User's Guide
to
SIPP - a 3D rendering library

version 3.1

Jonas Yngvesson
Inge Wallin

last updated 20 April 1994

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.
5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is

copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Applying These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 1, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program ‘Gnomovision’ (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That’s all there is to it!

1 What is SIPP?

SIPP is a library for creating 3-dimensional scenes and rendering them using a scan-line z-buffer algorithm. A scene is built up of objects which can be transformed with rotation, translation and scaling. The objects form hierarchies where each object can have arbitrarily many subobjects and subsurfaces. A surface is a number of connected polygons which are rendered with either Phong, Gouraud or flat shading. An image can also be rendered as a line drawing of the polygon edges without any shading at all.

The library also provides 3-dimensional texture mapping with automatic interpolation of texture coordinates. Simple anti-aliasing can be performed through oversampling. The scene can be illuminated by an arbitrary number of lightsources. These lightsources can be of three basic types: directional, point or spotlight. Light from spotlights can cast shadows.

It is possible to create several virtual cameras, and then specify one of them to use when rendering the image.

A major feature in SIPP is the ability for a user to provide his own shading function for a surface. This makes it easy to experiment with various shading models and to do special effects. A basic shading algorithm is provided with the library, and also a package of other, more special shaders.

Images can be rendered directly onto a file in the Portable Pixmap format (ppm) or, for line images, Portable Bitmap, (pbm) or, with a function defined by the user, into anything that it is capable of plotting a pixel (or drawing a line), e.g. a window in a window system or even a plotter file.

The object creation functions in SIPP are on a rather low level so to make it easier to build scenes, a set of object primitives, like sphere, cylinder, prism etc., is included.

1.1 Authors of SIPP

The following persons have written or contributed to SIPP.

- Jonas Yngvesson wrote most of the otherwise unattributed functions in SIPP as well as most of the documentation.

- Inge Wallin wrote the geometric functions, some object primitives, pixmap functions, several demonstration programs and the rest of the documentation.
- Mark Diekhans created the new object and surface structures and suggested most of the new memory management support. He also wrote the support for changing rendering direction, the callback function and rendering abortion.
- David Jones provided the code for the prism and cone primitives.
- Jon Buller wrote the noise and Dnoise functions (not specifically for SIPP though, they were posted on the net).
- Ray P. Bellis made the support for shared libraries under SunOS.
- Several other people have aided the development by reporting bugs, suggested enhancements, etc. etc. I will not mention any names, you know who you are.

1.2 Where can I get SIPP?

There will probably be a number of sites archiving SIPP. Currently the latest release can always be fetched via anonymous ftp from `isy.liu.se`, (IP no. 130.236.1.3) in the directory `pub/sipp`.

Two older versions (2.0 and 2.1) have been posted to comp.sources.misc. They are in Volume 16 and Volume 21 respectively and should be on any site that archives that group.

2 Installation

This section describes the installation of the SIPP rendering library. You should install not only the library itself, but also the on-line documentation so that your users will know how to use it. You can create typeset documentation from the file '`sipp.texinfo`' as well as an on-line Info file. The following steps are also described in the file '`INSTALL`' in the directory '`sipp-3.1`'.

2.1 Installation of the SIPP library

Edit the file '`Makefile`' to reflect the situation at your site. The things you might have to change are clearly marked in the beginning of that file. They are also described below.

- The `NOVOID` definition should be used if the C compiler on your system does not understand the type `void`.
- If the C library on your system does not contain the functions `memcpy()` or `memset()`, or the include file '`memory.h`' does not exist, you should use the `NOMEMCPY` definition.
- If your system does not support the `alloca()` function, the `ALLOCA` definition should be used. This will cause SIPP to use the portable version of `alloca()` available from the GNU project.
- The definitions of `LIBDIR`, `INCLUDEDIR`, `MANDIR` and `MANEXT` determines where in your file hierarchy SIPP will be installed. `LIBDIR` is the directory where the final library file ('`libsipp.a`') will be placed. When a program that uses SIPP is linked, this directory should be in the path where the linker looks for libraries, either direct or with the aid of the `-L` switch. `INCLUDEDIR` is the directory where the include files necessary to use SIPP will be placed. When a program that uses SIPP is compiled, this directory should be in the path where the compiler searches for include files, either direct or with the aid of the `-I` switch. `MANDIR` is the directory in which to place the UNIX style '`man`' page provided with SIPP. `MANEXT` determines what extension that manual file will get.
- If you are on a Sun you can create SIPP as a shared library (it is possible that this work for other systems too, but we have no way of testing that). Uncomment the definitions of `LIBSH` and `LIBHINST` and choose a set of C-compiler flags to match your setup.

Apart from these SIPP specific definitions, the usual C compiler and flags to this compiler must of course be set to values suitable on your system.

The only other item, apart from the '`Makefile`', is a definition in the include-file '`sipp.h`' in the '`libsipp`' directory. In this file a macro called `RANDOM()` is defined. If your system does not have the `drand48()` function, you must change this definition. The macro should return a random floating point number in the range (-1, 1).

By just typing `make` in the '`sipp-3.1`' directory, the library and the demonstration programs will be compiled. The library is not installed, but only compiled in place.

By typing `make library`, the library will be compiled in place but the demonstration programs will not.

Typing `make demos` will compile the demonstration programs only. Since the demos require it, however, the library will also be compiled if it was not done before.

Finally, typing `make install` will compile the library if it was not done before, and copy that, the include files and the manual pages to their appropriate places.

2.2 Installation of the on-line Info manual.

1. Create the Info files '`sipp`', '`sipp-1`', '`sipp-2`' and so on from '`sipp.texinfo`'. If you have the `makeinfo` program, you can do this by running it on '`sipp.texinfo`'. Otherwise you can do it with emacs by running these steps:
   1. Read '`sipp.texinfo`' into an emacs buffer.
   2. Type '`M-X texinfo-format-buffer`'
   3. Save the newly created Info file '`sipp`', '`sipp-1`', '`sipp-2`' and so on .
2. Move the Info file '`sipp`', '`sipp-1`', '`sipp-2`' and so on to the standard Info directory. Usually this is '`/usr/gnu/emacs/info`' or something similar. (See step 3 above).
3. Edit the file '`dir`' in the info directory and enter one line to contain a pointer to the Info file '`sipp`'. The line can, for instance, look like this:

```
* SIPP: (sipp).       3D rendering library.
```

2.3 How to make typeset documentation from sipp.texinfo

You can also make a typeset manual from the file '`sipp.texinfo`'. To do this, you must have the TeX text formatting program installed. Just follow these steps:

1. If the file '`texinfo.tex`' is not properly installed in the path given by the environment variable *TEXINPUTS*, get it and put it in the same directory as '`sipp.texinfo`' (the '`doc`' directory of

SIPP). This file contains macros used by the TEX text formatting program to produce typeset output from a texinfo file. You can get this from, e.g., `prep.ai.mit.edu` in the US or from `isy.liu.se` in Europe.

2. Run TEX by typing '`tex sipp.texinfo`'. You might need to do this twice to get all cross references correct. If you have the `texindex` program, you can create a sorted index by typing '`texindex sipp.cp sipp.fn`' between the two TEX passes. If you don't do this, you still get a typeset manual, but you will not get the index.
3. Convert the resulting device independent file '`sipp.dvi`' to a form which your printer can output and print it. If you have a postscript printer there is a program, `dvi2ps`, which can do this. There is also a program which comes together with TEX, `dvips`, which you can use.

3 Getting started

This chapter will be a small introduction of SIPP. We will go through the steps of creating a simple scene and then enhance it with some special effects. No specific details about the functions we use will be explained, they can be found in other parts of this manual.

