version, which in turn has now been supplanted by 1582.

This addendum lists the typical modifications you will want to consider for upgrading from GDL 1580 to GDL 1582, or later versions.

- (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 yet available for 1582. Instead of updating incrementally with patches, the intention starting with GDL 1582 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 11

Reference for GDL Objects and Operators

11.1 CL-LITE (Compile-and-Load Lite Utility)

11.1.1 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.

11.1.2 Function and Macro Definitions

- **CL-PATCH**

11.2 GENDL (Base Core Kernel Engine) Nicknames: Gdl, Genworks, Base

11.2.1 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 lt;direction-keywordgt; lt;numbergt; lt;direction-keywordgt; lt;numbergt;))` in an `:objects` specification.

  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.

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-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 specifqqied 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).

Strings-for-display *String or List of Strings*

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

Visible-children *List of GDL Instances*

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

Computed slots:

Aggregate *GDL Instance*

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

All-mixins *List of Symbols*

Lists all the superclasses of the type of this object.

Children *List of GDL Instances*

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

Direct-mixins *List of Symbols*

Lists the direct superclasses of the type of this object.

First? *Boolean*

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

Hidden-children *List of GDL Instances*

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

Index *Integer*

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

Last? *Boolean*

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

Leaf? *Boolean*

T if this object has no children, NIL otherwise.

Leaves *List of GDL Objects*

A Collection of the leaf nodes of the given object.

Name-for-display *Keyword symbol*

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

Next *GDL Instance*

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

Parent *GDL Instance*

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

Previous *GDL Instance*

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

Root-path *List of Symbols or of Pairs of Symbol and Integer*

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

Root-path-local *List of Symbols or of Pairs of Symbol and Integer*

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

Root? *Boolean*

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

Safe-children *List of GDL Instances*

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

Safe-hidden-children *List of GDL Instances*

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

Type *Symbol*

The GDL Type of this object.

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)

11.2.2 Function and Macro Definitions

- **ALIST2PLIST**
- **ALWAYS**

- **APPEND-ELEMENTS** [Macro]
- **CL-LITE**
- **CYCLIC-NTH**
- **DEFAULTING** [Macro]
- **DEFINE-FORMAT** [Macro]
- **DEFINE-LENS** [Macro]
- **DEFINE-OBJECT** [Macro]
- **DEFINE-OBJECT-AMENDMENT** [Macro]
- **DIV**
- **ENSURE-LIST**
- **FIND-DEPENDANTS**
- **FIND-DEPENDENCIES**
- **FIND-MESSAGES-USED-BY**
- **FIND-MESSAGES-WHICH-USE**
- **FLATTEN**
- **FORMAT-SLOT** [Macro]
- **FROUND-TO-NEAREST**
- **HALF**
- **IGNORE-ERRORS-WITH-BACKTRACE** [Macro]
- **INDEX-FILTER**
- **ISO-8601-DATE**
- **LASTCAR**
- **LEAST**
- **LIST-ELEMENTS** [Macro]
- **LIST-OF-NUMBERS**
- **MAKE-KEYWORD**
- **MAKE-OBJECT**
- **MAPSEND**

- **MAPTREE**
- **MAX-OF-ELEMENTS** [Macro]
- **MIN-OF-ELEMENTS** [Macro]
- **MOST**
- **NEAR-TO?**
- **NEAR-ZERO?**
- **NEVER**
- **NUMBER-FORMAT**
- **NUMBER-ROUND**
- **PLIST-KEYS**
- **PLIST-VALUES**
- **PRINT-MESSAGES** [Macro]
- **PRINT-VARIABLES** [Macro]
- **READ-SAFE-STRING**
- **READ-SNAPSHOT**
- **REMOVE-PLIST-ENTRY**
- **ROUND-TO-NEAREST**
- **SAFE-FLOAT**
- **SAFE-SORT**
- **SET-FORMAT-SLOT** [Macro]
- **SPLIT**
- **STATUS-MESSAGE**
- **STRING-APPEND**
- **SUM-ELEMENTS** [Macro]
- **THE** [Macro]
- **THE-CHILD** [Macro]
- **THE-ELEMENT** [Macro]
- **THE-OBJECT** [Macro]

- **TWICE**
- **UNDEFINE-OBJECT**
- **UNIVERSAL-TIME-FROM-ISO-8601**
- **WITH-ERROR-HANDLING** [Macro]
- **WITH-FORMAT** [Macro]
- **WITH-FORMAT-SLOTS** [Macro]
- **WRITE-ENV** [Macro]
- **WRITE-PLIST**
- **WRITE-THE** [Macro]
- **WRITE-THE-OBJECT** [Macro]
- [2]

11.2.3 Variables and Constants

- ***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?***
- ***OUT-OF-BOUNDS-SEQUENCE-REFERENCE-ACTION***
- ***REMEMBER-PREVIOUS-SLOT-VALUES?***

- ***ROOT-CHECKING-ENABLED?***
- ***RUN-WITH-CIRCULAR-REFERENCE-DETECTION?***
- ***RUN-WITH-DEPENDENCY-TRACKING?***
- ***UNDECLARED-PARAMETERS-ENABLED?***
- ***ZERO-EPSILON***
- **+PHI+**
- **2PI**
- **PI/2**

11.3 GDL-USER

11.4 GENDL-DOC

11.5 GEOM-BASE (Wireframe Geometry)

11.5.1 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.