The first two things in a program using SIPP should be inclusion of '`sipp.h`' and a call to `sipp_init()`. Then we can start using the functions in SIPP to create a scene:

```
#include <stdio.h>

#include <sipp.h>
#include <primitives.h>

main()
{
        FILE     *image_fd;

        Object   *sphere;
        Surf_desc sphere_surface;


        sipp_init();

        sphere_surface.ambient = 0.4;
        sphere_surface.specular = 0.6;
        sphere_surface.c3 = 0.1;
        sphere_surface.color.red = 0.70; /* firebrick red */
        sphere_surface.color.grn = 0.13;
        sphere_surface.color.blu = 0.13;
        sphere_surface.opacity.red = 1.0; /* Totally opaque */
        sphere_surface.opacity.grn = 1.0;
        sphere_surface.opacity.blu = 1.0;

        sphere = sipp_sphere(2.0, 40, &sphere_surface, basic_shader, WORLD);
        object_add_subobj(sipp_world, sphere);

        lightsource_create(1.0, 1.0, 1.0,  1.0, 1.0, 1.0,  LIGHT_DIRECTION);

        camera_params(sipp_camera, 0.0, 10.0, 0.0,
                      0.0, 0.0, 0.0,  0.0, 0.0, 1.0,  0.4);

        image_fd = fopen("ex1.ppm", "w");
        render_image_file(400, 400, image_fd, PHONG, 1);
}
```

If the program is stored in a file called 'ex1.c' we can create an executable program with the following command line:

```
cc -o ex1 ex1.c -lsipp -lm
```

When run, the program will create a PPM-file containing a 400x400 image of a red sphere lit by a single lightsource.

In the program we are going through the following steps: First we initialize the library with a call to `sipp_init()`. Next we fill in a description of surface properties in the kind of structure used in SIPP's basic internal shader. We then create a sphere that will be shaded with the basic shader using the previously defined surface properties and tell SIPP to install this sphere among the objects that should be considered when rendering. We create a lightsource and define where the camera is and where it is looking. Last we open a file and tell SIPP to render the scene into that file.

3.1 Enhancing the scene

A single red sphere is not a very exciting image so we will now enhance the image with some more interesting effects. We will put a wooden floor under the sphere and exchange the lightsource for a spotlight that will cast a shadow of the sphere onto the floor. The floor is created as a simple block and we use the wood shader supplied in the library. There will be rather high frequencies in the wood pattern so we will render the image with some oversampling to make it look better. The code looks like this:

```
#include <stdio.h>

#include <sipp.h>
#include <primitives.h>
#include <shaders.h>

main()
{
        FILE     *image_fd;

        Object   *sphere;
        Object   *floor;
        Surf_desc sphere_surface;
        Wood_desc floor_surface;
```

```
        sipp_init();
        sipp_shadows(TRUE, 600);

        sphere_surface.ambient = 0.5;
        sphere_surface.specular = 0.6;
        sphere_surface.c3 = 0.1;
        sphere_surface.color.red = 0.70; /* firebrick red */
        sphere_surface.color.grn = 0.13;
        sphere_surface.color.blu = 0.13;
        sphere_surface.opacity.red = 1.0; /* Totally opaque */
        sphere_surface.opacity.grn = 1.0;
        sphere_surface.opacity.blu = 1.0;

        sphere = sipp_sphere(2.0, 40, &sphere_surface, basic_shader, WORLD);
        object_add_subobj(sipp_world, sphere);

        floor_surface.ambient = 0.5;
        floor_surface.specular = 0.0;
        floor_surface.c3 = 0.99;
        floor_surface.scale = 3.0;
        floor_surface.base.red = 0.770; /* Very light brown */
        floor_surface.base.grn = 0.568;
        floor_surface.base.blu = 0.405;
        floor_surface.ring.red = 0.468; /* Darker brown */
        floor_surface.ring.grn = 0.296;
        floor_surface.ring.blu = 0.156;

        floor = sipp_block(20.0, 20.0, 1.0, &floor_surface, wood_shader,
                           WORLD);
        object_move(floor, 0.0, 0.0, -2.5); /* Place it under the sphere */
        object_add_subobj(sipp_world, floor);

        spotlight_create(10.0, 10.0, 10.0,  0.0, 0.0, 0.0,  40.0,
                         1.0, 1.0, 1.0,  SPOT_SOFT,  TRUE);

        camera_params(sipp_camera, 0.0, 10.0, 0.0,
                      0.0, 0.0, 0.0,  0.0, 0.0, 1.0,  0.4);

        image_fd = fopen("ex2.ppm", "w");
        render_image_file(400, 400, image_fd, PHONG, 2);
}
```

4 Basic concepts

This chapter introduces and briefly explains some of the basic concepts used in SIPP. They will later be used in this manual without further explanation.

4.1 Polygons

SIPP can actually only render polygons, so everything else must be built from those. SIPP can handle planar, convex or concave, polygons without holes. The polygons have a defined front and back side, and which is which is defined by the order in which the polygon vertices are given. Vertices must be given counterclockwise when looking at the front side of the polygon.

4.2 Surfaces

Surfaces are the first step above polygons in the object hierarchy supported by SIPP. A surface is a collection of polygons that is shaded by the same shader (See Section 4.5 [Shading functions], page 16) using the same surface description (See Section 4.6 [Surface descriptions], page 16). A pointer to that shader and surface description is stored in the surface. If polygons within a surface share vertices, the surface normal will be interpolated across the polygons at rendering time, to create the impression of a smooth surface.

4.3 Objects

Objects are the highest level in the object hierarchy. An object is a collection of surfaces and/or other objects, which are then called subobjects. Object trees can be built to arbitrary depths. Transformations can be applied to objects and if an object has subobjects the transformation will propagate recursively down the object tree. Every object has its current transformation relative to its parent object stored in a transformation matrix which can be read and written.

There is a predefined object called `sipp_world`. When SIPP renders a scene it always starts in this object, so all objects that are to be rendered must be subobjects (or subsubobjects etc.) to it. The world object can be transformed like any other object.

4.4 Texture coordinates

At each polygon vertex it is possible to specify up to three floating point numbers called texture coordinates. These numbers are linearly interpolated across the polygon and sent to the shader (See Section 4.5 [Shading functions], page 16) at rendering time. It is up to the implementor of the shader to decide what to use them for. Texture coordinates are not affected by object transformations.

4.5 Shading functions

Every surface (See Section 4.2 [Surfaces], page 15) in a scene has a shading function (or shader) associated with it. The shader is a regular C function, with a well defined interface, which is called for every pixel in the surface when it is rendered. SIPP supplies the shader with enough information for it to do a shading calculation, i.e. decide what color that particular pixel should have. The shader is also responsible for deciding the opacity of the surface. Besides the information supplied by SIPP (world position, lightsources, texture coordinates, etc.), the shader also gets a surface description (See Section 4.6 [Surface descriptions], page 16) which the user has defined.

4.6 Surface descriptions

Every surface (See Section 4.2 [Surfaces], page 15) has a description of its surface properties. These properties can be e.g. color, material, opacity, etc. Exactly what information is stored depends on which shader (See Section 4.5 [Shading functions], page 16) is used for shading the surface. The exact representation of this information is entirely up to the shader implementor.

4.7 Datatypes

The include file '`sipp.h`' defines several datatypes that are used when working with SIPP. We will describe them briefly here and also give the definitions for those that a user might need to access.

- `bool`

  A boolean type which can have the value `TRUE` or `FALSE`.

- `Object`

  This is an abstract data type holding information about an object. Functions that creates objects returns pointers to `Object` and all functions that operate on objects, e.g. transformations, take such pointers as parameters.

- `Surface`

  Similar to `Object` but contains information about a surface. The user only needs to handle pointers to this type also.

- `Color`

  This is a structure with three members describing a color in RGB-space. Each member is a double and should have a value in the range [0, 1].

```
typedef struct {
        double   red;
        double   grn;
        double   blu;
} Color;
```

- `Vector`

  Structure defining a 3-D vector. See Section 8.1.1 [Vector operations], page 35 for more detailed information.

```
typedef struct {
        double x;
        double y;
        double z;
} Vector;
```

- `Transf_mat`

  A transformation matrix is used in every object to hold its current transformation. The matrix is stored as a 4x3 matrix instead of a complete 4x4 matrix in order to save space. See Section 8.1.2 [Matrix operations], page 36 for more detailed information.

```
typedef struct {
        double   mat[4][3];
} Transf_mat;
```

- `Camera`

  `Camera` is a structure holding a virtual camera. All functions involved work with pointers to this type. SIPP provides a predefined `Camera` and a pointer to it called `sipp_camera`. This camera is the default viewpoint used when rendering a scene.