  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.

```
(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 11.1: Example Code for ANGULAR-DIMENSION

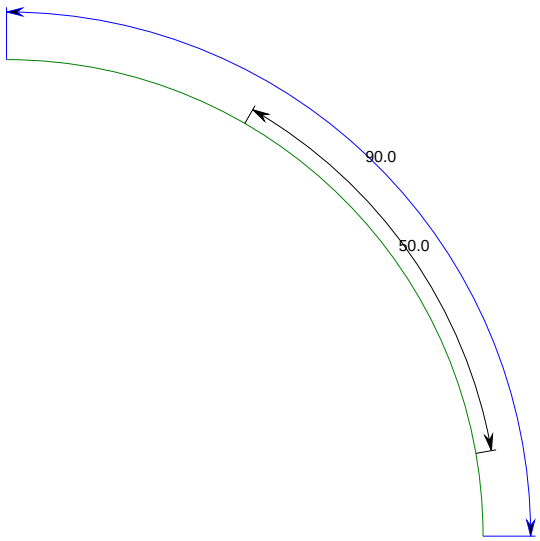


Figure 11.2: ANGULAR-DIMENSION example

Start-point *3D Point*

The start point of the arc being measured.

Text-along-leader-padding-factor *Number*

Amount of padding above leader for text-along-leader? t. This is multiplied by the character-size to get the actual padding amount. Defaults to 1/3.

Witness-1-to-center? *Boolean*

Determines whether a witness line extends all the way from the start-point to the center. Defaults to nil.

Witness-2-to-center? *Boolean*

Determines whether a witness line extends all the way from the end-point to the center. Defaults to nil.

Computed slots:

Dim-value *Number*

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

- **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):

```
(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 11.3: Example Code for ARC

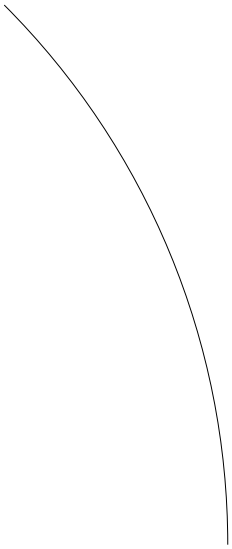


Figure 11.4: ARC example

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.

```
(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 11.5: Example Code for BASE-DRAWING

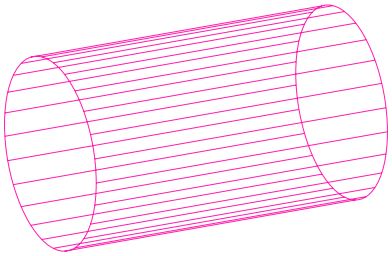


Figure 11.6: BASE-DRAWING example

Width *Number*

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

- **BASE-OBJECT**

  Mixins: VANILLA-MIXIN

  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`.

  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 wireframe representations of this object.

  :dash-pattern (currently PDF/PNG/JPEG only). This is a list of two or three numbers which indicate the length, in pixels, of the dashes and blank spaces in a dashed line. The optional third number indicates how far into the line or curve to start the dash pattern.

  Height *Number*

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

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

(define-object tower (base-object)

 :input-slots
  ((number-of-blocks 50) (twist-per-block 1)
   (block-height 1) (block-width 5) (block-length 7))

 :objects
  ((blocks :type 'box
           :sequence (:size (the number-of-blocks))
           :center (translate (the center)
                              :up (* (the-child index)
                                     (the-child height)))
           :width (the block-width)
           :height (the block-height)
           :length (the block-length)
           :orientation (alignment
                         :rear (if (the-child first?)
                                   (rotate-vector-d (the (face-normal-vector  :rear))
                                                    (the twist-per-block)
                                                    (the (face-normal-vector :top)))
                                   (rotate-vector-d (the-child previous
                                                    (face-normal-vector :rear))
                                                    (the twist-per-block)
                                                    (the (face-normal-vector :top))))
                         :top (the (face-normal-vector :top))))))
;;
;;Test run
;;
#|
gdl-user(46): (setq self (make-object 'tower))
#&lt;tower @ #x750666f2&gt;
gdl-user(47): (setq test-center (the (blocks 10) center))
#(0.0 0.0 10.0)
gdl-user(48): (the (blocks 10) (global-to-local test-center))
#(0.0 0.0 0.0)
gdl-user(49): (the (blocks 10) (local-to-global (the (blocks 10)
                                                     (global-to-local test-center))))
#(0.0 0.0 10.0)
gdl-user(50):
gdl-user(50): (setq test-vertex (the (blocks 10) (vertex :top :right :rear)))
#(1.7862364748012536 3.9127176305081863 10.5)
gdl-user(51): (the (blocks 10) (global-to-local test-vertex))
#(2.500000000000001 3.500000000000001 0.5)
gdl-user(52): (the (blocks 10) (local-to-global (the (blocks 10)
                                                     (global-to-local test-vertex))))
#(1.786236474801254 3.9127176305081877 10.5)
gdl-user(53):
|#
;;
;;
;;
```

Figure 11.7: Example Code for BASE-OBJECT

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.