- `Lightsource`

  This structure hold information about a lightsource. Two members in the struct are of interest to users writing their own shaders.

```
Color         color;
```

  and

```
Lightsource  *next;
```

color decides the color of the light emitted from the lightsource and next points to the next lightsource defined in the scene (or NULL). See Section 14.2 [Writing your own shaders], page 69 for a description of how to use this information.

- Surf_desc

  This is the surface description (See Section 4.6 [Surface descriptions], page 16) for the internal shader, basic_shader() (see Section 14.1.1 [The basic shader], page 62). It has the following definition:

```
typedef struct {
        double  ambient;
        double  specular;
        double  c3;
        Color   color;
        Color   opacity;
} Surf_desc;
```

  ambient is a number in the range [0, 1] specifying how much of the surface color that is visible when the object is not lit by any lightsource.

  specular is a number in the range [0, 1] specifying how much light that is reflected in a specular highlight on the surface.

  c3 is also a number in the range [0, 1]. It specifies how "shiny" the surface is. 0 means a very shiny surface while 1 indicates a rather dull one.

  color is simply the color of the surface.

  opacity specifies how opaque the surface is. This is stored as a color to allow different opacities for the different color bands. The values should be in the range [0, 1] with 1 indicating a completely opaque object and 0 a completely transparent (invisible) one.

- Surf_desc_hdr

  Header used for memory management of dynamically allocated surface descriptions. See Section 7.2 [Surface description memory], page 33

```
typedef struct {
        int     ref_count;
        void  (*free_func)();
        void   *client_data;
} Surf_desc_hdr;
```

  ref_count is the reference counter for the surface description. This is for internal use by SIPP only!

  free_func point to a function which will be called to release the memory used by the header and the surface description when there are no more references to them.

  client_data is a generic pointer where the user can store references to anything the free_func might need.

5 Initializations

Before using any of the functions, SIPP needs to be initialized. Initialization is done with a call to the following function:

```
void
sipp_init()
```

Apart from initializations, some default settings are created:

- Backfacing polygons are culled.
- Background color is black.
- No shadows are cast.
- The camera is placed in (0 0 10), looking at the origin and with the world y-axis as the up-axis.
- Rendering direction is top to bottom.

`sipp_init()` takes no parameters

There are also some functions that determine various global behavior of SIPP. These functions can be called at any time:

```
void
sipp_background(red, green, blue)
        double  red;
        double  green;
        double  blue;
```

This function sets the background color in the rendered image. The parameters are doubles in the range [0, 1]. The default value (set by `sipp_init()`) is black.

```
void
sipp_show_backfaces(flag)
        bool  flag;
```

Normally SIPP checks if a polygon is facing away from the viewpoint and if that is the case, the polygon is not considered in the rendering. There are times when this is not desirable. If one have a database of polygons with inconsistent orientations (see Section 4.1 [Polygons], page 15), it is necessary to render all polygons in it. There are also cases when objects have holes and backfacing polygons are visible through that hole. If `flag` is `TRUE` SIPP will render all polygons, if `flag` is `FALSE` (default), backfacing polygons will be culled.

```
void
sipp_shadows(flag, size)
        bool  flag;
        int   size;
```

This function tells SIPP if objects should cast shadows. When `flag` is `TRUE` shadows are cast. The default is not to do it. Only some types of lightsources are capable of producing shadows and it is possible to turn that ability on and off for each such lightsource (See Chapter 10 [Lights], page 44). SIPP uses a technique called *depth maps* to do shadows (See Section 11.1 [Shadows], page 50). It is a kind of texture mapping and the size of the depth maps are defined by the parameter `size`. As a rule of thumb one could say that the depth maps should be at least as large as the image itself but this may vary from case to case.

When `sipp_shadows()` have been called with argument `TRUE`, depthmaps will automatically be created at the start of a rendering and deleted at the end of it. Sometimes this behaviour is undesirable, e.g. when doing an animated walk-trough of a scene one would like to render the depthmaps only once and then use them when rendering each frame without having to regenerate them each time. This is supported through the functions `shadowmaps_create()` and `shadowmaps_destruct()` (See Section 11.1 [Shadows], page 50) which will explicitly create depthmaps and keep them around until they are explicitly destructed.

A word of warning: Rendering images with shadows requires very large amounts of memory and takes considerably longer time than doing it without them.

```
sipp_render_direction(direction)
        bool direction;
```

Some image formats (notably Utah RLE) store images with the scanlines in the order bottom to top. The default file format in SIPP (‘`ppm`’) stores the image with the scanlines in the order top to bottom. This function decides in which order SIPP should render the scanlines. The `direction` argument can have the values `TOP_TO_BOTTOM` or `BOTTOM_TO_TOP`.

```
sipp_set_update_callback(proc, client_data, period)
        void  (*proc)();
        void   *client_data;
        int     period;
```

If SIPP is used in a more interactive application, for instance under a window system, it is very useful for the application to gain control sometimes during a rendering. Perhaps one need to check for window system events or something similar. This function registers a callback function which SIPP will call at regular intervals during a rendering. The callback function will be called with a single argument: the `client_data` pointer. How often the function will be called is controlled with the `period` argument. The unit for this period is (very approximately) the time it takes to render one pixel.

A useful capability for the callback function is to be able to terminate the ongoing rendering in a reasonable way. This is supported through the function `sipp_render_terminate` (See Chapter 13 [Rendering], page 54).

The next two functions are used to control some properties of the memory management in SIPP. They are probably only of interest if you develop fairly advanced applications on top of SIPP, See Chapter 7 [Memory management], page 30.

```
sipp_user_refcount(flag)
        bool  flag;
```

Selects if user references to surfaces and objects should be counted. If `flag` is `TRUE` they are counted otherwise not. See Section 7.1 [Objects and surfaces], page 30 for a detailed description of how this affect things.

```
bool
sipp_surface_desc_headers(flag)
        bool  flag;
```

This function indicates if surface descriptions are equipped with `Surf_desc_hdr` headers. If `flag` is `TRUE` they are otherwise not. The function returns the old value of the flag. For a more detailed description of how to use these headers: See Section 7.2 [Surface description memory], page 33

```
bool
sipp_ffd_desc_headers(flag)
        bool  flag;
```

Currently, in the experimental version, memory management for free form deformation descriptions is handled in the same way as for surface descriptions, even using the same optional header structure (`Surf_desc_hdr`).

This function indicates if free form deformation descriptions are equipped with `Surf_desc_hdr` headers. If `flag` is `TRUE` they are otherwise not. The function returns the old value of the flag. For a more detailed description of how to use these headers: See Section 7.2 [Surface description memory], page 33

6 Creating objects

This chapter describes how to build SIPP objects from polygons and up. In the library there are also a number of functions that create complete objects on a higher level (see Chapter 15 [Object primitives], page 72). Those functions all use the low level tools described here.

6.1 Creating polygons and surfaces

To build polygons and surfaces, SIPP uses two stacks, a vertex stack and a polygon stack. Polygons are created by pushing vertices onto the vertex stack and then calling a function that creates a polygon from these vertices and push this newly created polygon onto the polygon stack. When a number of polygons have been defined they can then be combined into a surface.

The order in which vertices are pushed are important because this determines the front and the back face of the polygon. Vertices should be pushed in *counterclockwise* order when looking at the front face of the polygon.

Note also that if polygons share vertices, these vertices should be pushed for each polygon. SIPP looks up shared vertices automagically.

The following functions are used in the described process:

```
void
vertex_push(x, y, z)
        double  x, y, z;
```

Push a vertex onto the vertex stack.

```
void
vertex_tx_push(x, y, z,  u, v, w)
        double  x, y, z;
        double  u, v, w;
```

Push a vertex with texture coordinates defined by (u, v, w) onto the vertex stack. Calls to `vertex_push()` and `vertex_tx_push()` should not be mixed within a polygon since that would make texture interpolation to produce garbage. `vertex_push()` gives the vertex texture coordinates (0 0 0).

```
void
vertex_n_push(x, y, z,  nx, ny, nz)
        double  x, y, z;
        double  nx, ny, nz;
```

Push a vertex with a specified normal vector defined by (nx, ny, nz) onto the vertex stack. `vertex_n_push()` gives the vertex texture coordinates (0 0 0).

```
void
vertex_n_tx_push(x, y, z,  u, v, w,  nx, ny, nz)
        double  x, y, z;
        double  u, v, w;
        double  nx, ny, nz;
```

Push a vertex with texture coordinates defined by (u, v, w) and a specified normal vector defined by (nx, ny, nz) onto the vertex stack. The same consideration about mixing vertices with and without texture coordinates within a polygon as in `vertex_tx_push()` applies here.

```
void
polygon_push()
```

Takes all vertices currently on the vertex stack and creates a polygon from them. The new polygon is pushed onto the polygon stack and the vertex stack is emptied.

```
Surface *
surface_basic_create(ambient, red, green, blue, specular, c3,
     opred, opgreen, opblue)
        double  ambient;
        double  red, green, blue;
        double  specular;
        double  c3;
        double  opred, opgreen, opblue;
```

Takes all polygons currently on the polygon stack, creates a surface from them and returns a pointer to the new surface. The created surface will be shaded with the basic shading function `basic_shader()` and the arguments to `surface_basic_create()` are the values that will be placed in the surface description, which for `basic_shader()` is of type `Surf_desc`(see Chapter 4 [Basic concepts], page 15 and Chapter 14 [Shaders], page 62).

```
Surface *
surface_create(surface_desc, shader)
        void    *surface_desc;
        Shader  *shader;
```

Takes all polygons currently on the polygon stack, creates a surface from them and returns a pointer to the new surface. The created surface will be shaded with the shading function `shader` using the surface description pointed to by `surface_desc` (see Chapter 14 [Shaders], page 62).

```
void
surface_unref(surface)
        Surface  *surface;
```

Tell SIPP that the reference `surface` is not needed anymore. If there are no other references to the structure `surface` pointed to, it will be deleted. Note that the behavior changes slightly dependent on the memory management approach selected with `sipp_user_refcount()`, See Section 7.1 [Objects and surfaces], page 30.

```
void
surface_basic_shader(surface, ambient, red, green, blue, specular, c3,
     opred, opgreen, opblue)
        Surface *surface;
        double   ambient;
        double   red, green, blue;
        double   specular;
        double   c3;
        double   opred, opgreen, opblue;
```

This function is used when a previously created surface should be changed so that it is shaded with `basic_shader()`. This function can also be used to set new values in the surface description if `surface` is already shaded with `basic_shader()`.

```
void
surface_set_shader(surface, surface_desc, shader)
        Surface *surface;
        void    *surface_desc;
        Shader  *shader;
```

This function is used when a previously created surface should be changed so that it is shaded with another shader than the one specified at creation time.

6.2 Building objects

An object in SIPP is a more abstract concept than surfaces and polygons. It is a general "container" which can hold several surfaces and also several other objects, which are then called subobjects. Such hierarchies, or trees, of objects can be built to arbitrary depths. When an object is transformed in some way, the transformation is propagated down to all objects below it in the tree.

When SIPP renders a scene it begins in the predefined object `sipp_world` and recursively traverses the tree under it, rendering all objects it finds. This means that it is perfectly possible to create objects that will not be rendered. For an object to be rendered it must be installed somewhere in the tree below `sipp_world`.

To build objects and object trees the following functions are provided:

```
Object *
object_create()
```

This function creates a new object and returns a pointer to it. The new object contains no surfaces or subobjects, and is not installed in any tree.

```
void
object_unref(object)
        Object  *object;
```

Tell SIPP that the reference `object` is not needed anymore. If there are no other references to the structure `object` pointed to, it will be deleted. SIPP keeps track of internal references though, and if some parts of the tree below `object` are referenced from other objects (see Section 6.3 [Duplicating objects], page 28), those parts are not deleted. Note that the behavior changes slightly dependent on the memory management approach selected with `sipp_user_refcount()`, See Section 7.1 [Objects and surfaces], page 30. It is not possible to delete `sipp_world`.

(This function used to be called `object_delete()` and that function is still available for compatibility reasons but will disappear in future versions.)

```
void
object_add_surface(object, surface)
        Object  *object;
        Surface *surface;
```

Install a surface in an object.

```
bool
object_sub_surface(object, surface)
        Object  *object;
        Surface *surface;
```

Remove a surface from an object. If `surface` was not found in `object`, `FALSE` is returned, otherwise `TRUE`.

```
void
object_add_subobj(object, subobject)
        Object  *object;
        Object  *subobject;
```

Install `subobject` as a subobject in `object`. Any transformations of `subobject` will now be performed relative the local coordinate system in `object`.

A word of warning: There is no detection of "circular lists" in SIPP. This means that if an object is installed as a subobject in an object that is already below it in the tree, SIPP will go into eternal recursion and crash when it tries to render the scene.

```
bool
object_sub_subobj(object, subobject)
        Object  *object;
        Object  *subobject;
```

Remove `subobject` as a subobject in `object`. If `subobject` was not found in `object`, `FALSE` is returned, otherwise `TRUE`.

6.3 Duplicating objects

If a complicated object has been built, it is often convenient to be able to copy and reuse it. SIPP supports three levels of copying object hierarchies:

```
Object *
object_instance(object)
        Object  *object;
```

Create a new instance of an object and return a pointer to it. This is the "shallowest" version of object copy in SIPP. It only creates a copy of the top level object, pointed to by `object`, and let the new instance reference the same surfaces and subobjects as the original. This saves space but has the property that if a subobject of one of the instances are changed in some way (transformed, new subobjects, etc.) the same change will appear in the other. The new object will have the identity matrix as its transformation matrix.

```
Object *
object_dup(object)
        Object  *object;
```

This version of object duplication copies not only the top level object, but also all the subobjects recursively. All copied objects in the tree will reference the same surfaces though, so even if object changes will be unique in the two copies, surface changes (new color, new shader, etc.) in one copy will affect both. The new object will have the identity matrix as its transformation matrix.

```
Object *
object_deep_dup(object)
        Object  *object;
```

Copy a complete object tree, objects, surfaces and all. The new object will have the identity matrix as its transformation matrix.

7 Memory management

This chapter mostly concerns users who develop more advanced applications on top of SIPP and need to worry about memory leaks and such. If you only use SIPP as a simple tool for rendering, like the demo programs (one program -> one scene) you can safely skip to the next chapter.

7.1 Objects and surfaces

Managing the memory used for surfaces and objects basically consist of choosing one of two approaches: either follow a few simple rules and then let SIPP handle it by itself or do more of it yourself and gain some consistency. The first approach is the default.

To illustrate the difference between the two modes we will give a small example. Consider the following piece of code:

```
Object *obj1, *obj2;

obj1 = object_create();
obj2 = object_create();
object_add_subobj(obj1, obj2);
...
```

When this is executed we get the following situation:

```
             +--------+
obj1--------->|  A     |
             |        |
             +--------+
                 |
                 V
             +--------+
obj2--------->|  B     |
             |        |
             +--------+
```

That is, the user has two pointers, `obj1` and `obj2`, and the object referenced by `obj1` (**A**) has an internal reference to the object referenced by `obj2` (**B**). All is well until the user decides to delete one of the objects, then we run into a conflict. If, for instance, the user calls `object_unref(obj2)`, what should we do? We can not free the memory occupied by **B** since **A** has a pointer to it and that would then point into freed memory. Obviously we should keep **B** around until nothing else references it and then delete it. The way to do this is to keep *reference counters* in the objects. When a reference is added to an object, for instance with `object_add_subobj()`, we increment the reference counter. When a reference is removed we decrement it.

Reference counters work great as long as we only need to consider the internal references between objects, since we have full control in the library over when they are created and removed (if the user is nice and only uses the library functions to manipulate the structures, of course). The problem is the reference which the user gets upon creation (`obj1` and `obj2` in the above example). Should they be counted or not?

If the users references are counted it is easy to create memory leaks. Suppose the above example had read:

```
Object *obj1, *obj2;

obj1 = object_create();
object_add_subobj(obj1, object_create());
...
```

There is no way for SIPP to know that the user never stored the result from the second `object_create()`, but sent it directly as an argument to a funtion. If the users references are counted, **B** would have a reference count of 2 while there is actually only 1 reference. Thus **B** will never be deleted.