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 (list (the box) (the length-dim)))

   (length-dim :type 'horizontal-dimension
               :hidden? t
               :start-point (the box (vertex :rear :top :left))
               :end-point (the box (vertex :rear :top :right)))

   (box :type 'box
        :hidden? t
        :length 5 :width 10 :height 15)))

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

  :input-slots (objects)

  :objects

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

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

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

Figure 11.8: Example Code for BASE-VIEW

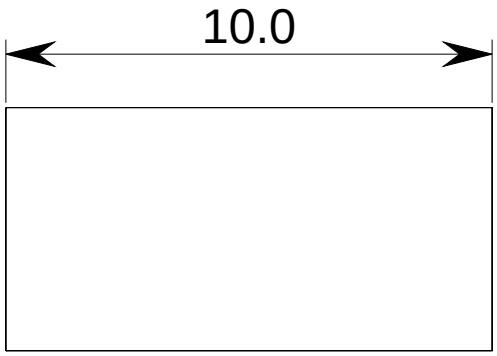


Figure 11.9: BASE-VIEW example

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 11.10: Example Code for BEZIER-CURVE

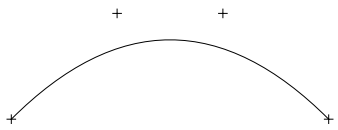


Figure 11.11: BEZIER-CURVE example

```
(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 11.12: 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.

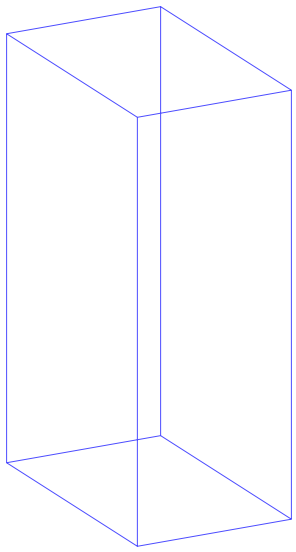


Figure 11.13: BOX example

```
(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 11.14: Example Code for C-CYLINDER

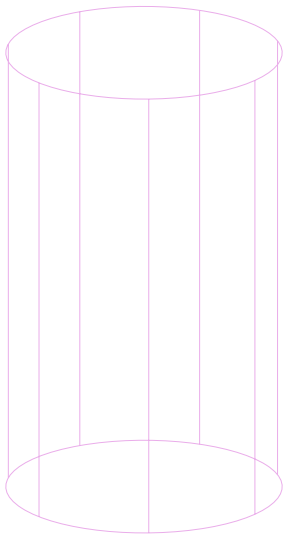


Figure 11.15: C-CYLINDER example

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 11.16: Example Code for CENTER-LINE

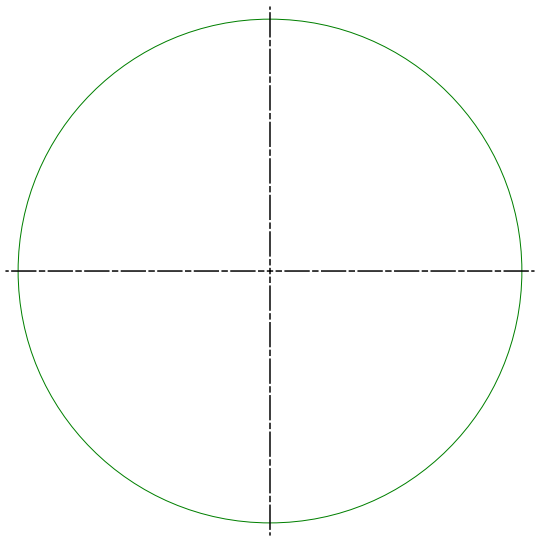


Figure 11.17: CENTER-LINE example

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

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

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

Figure 11.18: 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.

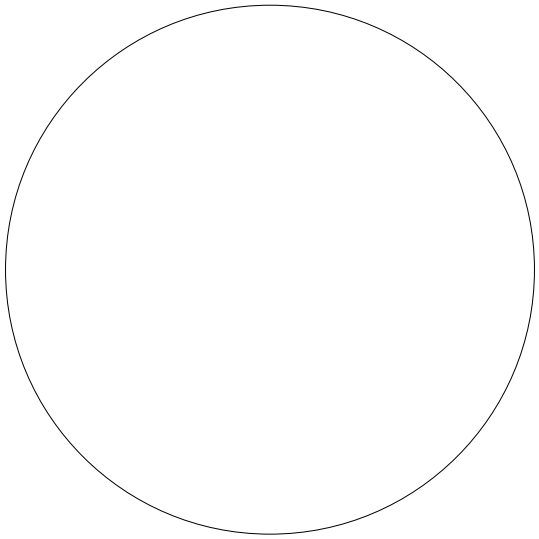


Figure 11.19: CIRCLE example

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 11.20: Example Code for CONE