If, on the other hand, users references are not counted we have a potential for memory corruption instead. Consider the example:

```
Object *obj1, *obj2;

obj1 = object_create();
obj2 = object_create();
object_add_subobj(obj1, obj2);
object_unref(obj2);
...
```

When `object_unref(obj2)` is called, **B** has only one reference (the users are not counted, remember), it is decremented and reach zero so **B** gets deleted. The internal reference in **A** now points to freed memory...

SIPP uses reference counters in the `Surface` and `Object` structures and let the user decide if users references shoud be counted or not. The default approach does not count them while the other does. Which approach that should be used is selected with an initial call to the function

```
void
sipp_user_refcount(flag)
        bool  flag;
```

If `flag` is `FALSE` the default approach is used, if `flag` is `TRUE` the users references will be counted.

IMPORTANT: Don't call this function more than once in an application! If some objects are created in one mode and some in the other the result is unpredictable!

So, how does this affect the user? Well, in the default approach the user must be careful with what he does with his references since SIPP is not "aware" of them. The simple rule to follow in the default approach is:
NEVER call `surface_unref()` or `object_unref()` on a surface or object that you have installed somewhere in SIPP (using `object_add_surface()` or `object_add_subobj()`). The only exception is if you have subsequently uninstalled it (using `object_sub_surface()` or `object_sub_subobj()`) and are positive that nothing else reference it.

In the other approach, where the users reference are counted, the user faces the risk of creating memory leaks instead of memory corruption. Therefore the user must always explicitly tell SIPP when he doesn't need his references any longer.

The rule here becomes:
ALWAYS call `surface_unref()` or `object_unref()` on a surface or object which you don't need your own reference to any longer.

Note that this rule makes the following code illegal:

```
Object *obj;

obj = object_create();
```

```
object_add_surface(obj, surface_create(data, shader));
...
```

This is because the users reference returned by `surface_create()` are discarded without `surface_unref()` gets called on it.

7.2 Surface descriptions

Managing memory for surface descriptions is slightly more complicated than for objects and surfaces. The reason is that SIPP has no control of the data structure used in a surface description, or how they are created. This is true also for the experimantal free form deformation descriptions, and exactly the same approach is used to manage memory for these as for the surface descriptions.

If you are only using statically allocated surface descriptions you will have no problems, but if you are using dynamic allocation you need a way to get the memory released when it is no longer needed. To support this SIPP allows you to add a header to your surface description structures. This header is of type `Surf_desc_hdr` and is defined in '`sipp.h`' (See Section 4.7 [Datatypes], page 16. The header contains a reference counter and a pointer to a function which will be called when it should be deleted. It also contains a generic data pointer which can be used to store any extra data the releasing function might need.

The releasing function will be called with a pointer to the header as a single argument.

When you create a surface with `surface_create()` (or installing a new shader in an existing surface with `surface_set_shader()`) SIPP must know if the pointer you send it is a direct pointer to the surface description, or a pointer to a `Surf_desc_hdr`-header immediately preceding it. This is selected by calling:

```
bool
sipp_surface_desc_headers(flag)
        bool  flag;
```

If `flag` is `TRUE` the surface descriptions installed from now on are assumed to have headers. This will be the case until the function is called with `flag` set to `FALSE`. The function returns the old value of the flag. Default is `FALSE`. You can mix surface descriptions with and without headers in an application without problem.

Note that if you are using "header-mode", it is the pointer to the *header* that should be sent to `surface_create()` and `surface_set_shader()`, not the pointer to the surface description itself.

If you are satisfied with the simple malloc-free pair for allocating the surface descriptions, SIPP has two macros which comes in handy when you want to use the described scheme with headers.

`SIPP_SURF_HDR_ALLOC(type)` allocates a contiguous piece of memory which contains an initial `Surf_desc_hdr`-header immediately followed by a data structure of type `type`. You have to fill in the `free_func` and `client_data` elements of the header yourself. When you have this piece of memory you need to get a pointer to your surface description in it. You can use the macro `SIPP_SURFP_HDR(type, hdr)` for that, here is an example:

```
Surf_desc_hdr    *surf_hdr;
My_pet_surf_desc *my_surf_desc;

surf_hdr = SIPP_SURF_HDR_ALLOC(My_pet_surf_desc);
surf_hdr->free_func = free;
surf_hdr->client_data = NULL; /* free() don't need anything here */
my_surf_desc = SIPP_SURFP_HDR(My_pet_surf_desc, surf_hdr);
```

If you are using something other than malloc-free, you need to do more work yourself. The best thing is of course if you can make your allocating function include the header in its allocation. If not you will need to fake it, for instance by using `malloc()` to allocate something of the same size as your surface description but with a header. Then you can copy the contents of your structure to the new one. You can store the old pointer in the `client_data` element in the header and then write your own releasing function which clean up this mess.

`SIPP_SURF_HDR_ALLOC()` is `malloc()`-specific but `SIPP_SURFP_HDR()` is general and works for any data structure headed by a `Surf_desc_hdr` header, no matter how it was created.

8 Transformations

All objects can be transformed with the usual homogeneous transformations: scaling, translation and rotation. The transformation is stored in a *transformation matrix* for each object. This matrix can also be read and written directly.

The same transformations that can be applied to objects can also be applied to the matrices directly. There is also a *vector* type defined and a number of operations defined on it.

8.1 Geometric operations

To use the vector and matrix functions and macros defined in the following section, you must include the following line into your program:

```
#include <geometric.h>
```

In geometric.h include file, all data types, macros and functions defined in this section are declared.

8.1.1 Vector operations

SIPP uses row vectors and not column vectors. A vector is defined as follows:

```
typedef struct {
    double   x;
    double   y;
    double   z;
} Vector;
```

This vector type is used both for directional vectors and points positional vectors. In the description below, lower case letters denote scalar values and upper case letters denote vectors. All operations are macros except the last one, `vecnorm()`.

`MakeVector(V, xx, yy, zz)`
: Put `xx`, `yy` and `zz` in the `x`, `y` and `z` slot of the Vector `V` respectively.

`VecNegate(A)`
: Negate all components of the Vector `A`.

`VecDot(A, B)`
: Return the dot product of the two Vectors `A` and `B`.

`VecLen(A)`
: Return the length of the Vector `A`.

`VecCopy(A, B)`
: Copy the Vector `B` to the Vector `A` (`A = B;` using C notation).

`VecAdd(C, A, B)`
: Add the two Vectors `A` and `B` and put the result in `C` (`C = A + B;` using C notation).

`VecSub(C, A, B)`
: Subtract the Vector `B` from Vector `A` and put the result in `C` (`C = A - B;` using C notation).

`VecScalMul(B, a, A)`
: Multiply the Vector `A` with the scalar `a` and put the result in Vector `B` (`B = a * A;` using C notation).

`VecAddS(C, a, A, B)`
: Multiply the Vector `A` with the scalar `a`, add it to Vector `B` and put the result in Vector `C` (`C = a * A + B;` using C notation).

`VecComb(C, a, A, b, B)`
: Linearly combine the two Vectors `A` and `B` and put the result in Vector `C` (`C` = `a * A + b * B;` using C notation).

`VecCross(C, A, B)`
: Cross multiply Vector `A` with Vector `B` and put the result in `C` (`C = A` X `B;`).

`void vecnorm(v)`
: `Vector *v;`

  Normalize the vector `v`, i.e. keep the direction but make it have length 1. The length of `v` should not be equal to 0 to begin with. **NOTE:** This is the only function operating on vectors in sipp. All the other operations are macros.

8.1.2 Matrix operations

An full homogeneous transformation matrix has 4 x 4 elements. However, all linear transformations use only 4 x 3 values so to save space a SIPP transformation matrix only store 4 x 3 values.

Also, if 4 x 4 matrices are used, all vectors must have 4 elements which we want to avoid. Thus the transformation matrix used in sipp is defined as follows:

```
typedef struct {
    double mat[4][3];
} Transf_mat;
```

We wrap a `struct` around the two-dimensional array since we want to be able to say things like `&mat` without being forced to write `(Transf_mat *) &mat[0]` which we find horrendously ugly.

SIPP has a predefined identity matrix declared in geometric.h which you can use:

```
extern Transf_mat   ident_matrix;
```

The rest of this section describes the macro and functions defined in the SIPP library which operate on SIPP transformation matrices.