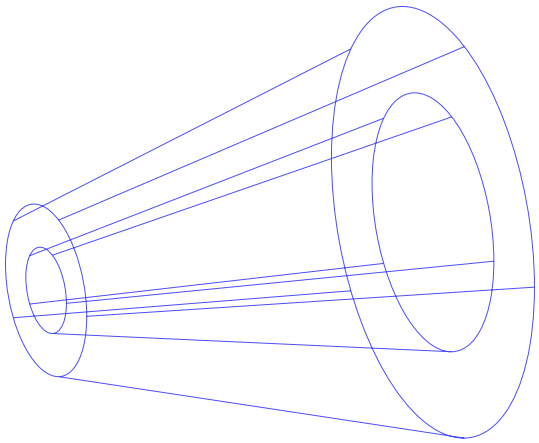


Figure 11.21: CONE example

```
(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 11.22: Example Code for CYLINDER

Height *Number*

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

Width *Number*

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

- **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.

  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.

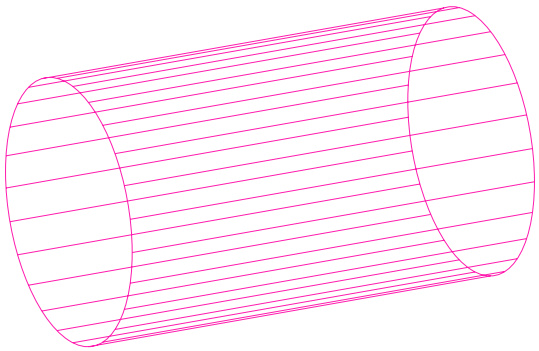


Figure 11.23: CYLINDER example

Inner-radius *Number*
Radius of the hollow inner portion for a hollow cylinder.

Number-of-sections *Integer*
Number of vertical sections to be drawn in wireframe rendering mode.

Top-cap? *Boolean*
Determines whether to include bottom cap in shaded renderings. Defaults to T.

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

Computed slots:

Direction-vector *3D Vector*
Points from the start to the end.

End *3D Point*
The center of the end cap.

Hollow? *Boolean*
Indicates whether there is an inner-radius and thus the cylinder is hollow.

Start *3D Point*
The center of the start cap.

- **ELLIPSE**

  Mixins: ARCOID-MIXIN, BASE-OBJECT

```
(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 11.24: Example Code for ELLIPSE

Description A curve which is the locus of all points in the plane the sum of whose distances from two fixed points (the foci) is a given positive constant. This is a simplified 3D ellipse which will snap to the nearest quarter if you make it a partial ellipse. For a full ellipse, do not specify start-angle or end-angle.

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.

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

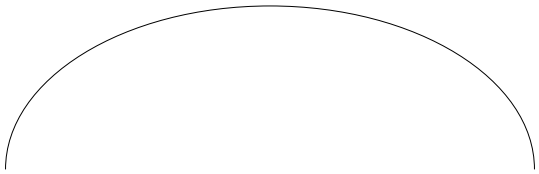


Figure 11.25: ELLIPSE example

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

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

```
(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 11.26: Example Code for GENERAL-NOTE

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

Figure 11.27: GENERAL-NOTE example

- 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

  Description A sequence of points connected by straight line segments, whose corners are filleted according to specified radii. Please see global-filleted-polyline-mixin for documentation on the messages.

- **GLOBAL-FILLETED-POLYLINE-MIXIN**

  Mixins: GLOBAL-POLYLINE-MIXIN

  Input slots (required):

```
(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 11.28: Example Code for GLOBAL-FILLETED-POLYGON-PROJECTION

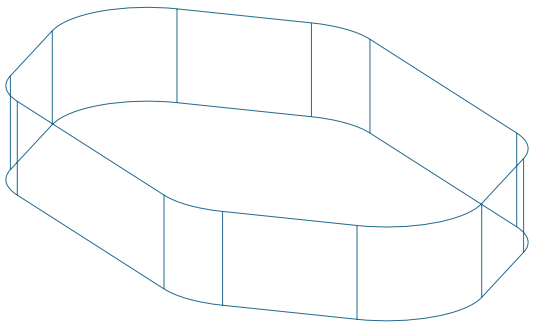


Figure 11.29: GLOBAL-FILLETED-POLYGON-PROJECTION example

```
(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 11.30: Example Code for GLOBAL-FILLETED-POLYLINE

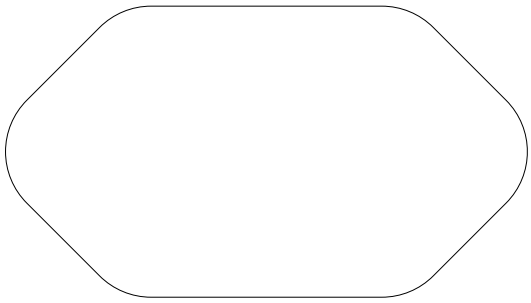


Figure 11.31: GLOBAL-FILLETED-POLYLINE example

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

Input slots (optional):

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.

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

```
(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 11.32: Example Code for GLOBAL-POLYGON-PROJECTION

Description A polygon "extruded" for a given distance along a single vector. For planar polygons, the projection vector must not be orthogonal to the normal of the plane of the polygon. The vertices and projection-vector are given in the global coordinate system, so the local center and orientation do not affect the positioning or orientation of this part.

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.

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.

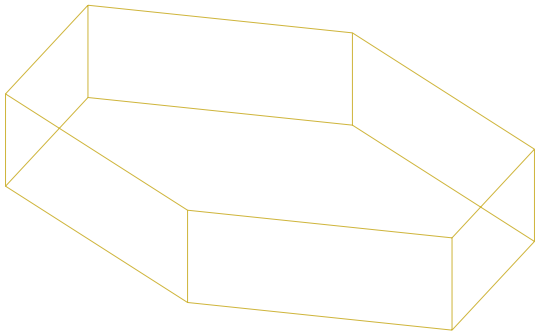


Figure 11.33: GLOBAL-POLYGON-PROJECTION example

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

  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

  Input slots (required):

  Vertex-list *List of 3D Points*

  The vertices ("corners") of the polyline.