`MatCopy(A, B)`

This macro copies the matrix `B` to the matrix `A`. `A` and `B` must both be pointers. **NOTE:** This is the only macro operating on matrices in SIPP. All other operations listed here are functions.

`Transf_mat *transf_mat_create(initmat)`

`Transf_mat *initmat;`

Allocate memory for a new transformation matrix and if `initmat` is equal to `NULL`, set the new matrix to the identity matrix. Otherwise set the new matrix to the contents of `initmat`. Return a pointer to the new matrix.

`Transf_mat *transf_mat_destruct(mat)`

`Transf_mat *initmat;`

Free the memory associated with the matrix `mat`.

`void mat_translate(mat, dx, dy, dz)`

`Transf_mat *mat;`
`double dx;`
`double dy;`
`double dz;`

Set `mat` to the transformation matrix that represents the concatenation of the previous transformation in `mat` and a translation along the vector (`dx`, `dy`, `dz`).

```
void mat_rotate_x(mat, ang)
```

```
Transf_mat *mat;
double ang;
```

Set mat to the transformation matrix that represents the concatenation of the previous transformation in mat and a rotation with the angle ang around the X axis. The angle ang is expressed in radians.

```
void mat_rotate_y(mat, ang)
```

```
Transf_mat *mat;
double ang;
```

Set mat to the transformation matrix that represents the concatenation of the previous transformation in mat and a rotation with the angle ang around the Y axis. The angle ang is expressed in radians.

```
void mat_rotate_z(mat, ang)
```

```
Transf_mat *mat;
double ang;
```

Set mat to the transformation matrix that represents the concatenation of the previous transformation in mat and a rotation with the angle ang around the Z axis. The angle ang is expressed in radians.

```
void mat_rotate(mat, point, vector, ang)
```

```
Transf_mat *mat;
Vector *point;
Vector *vector;
double ang;
```

Set mat to the transformation matrix that represents the concatenation of the previous transformation in mat and a rotation with the angle ang around the line represented by the point point and the vector vector. The angle ang is expressed in radians.

```
void mat_scale(mat, xscale, yscale, zscale)
```

```
Transf_mat *mat;
double xscale;
double yscale;
double zscale;
```

Set mat to the transformation matrix that represents the concatenation of the previous transformation in mat and a scaling with the scaling factors (xscale, yscale, zscale).

```
void mat_mirror_plane(mat, point, normal)
```

```
Transf_mat *mat;
Vector *point;
Vector *normal;
```

Set `mat` to the transformation matrix that represents the concatenation of the previous transformation in `mat` and a mirroring in the plane defined by the point `point` and the normal vector `normal`.

`void mat_mul(res, a, b)`

```
Transf_mat *res;
Transf_mat *a;
Transf_mat *b;
```

Multiply the two matrices `a` and `b` and put the result in the matrix `res`. All three parameters are pointers to matrices. It is possible for `res` to point at the same matrix as either `a` or `b` since the result is stored in a temporary matrix during the computations.

`void point_transform(res, vec, mat)`

```
Vector *res;
Vector *vec;
Transf_mat *mat;
```

Transform the point (vector) `vec` with the transformation matrix `mat` and put the result into the vector `res`. The two vectors `res` and `vec` should not be the same vector since no temporary is used during the computations.

8.2 Object transformations

There are two functions for reading and writing such matrices from and to objects:

```
Transf_mat *
object_get_transf(object, matrix)
        Object      *object;
        Transf_mat  *matrix;
```

This function retrieves the transformation matrix of the object pointed to by `object`. If `matrix` is `NULL` the function will allocate space for a matrix, copy the object's matrix into this space and return a pointer to the new matrix. If `matrix` is not `NULL` the objects transformation matrix is copied into the space its pointing to and the same pointer is returned.

```
void
object_set_transf(object, matrix)
        Object      *object;
        Transf_mat  *matrix;
```

This function copies the matrix pointed to by `matrix` into `object`'s transformation matrix.

There is also a special function for resetting an object's transformation matrix to the identity matrix, i.e. no transformation at all:

```
void
object_clear_transf(object)
        Object  *object;
```

8.2.1 Applying transformations

The transformations in this section are all applied to an object without altering its previous transformations, i.e. they will be applied after the previous transformations have been completed. What actually happens is that the matrix that specifies the new transformation is post multiplied into the objects current matrix.

There are four functions for rotating objects:

```
void
object_rot_x(object, angle)
        Object  *object;
        double   angle;

void
object_rot_y(object, angle)
        Object  *object;
        double   angle;

void
object_rot_z(object, angle)
        Object  *object;
        double   angle;

void
object_rot(object, point, vector, angle)
        Object  *object;
        Vector  *point;
        Vector  *vector;
        double   angle;
```

The first three functions rotate an object about one of the primary axes in the parent object's local coordinate system. `angle` is the rotation angle given in radians. Positive rotation is given by the "right hand rule", i.e. counterclockwise when looking along the axis towards the origin.

The fourth function is a more general rotation. It specifies a rotation of an object about an arbitrary axis. The axis is defined as passing through `point` in the direction of `vector`, both are described in the parent object's local coordinate system. `angle` is the rotation angle in radians and positive rotation is defined in the same way as for the previous three functions.

For scaling an object the following function is used:

```
void
object_scale(object, sx, sy, sz)
        Object  *object;
        double   sx, sy, sz;
```

The object pointed to by `object` is scaled towards the origin along the three principal axes with the three scaling factors `sx`, `sy` and `sz` respectively.

The last standard transformation is translation:

```
void
object_move(object, dx, dy, dz)
        Object  *object;
        double   dx, dy, dz;
```

The object is translated along the vector (`dx dy dz`) from its current position. Note that the movement is relative and not absolute. The translation vector is given in the parent coordinate system.

There is also a general transformation function that post-multiplies any transformation matrix into the current matrix of an object:

```
void
object_transform(object, matrix)
        Object      *object;
        Transf_mat  *matrix;
```

9 Deformations

There is a rudimantary (end experimental) support for doing free form deformations (ffd's or non affine transformations) of surfaces and textures in SIPP. This is implemented in a way similar to shading functions (See Chapter 14 [Shaders], page 62): In a surface the user can install a pointer to a function which does the deformation. SIPP calls this function for each vertex in the surface before the rendering starts. The function computes new values for the position and texture coordinates and SIPP will use these new values in the rendering. There is no restrictions on the new values but too radical changes most probably will cause very strange result. Most notably, if vertices are moved so that non planar polygons are created, the calculation of surface normals will get unpredictable results. Either be careful about what you do or build everything from triangles which are always planar.

The ffd funcion is called when the polygons are still in their original, untransformed state. This means that the coordinates available inside the function is exactly the same as those passed to the vertex_push()-type function used when the polygon was initially created.

Note also that the ffd function is called for each *vertex* and not for each pixel as the shaders. This means that coarsly tesselated surfaces will probably look strange if you start moving the vertices around. This is much less powerful than the RenderMan displacement- and transformation shaders but it is still possible to get interesting result if used with some care.

The deformation function should have the following interface:

```
void
my_ffd(ffd_data, world, texture, new_world, new_texture)
        void          *ffd_data;
        Vector        *position;
        Vector        *texture;
        Vector        *new_position;
        Vector        *new_texture;
```

`ffd_data` is a pointer to a data structure the user has installed in the surface (see below). This should contain any other data the user wants to pass to the deformation function.

`position` is the position in local object coordinates of the vertex that is being deformed.

`texture` contains the values of the texture coordinates at the vertex.

`new_position` is where the new value for the position should be placed upon return.

`new_texture` is where the new value for the texture coordinates should be placed upon return.

To install a deformation funtion in a surface you call this function:

```
void
surface_set_ffd(surface, ffd_func, ffd_data)
Surface  *surface;
void    (*ffd_func)();
void     *ffd_data;
```

`surface` points to the surface to which the deformation should be applied.

`ffd_func` is a pointer to a function with the interface described above.

`ffd_data` points to any data structure which the user want to pass to the deformation function. This pointer is passed unaltered as the first argument to `ffd_func`.

10 Lights

SIPP supports two basic kinds of lights, simple *lightsources* and *spotlights*. The main difference is that spotlights can cast shadows, while simple lightsources can not. The functions that create any of these lights return a pointer to a `Lightsource` structure. This pointer is used for later manipulations of the light such as moving it or turning it off or on. If there is no need for later manipulations these pointers can safely be discarded. SIPP keeps track of all created lightsources internally.

10.1 Creating lights

Simple lightsources can be of two types, directional and point lightsources. Directional lightsources emit light that is parallel in every point in the scene, similar to light from the sun. Point lightsources emit light from a single point in space.

```
Lightsource *
lightsource_create(x, y, z, red, green, blue, type)
        double x, y, z;
        double red, green, blue;
        int    type;
```

- `x, y, z`

  If a directional lightsource is created these numbers specifies a vector pointing to the lightsource. If it is a point lightsource the numbers specify the exact location of it.
- `red, green, blue`

  These numbers indicate the color of the emitted light. All three should be in the range [0, 1].
- `type`

  This parameter defines which type of lightsource that is created. It should be one of the predefined values `LIGHT_DIRECTION` or `LIGHT_POINT`.

A spotlight emits a "cone" of light. There are two types of spotlights in SIPP. One has a sharp edge on its lightcone and the other has a soft edge that blends out smoothly. Rendering scenes with soft edged spotlights takes slightly longer time than scenes with only sharp edged spotlights.

```
Lightsource *
spotlight_create(x1, y1, z1, x2, y2, z2, opening, red, green, blue,
```

```
                 type, shadow)
          double x1, y1, z1;
          double x2, y2, z2;
          double opening;
          double red, green, blue;
          int    type;
          bool   shadow;
```

- `x1, y1, z1`

  This is the position of the spotlight.

- `x2, y2, z2`

  This is a point at which the spotlight is pointing. It is in the middle of the lightcone.

- `opening`

  This defines, in degrees, the opening angle of the lightcone. The cone defined will be completely lit, a soft edged lightcone will start to blend out outside this angle.

- `red, green, blue`

  The color of the emitted light. All three numbers are in the range [0, 1].

- `type`

  Tells SIPP which type of spotlight to create. Should be one of the predefined values `SPOT_SHARP` or `SPOT_SOFT`.

- `shadow`

  If `TRUE`, the light from the spotlight will be able to cast shadows, otherwise not. Whether shadows actually are cast or not depend on which value `sipp_shadows()` (See Chapter 5 [Initializations], page 19) was called with last.

There is also a function for releasing the memory used by a lightsource or a spotlight.

```
void
light_destruct(light)
        Lightsource  *light;
```

- `light`

  Pointer to the lightsource or spotlight that is to be destructed.

10.2 Manipulating lights

When lights have been created they can be manipulated in various ways. There are functions that are specific for lightsources, functions specific for spotlights and generic functions which works for both kind of lights.

```
void
lightsource_put(lightsrc, x, y, z)
        Lightsource  *lightsrc;
        double        x, y, z;
```

This function is used to modify the direction, or position, of a lightsource. If (`x, y, z`) are interpreted as a position or as a direction vector depends on whether `lightsrc` is pointing at a point lightsource or a directional lightsource.

```
void
spotlight_pos(spot, x, y, z)
        Lightsource  *spot;
        double        x, y, z;
```

Modify the position of a spotlight.

```
void
spotlight_at(spot, x, y, z)
        Lightsource  *spot;
        double        x, y, z;
```

Modify the position the spotlight is pointing at.

```
void
spotlight_opening(spot, opening)
        Lightsource  *spot;
        double        opening;
```

Modify the opening angle of the lightcone of a spotlight. `opening` is given in degrees.

```
void
spotlight_shadows(spot, flag)
        Lightsource  *spot;
        bool          flag;
```

Turn shadow casting on or off for a specific spotlight. `flag` set to `TRUE` means that the spotlight can cast shadows.

```
void
light_color(light, red, green, blue)
        Lightsource  *light;
        double        red, green, blue;
```

Change the color of the emitted light from a lightsource or a spotlight. (`red, green, blue`) are all numbers in the range [0, 1].

```
void
light_active(light, flag)
        Lightsource  *light;
        bool          flag;
```

Turn a lightsource or a spotlight on or off. If `flag` is `TRUE` the light is activated.

The last function is not really a manipulation function. It evaluates how much light from a certain lightsource or spotlight that reaches a specific point in the scene. It also calculates a vector pointing from this point at the light. The return value is a number in the range [0, 1] where 1 means that all light from the lightsource reaches the point and 0 means that none of the light reaches it. The function is intended to be used in shading functions. We describe it formally here and refer to the chapter on how to write your own shaders for instructions and examples of how to use it (See Section 14.2 [Writing your own shaders], page 69).

```
double
light_eval(light, position, light_vector)
        Lightsource  *light;
        Vector       *position;
        Vector       *light_vector;
```

- `light`

Pointer to the lightsource or spotlight to evaluate.

- `position`

  Pointer to a vector specifying which point in the scene we want to check the illumination for. The position is given in the world coordinate system.

- `light_vector`

  Points to a space where `light_eval()` will store a normalized vector pointing from `position` at the light.

11 Shadows

SIPP creates shadows with a technique called *depth maps*. A detailed description of this technique can be found in the article *Rendering Antialiased Shadows with Depth Maps* by Reeves, Salesin and Cook in the Proceedings of SIGGRAPH 1987.

In principle, a depth map is generated for each light that should cast shadows. The depth map is simply an image of the scene, as seen from the light, but instead of a color we store the depth (Z-buffer value) in each "pixel". The finished map will contain the distance to the object closest to the light in each point.

When the scene is rendered we transform each point we are shading into depth map coordinates and if it is further away from the light than the value stored in the corresponding point in the depth map, the point is in shadow. The actual implementation is of course a bit more complicated with some sampling and filtering but we won't go into that.

The reason we describe this algorithm at all is that it is easier to understand how to get good looking shadows and why shadows sometimes look weird if one have an understanding of the underlying process.

First of all: The shadows are generated by sampling in the depth maps. Sampling usually means we are in danger of aliasing and this is very true in our case. SIPP automatically fits the depth map for a spotlight so that it covers all area lit by the spotlight's light cone (See Section 10.1 [Creating lights], page 44). If this area is large and the depth map resolution is low, the shadows will get very jagged.

Also, if we have a large surface that is close to perpendicular to the depth map plane, the depth map "pixels" will be projected as long stripes on that surface, so even if the depth map resolution is high, a shadow cast on such a surface will suffer from aliasing (be jagged).

So, if the edges of a shadow look weird, try increasing the size of the depth map (the depth map size is set with `sipp_shadows()`, See Chapter 5 [Initializations], page 19). If they still look weird, or you run out of memory, try changing the position of the lightsource that generate the shadow. After some tweaking it is usually possible to get fairly decent shadows.

11.1 Generating depth maps

The are two ways to generate depth maps in SIPP, either automatically for each new rendering, or explicitly on a command which will then keep the depth maps around until they are explicitly deleted.

To make SIPP automatically generate the depth maps for each new rendering, and delete them afterwards, call the function `sipp_shadows()` with the argument `TRUE` before starting a rendering (See Chapter 5 [Initializations], page 19).

To explicitly create the depth maps call the following function:

```
void
shadowmaps_create(size)
        int  size;
```

The argument `size` determines the size of the generated depth maps, they will be (`size x size`) "pixels". When a rendering is performed after this function has been called, the generated depth maps are used to create the shadows in the scene. The depth maps are *not* deleted automatically afterwards. This is very useful if a static scene is rendered several times, perhaps from different viewpoints, since the time to generate depth maps is only spent once.

Another usage is to generate the depth maps before some of the objects are installed in the scene, if it is certain that they will never cast a shadow on any other object (a floor being a typical example). The depth map generation will the be considerably faster since fewer objects need to be rendered. When the remaining objects are installed, the rendering can be started and shadows is still cast on them but not from them.