  Computed slots:

```
(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 11.34: Example Code for GLOBAL-POLYLINE

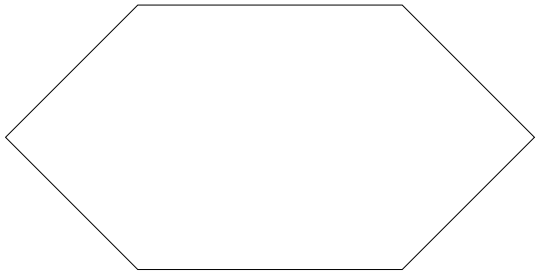


Figure 11.35: GLOBAL-POLYLINE example

```
(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 11.36: Example Code for HORIZONTAL-DIMENSION

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

  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

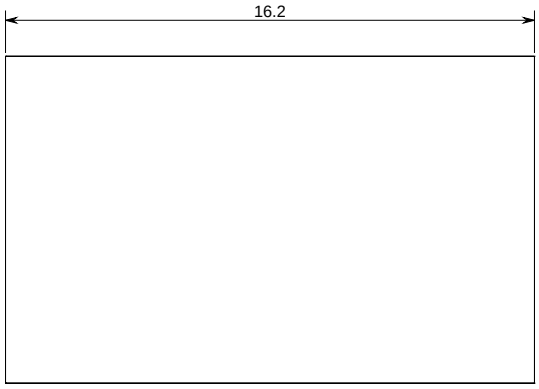


Figure 11.37: HORIZONTAL-DIMENSION example

```
(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 11.38: Example Code for LABEL

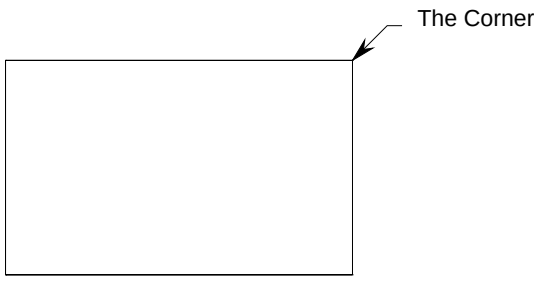


Figure 11.39: LABEL example

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`

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):

```
(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 11.40: Example Code for LINE

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.

- **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.

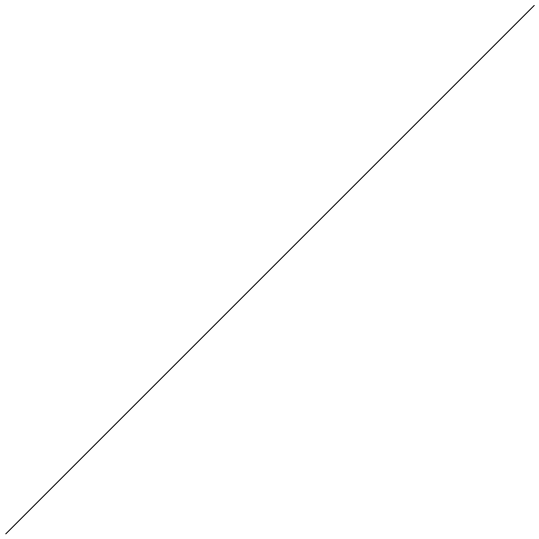


Figure 11.41: LINE example

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.

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.

  Computed slots:

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

  Dim-text-start *3D Point*

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

  Leader-direction-1-vector Must be specified in the subclass except for angular

  Leader-direction-2-vector Must be specified in the subclass except for angular

  Witness-direction-vector Must be specified in the subclass except for angular

- **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).

```
(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 11.42: Example Code for PARALLEL-DIMENSION

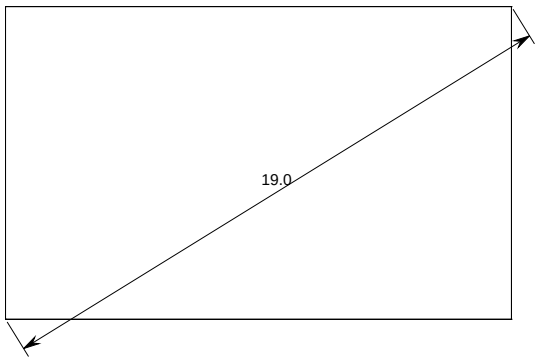


Figure 11.43: PARALLEL-DIMENSION example

```
(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 11.44: Example Code for PIE-CHART

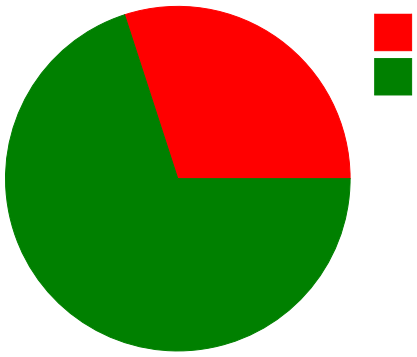


Figure 11.45: PIE-CHART example

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`.

Labelscolors *List of lists, each containing a string and a keyword symbol*

This list should be the same length as `data`. These colors and labels will be assigned to each pie piece and to the legend. Defaults to NIL, must be specified as non-NIL to get a result.

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

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.

```
(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 11.46: Example Code for POINT

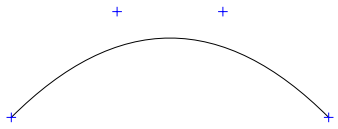


Figure 11.47: POINT example

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.

- **RENDERER-MIXIN**

Mixins: VANILLA-MIXIN

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

Input slots (required):

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.

Input slots (optional):

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`.

```
(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 11.48: Example Code for ROUTE-PIPE

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)
- `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):

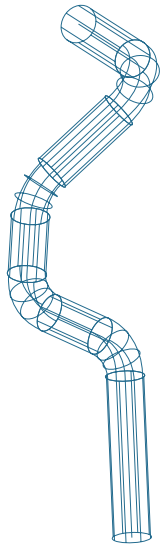


Figure 11.49: ROUTE-PIPE example

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):

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

  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):

  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.

```
(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 11.50: Example Code for SPHERE

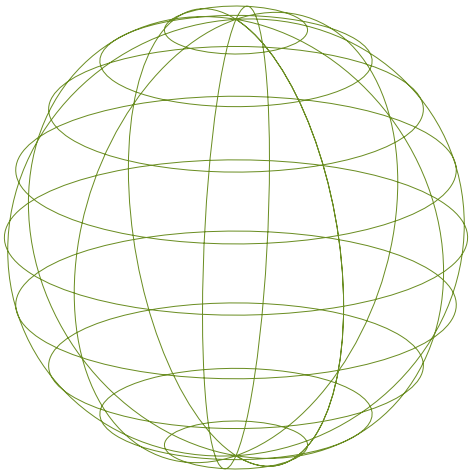


Figure 11.51: SPHERE example

```
(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 11.52: Example Code for SPHERICAL-CAP

Start-horizontal-arc *Angle in radians*

Starting horizontal angle for a partial sphere. Default is 0.

Start-vertical-arc *Angle in radians*

Starting vertical angle for a partial sphere. Default is -pi/2.

Computed slots:

Height *Number*

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

Length *Number*

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

Width *Number*

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

- **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):

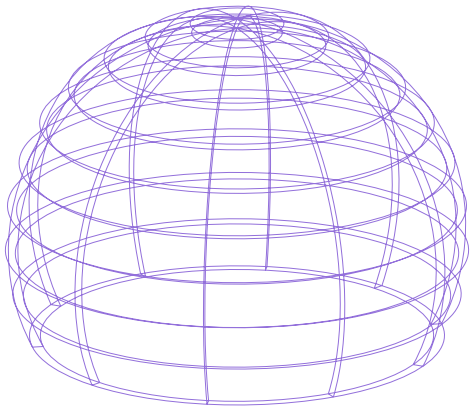


Figure 11.53: SPHERICAL-CAP example

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.

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

  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.

```
(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 11.54: Example Code for TORUS

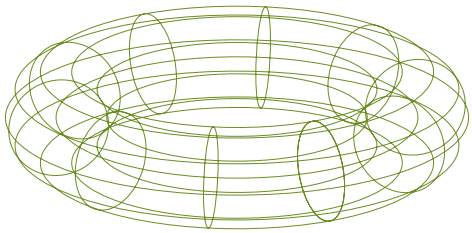


Figure 11.55: TORUS example

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

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.

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:

  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

```
(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 11.56: Example Code for VERTICAL-DIMENSION

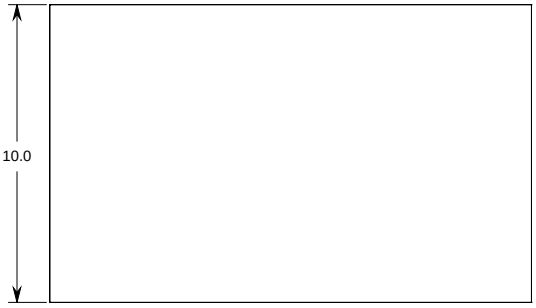


Figure 11.57: VERTICAL-DIMENSION example

11.5.2 Function and Macro Definitions

- **3D-DISTANCE**
- **3D-VECTOR-TO-ARRAY**
- **ACOSD**
- **ADD-MATRICES**
- **ADD-VECTORS**
- **ALIGNMENT**
- **ANGLE-BETWEEN-VECTORS**
- **ANGLE-BETWEEN-VECTORS-D**
- **APPLY-MAKE-POINT**
- **ARRAY-TO-3D-VECTOR**
- **ARRAY-TO-LIST**
- **ASIND**
- **ATAND**
- **COINCIDENT-POINT?**
- **CREATE-OBLIQUENESS**
- **CROSS-VECTORS**
- **DEGREE**
- **DEGREES-TO-RADIANS**
- **DISTANCE-TO-LINE**
- **DOT-VECTORS**
- **EQUI-SPACE-POINTS**
- **GET-U**
- **GET-V**
- **GET-W**
- **GET-X**
- **GET-Y**
- **GET-Z**

- **INTER-CIRCLE-SPHERE**
- **INTER-LINE-PLANE**
- **INTER-LINE-SPHERE**
- **LENGTH-VECTOR**
- **MAKE-POINT** [Macro]
- **MAKE-TRANSFORM**
- **MAKE-VECTOR** [Macro]
- **MATRIX*VECTOR**
- **MATRIX-TO-QUATERNION**
- **MERGE-DISPLAY-CONTROLS** [Macro]
- **MIDPOINT**
- **MULTIPLY-MATRICES**
- **ORTHOGONAL-COMPONENT**
- **PARALLEL-VECTORS?**
- **PROJ-POINT-ON-LINE**
- **PROJECTED-VECTOR**
- **PYTHAGORIZE**
- **QUATERNION-TO-MATRIX**
- **QUATERNION-TO-ROTATION**
- **RADIANS-TO-DEGREES**
- **RADIANS-TO-GRADS**
- **REVERSE-VECTOR**
- **ROLL** [Macro]
- **ROTATE-POINT**
- **ROTATE-POINT-D**
- **ROTATE-VECTOR**
- **ROTATE-VECTOR-D**
- **ROTATION**

- **SAME-DIRECTION-VECTORS?**
- **SCALAR*MATRIX**
- **SCALAR*VECTOR**
- **SORT-POINTS-ALONG-VECTOR**
- **SUBTRACT-VECTORS**
- **TRANSFORM-AND-TRANSLATE-POINT**
- **TRANSFORM-NUMERIC-POINT**
- **TRANSLATE** [Macro]
- **TRANSLATE-ALONG-VECTOR**
- **TRANSPOSE-MATRIX**
- **UNITIZE-VECTOR**
- **ZERO-VECTOR?**

11.5.3 Variables and Constants

- ***BREAK-LEADERS?***
- ***GS-GRAPHICS-ALPHA-BITS***
- ***GS-TEXT-ALPHA-BITS***
- ***HASH-TRANSFORMS?***
- ***ZERO-VECTOR-CHECKING?***
- **+POSTNET-BITS+**

11.6 GWL (Generative Web Language (GWL))

11.6.1 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.

Computed slots (settable):

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.

Hidden objects:

Image-format-selector *Object of type menu-form-control*

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

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.

  Doctype-string *String or nil*

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

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.

Additional-header-js-content *valid javascript*

This javascript is added to the head of the page -just before- the body. When jquery is loaded (by setting the input-slot (use-jquery t)), the javascript can use the '$([selector])'

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

shortcuts to do the magic. The javascript is automagically wrapped in the appropriate html tags to ensure a good execution by the javascript engine. When the use-jquery slot is true (t) than the javascript is wrapped in a '$(document).ready' function as a Good Practice#0153;.

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.

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.

Align *String or nil*

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

Allow-invalid-type? *Boolean*

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

Allow-invalid? *Boolean*

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

Allow-nil? *Boolean*

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

Alt *String or nil*

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

Append-error-string? *Boolean*

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

Default *Lisp value of a type compatible with (the domain)*

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

Disabled? *Boolean*

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

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

. This specifies the expected and acceptable type for the submitted form value. If possible, the submitted value will be coerced into the specified type. The default is

```
(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
```

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.

Onclick *String or nil*

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

Ondblclick *String or nil*

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

Onfocus *String or nil*

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

Onkeydown *String or nil*

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

Onkeypress *String or nil*

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

Onkeyup *String or nil*

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

Onmousedown *String or nil*

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

Onmousemove *String or nil*

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

Onmouseout *String or nil*

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

Onmouseover *String or nil*

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

Onmouseup *String or nil*

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

Onselect *String or nil*

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

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):

  Check-sanity? *Boolean*

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

  Return-object *GDL object*

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

  Target *String*

  Name of a browser frame or window to display this page. Default of NIL indicates to use the same window.

  Transitory-slots *List of keyword symbols*

  Messages corresponding to form fields which should not be retained against Updates to the model (e.g. calls to the update! function or hitting the Update button or link in the browser in development mode). Defaults to NIL (the empty list).

  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:

  After-present! *Void*

  This is an empty function by default, but can be overridden in the respondent of a form, to do some processing after the respondent's `write-html-sheet` function runs to present the object.

After-set! *Void*

This is an empty function by default, but can be overridden in the requestor of a form, to do some processing after the requestor's form values are set into the specified bashee.

Before-present! *Void*

This is an empty function by default, but can be overridden in the respondent of a form, to do some processing before the respondent's `write-html-sheet` function runs to present the object. This can be useful especially for objects which are subclasses of higher-level mixins such as `application-mixin` and `node-mixin`, where you do not have direct access to the `write-html-sheet` function and typically only define the `model-inputs` function. It is not always reliable to do processing in the `model-inputs` function, since some slots which depend on your intended modifications may already have been evaluated by the time the `model-inputs` function runs.

Before-response! *Void*

This is an empty function by default, but can be overridden in a user specialization of base-html-sheet, to do some processing before the header-plist is evaluated and before the HTTP response is actually initiated.

Before-set! *Void*

This is an empty function by default, but can be overridden in the requestor of a form, to do some processing before the requestor's form values are set into the specified bashee.

Check-sanity *NIL or error object*

This function checks the "sanity" of this object. By default, it checks that following the object's root-path from the root resolves to this object. If the act of following the root-path throws an error, this error will be returned. Otherwise, if the result of following the root-path does not match the identity of this object, an error is thrown indicating this. Otherwise, NIL is returned and no error is thrown. You can override this function to do what you wish. It should return NIL if the object is found to be "sane" and an throw an error otherwise. If check-sanity? is set to T in this object, this function will be invoked automatically within an ignore-errors by the function handling the GWL "/answer" form action URI when this object is a respondent, before the main-sheet is presented.

Process-cookies! *Void*

This is an empty function by default, but can be overridden in a user specialization of base-html-sheet, to do some processing before the header-plist is evaluated and before the HTTP response is actually initiated, but after the cookies-received have been set.

Restore-form-controls! *Void*

Calls restore-defaults! on all the form-controls in this sheet.

Sanity-error *Void*

Emits a page explaining the sanity error. This will be invoked instead of the write-main-sheet if check-sanity? is set to T and the check-sanity throws an error. You may override this function to do what you wish. By default a minimal error message is displayed and a link to the root object is presented.

Select-choices *Void*

Writes an HTML Select field with Options.

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.

- **GEOMETRY-VIEW-MIXIN**

  Mixins: VANILLA-MIXIN

  Input slots (optional):

Length *Number*

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

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`.

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):

```
...

:objects
((menu-1 :type 'menu-form-control
         :choice-plist (list 1 "one" 2 "two")))

...
```

Figure 11.60: Example Code for MENU-FORM-CONTROL

Choice-list *List*

Display values, also used as return values, for selection list. Specify this or choice-plist, not both.

Choice-plist *Plist*

Keywords and display values for the selection list. Specify this or choice-list, not both.

Choice-styles *Plist*

Keywords and CSS style for display of each choice. The keys should correspond to the keys in choice-plist, or the items in choice-list if no choice-plist is given.

Disabled-keys *List of keyword symbols*

Each of these should match a key in the choice-plist, and where there is a match, that key will be disabled in the rendering.

Multiple? *Boolean*

Are multiple selections allowed? Default is nil.

Possible-nil? *Boolean*

Indicates whether this should be included in possible-nils. Defaults to (the multiple?)

Size *Number*

How many choices to display

Test *Predicate function of two arguments*

Defaults based on type of first in choice-plist: eql for keywords, string-equal for strings, and equalp otherwise.

- **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

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.

```
FLAG -- fill in!!!
```

Figure 11.61: Example Code for SHEET-SECTION

- **SKELETON-FORM-CONTROL**

  Mixins: SKELETON-UI-ELEMENT, VANILLA-MIXIN

  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.

  Primary? *Boolean*

  Set this to t if the form-control should always occur first in an outputted snapshot file. Defaults to nil.

  Computed slots:

  Field-name *Keyword symbol*

  The name of this field. Computed from the object name within the tree.

  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 11.62: Example Code for SKELETON-UI-ELEMENT

Id *Keyword symbol*

The ID attribute for this tag. Defaults to (the field-name).

- **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.

  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.

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.

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.

This should be used with (str .) [in cl-who] or (:princ .) [in htmlGen] to output this section of the page, including the wrapping :div tag.

Ordered-form-controls *List of GDL objects, which should be of type 'base-form-control*

.

[Note – this slot is not really necessary for protecting out-of-bounds sequence references anymore, the form-control processor protects against this by itself now].

These objects are validated and bashed first, in the order given. If the cardinality of one form-control depends on another as in the example below, then you should list those dependent objects first. Default is nil.

Possible-nils *List of keyword symbols*

Messages corresponding to form fields which could be missing from form submission (e.g. checkbox fields). Defaults to the names of any children or hidden-children of type menu-form-control or checkbox-form-control.

Preset-all? *Boolean*

This switch determines whether all form-controls should be preset before the final setting, in order to allow any interdependencies to be detected for validation or detecting changed values. If this is specified as a non-nil value, then any nil values of (the preset?) on individual form controls will be ignored. If this is specified as nil, then (the preset?) of individual form-controls (default of these is also nil) will be respected. Default is nil.

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.

11.6.2 Function and Macro Definitions

- **BASE64-DECODE-LIST**
- **BASE64-DECODE-SAFE**
- **BASE64-ENCODE-LIST**
- **BASE64-ENCODE-SAFE**
- **CLEAR-ALL-INSTANCES**
- **CLEAR-INSTANCE**
- **CLEAR-OLD-TIMERS**
- **GWL-MAKE-OBJECT**
- **PUBLISH-GWL-APP**
- **PUBLISH-SHARED**
- **PUBLISH-STRING-CONTENT**
- **RELATIVIZE-PATHNAME**
- **SESSION-CONTROL-AUTO-REFRESH**
- **SESSION-REPORT**

- **WITH-CL-WHO** [Macro]
- **WITH-CL-WHO-STRING** [Macro]
- **WITH-HTML-FORM** [Macro]

11.6.3 Variables and Constants

- ***BREAK-ON-SET-SELF?***
- ***BYPASS-SECURITY-CHECK?***
- ***DEVELOPING?***
- ***ENT***
- ***FAILED-REQUEST-URL***
- ***INSTANCE-HASH-TABLE***
- ***JUMP-TO-TOPLEVEL-ON-SET-SELF?***
- ***MAX-ID-VALUE***
- ***QUERY***
- ***REAP-EXPIRED-SESSIONS?***
- ***RECOVERY-URL-DEFAULT***
- ***REQ***

11.7 RAPHAEL

11.8 ROBOT (Simplified Android Robot example)

11.9 SURF (NURBS Surface and Solids Geometry Primitives)

11.10 TASTY (Web-based Development Environment (tasty))

11.11 TREE (Tree component used by Tasty and potentially as a UI component on its own)

11.11.1 Object Definitions

- **NEWERTREE**

  Mixins: SHEET-SECTION

  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

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

11.12 YADD (Yet Another Definition Documenter (yadd))

11.12.1 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-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. When jquery is loaded (by setting the input-slot (use-jquery t)), the javascript can use the '$([selector])' shortcuts to do the magic. The javascript is automagically wrapped in the appropriate html tags to ensure a good execution by the javascript engine. When the use-jquery slot is true (t) than the javascript is wrapped in a '$(document).ready' function as a Good Practice#0153;.

  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