To delete the depth maps when they are not needed anymore, or the scene has changed too much to use the same depth maps, call the following function:

```
void
shadowmaps_destruct()
```

12 Viewpoint and cameras

The viewpoint model used in SIPP are a fairly standard one. A point where the camera is located, a point which the camera looks at, a vector telling which direction is up and the focal distance in the camera. The user can create several *virtual cameras* and tell SIPP to use any of them as viewpoint when rendering an image. There is also a predefined camera called `sipp_camera` which is the default viewpoint. When `sipp_init()` is called, this camera is initialized to be located in (0 0 10), looking at the origin, with the world y-axis as the up direction and a focal factor (see below) of 0.25. The user can of course change these values to whatever he likes.

To create and manipulate cameras, SIPP provide the following functions:

```
Camera *
camera_create()
```

This function creates a new virtual camera and initializes it to the same default setting as `sipp_init()` does with `sipp_camera` (see above).

```
void
camera_destruct(camera)
        Camera  *camera;
```

Release the memory used by a virtual camera. `sipp_camera` can't be destructed and if the camera which is currently used as viewpoint is destructed, the current viewpoint will be reset to `sipp_camera`.

```
void
camera_position(camera, x, y, z)
        Camera  *camera;
        double   x, y, z;
```

Place `camera` at the position (`x, y, z`) in the world coordinate system.

```
void
camera_look_at(camera, x, y, z)
        Camera  *camera;
```

```
        double   x, y, z;
```

Set camera to look at the point (x, y, z) in the world coordinate system.

```
void
camera_up(camera, x, y, z)
        Camera  *camera;
        double   x, y, z;
```

Set the up direction of camera to be the vector (x, y, z) in the world coordinate system. The up direction is not allowed to be parallel to the viewing direction, i.e. the vector from the camera position to the point it is looking at.

```
void
camera_focal(camera, focal)
        Camera  *camera;
        double   focal;
```

Set camera's focal factor to be focal. The focal factor is the ratio between half the screen height and the distance from the viewpoint to the screen. Another way of describing it is tan(v/2) where v is the opening angle of the view. A large focal factor will result in a wide angle view while a small factor will give a telescopic effect. See figure below:

```
                               screen
                               |
                               | s
    viewpoint                  |
        *----------------------|
                    d          |
                               |
                               |


        focal = s / d
```

```
void
camera_params(camera, x1, y1, z1, x2, y2, z2, ux, uy, uz, focal)
        Camera  *camera;
```

```
        double   x1, y1, z1;
        double   x2, y2, z2;
        double   ux, uy, uz;
        double   focal;
```

Set all parameters of a camera in one call. (`x1, y1, z1`) is the position, (`x2, y2, z2`) is the point the camera is looking at, (`ux, uy, uz`) is the up direction and `focal` is the focal factor. Note that the up direction is not allowed to be parallel to the viewing direction, i.e. the vector from the camera position to the point it is looking at.

```
void
camera_use(camera)
        Camera  *camera;
```

Tell SIPP to use `camera` as the current viewpoint.

13 Rendering

SIPP can render images in four different modes:

- `PHONG` rendering interpolates surface normal and texture coordinates across polygons and calls the appropriate shading function in each point. This mode is the slowest but produces the best results and is the only mode where any texturing effects can be used. Note that most of the interesting effects that is possible to produce with SIPP, e.g. shadows and position dependent light (spotlights, point lights), are in fact texturing effects.
- `GOURAUD` rendering only calls the shader in the vertices of the polygons and then interpolates the calculated colors across them. The opacities returned from the shader is interpolated in the same manner.
- `FLAT` rendering calls the shader once per polygon and then fills the whole polygon with the resulting color. The whole polygon will also get the opacity returned from the shader.
- `LINE` rendering will produce a monochrome line image with only the edges of the polygons drawn. No shaders are involved. No hidden line elimination are performed but backfacing polygons are not drawn unless specifically ordered with `sipp_show_backfaces()` (See Chapter 5 [Initializations], page 19).

There are two ways of rendering the currently specified scene. They differ in the place to which the rendered image is sent.

13.1 Rendering to file

There are two functions for rendering into a file.

```
void
render_image_file(width, height, file, mode, oversampling)
        int    width, height;
        FILE  *file;
        int    mode;
        int    oversampling;
```

`width, height`

These parameters specify the size of the image in pixels. If the two sizes are different, the focal factor of the camera (See Chapter 12 [Viewpoint and cameras], page 51) is defined to refer to the smaller of the two.

`file`

This is a pointer to an open file on which the image will be written. If the system supports it, it could just as well be a pipe or a socket of course.

`mode`

This defines the rendering mode, `LINE`, `FLAT`, `GOURAUD` or `PHONG` as described earlier.

`oversampling`

This parameter defines how much oversampling should be performed for anti-aliasing. Each pixel will be rendered internally as a mesh of (`oversampling x oversampling`) subpixels and the average color in this mesh will be used to represent the final pixel. This parameter is ignored in `LINE` mode.

The other function for rendering into a file is useful when doing animations. Since video formats are usually interlaced, it is possible to get a smoother motion if each *field* (half-frame) is rendered separately and the motion is updated between these fields instead of between frames. Unfortunately `LINE` rendering can not be used when rendering fields.

```
void
render_field_file(width, height, file, mode, oversampling, field)
        int    width, height;
        FILE  *file;
        int    mode;
        int    oversampling;
        int    field;
```

`width, height`

These parameters specify the size of the image in pixels. It is the height of the *frame* that should be specified in `height`, not the field, the field height is determined internally.

`file`

This is a pointer to an open file on which the field will be written. If the system supports it, it could just as well be a pipe or a socket of course.

`mode`

This defines the rendering mode, `FLAT`, `GOURAUD` or `PHONG` as described earlier.

`oversampling`

This parameter defines how much oversampling should be performed for anti-aliasing.

`field`

This defines if an odd or even field should be produced. The value should be one of the predefined constants `ODD` or `EVEN`. `ODD` will result in only odd scanlines being rendered, with 0 being the top scanline.

13.2 Rendering to other devices

Sometimes one does not want the rendered image to be stored in a file. Perhaps it should be displayed in a window or further processed in some way. SIPP provides a way to have a function called for each rendered pixel, or for each line if a line image is rendered. The function is given information about which pixel it is and what resulting color it got. Since one of the most used applications of this probably is rendering to a pixmap in memory, SIPP has special support for that. See Section 13.3 [Rendering to in-core images], page 58.

Use a call to the following function to render to another device than a file:

```
void
render_image_func(width, height, pix_func, data, mode, oversampling)
        int    width, height;
        void (*pix_func)();
        void  *data;
        int    mode;
        int    oversampling;
```

`width, height`

These parameters specify the size of the image in pixels. If the two sizes are different, the focal factor of the camera (See Chapter 12 [Viewpoint and cameras], page 51) is defined to refer to the smaller of the two.

`pix_func`

This is a pointer to a function that SIPP calls once for each rendered pixel. If `LINE` rendering is used it is called for each line instead. The function must have the following interface:

```
void
my_pixel_function(data, col, row, red, green, blue)
        my_data       *data;
        int            col, row;
        unsigned char  red, green, blue;
```

`data` This is the same `data` pointer that was passed to `render_image_func()`.

`col, row` Specifies position of the pixel. (0, 0) is upper left.

`red, green, blue` This is the color of the pixel quantified to 24 bits, 8 bits for each of red, green and blue.

If `LINE` rendering is used instead, the user provided function is called for each rendered line instead of each pixel and should have the following interface:

```
void
my_line_function(data, col1, row1, col2, row2)
        my_data       *data;
        int            col1, row1;
```

```
        int             col2, row2;
```

`data` This is the same `data` pointer that was passed to `render_image_func()`.

`row1, col1, row2, col2` Specification of the two endpoints of the line. (0, 0) is upper left,

`data`

This is a pointer to any data structure that the pixel function (see next item) needs. It could be a pointer to a specific pixmap or window or whatever.

`mode`

This defines the rendering mode, `LINE`, `FLAT`, `GOURAUD` or `PHONG` as described earlier.

`oversampling`

This parameter defines how much oversampling should be performed for anti-aliasing. Each pixel will be rendered internally as a mesh of (`oversampling x oversampling`) subpixels and the average color in this mesh will be used to represent the final pixel. This parameter is ignored in `LINE` mode.

There is also a corresponding function to `render_field_file()` for rendering a field into a user defined place. As in that function, `LINE` rendering can not be used when rendering fields.

```
void
render_field_func(width, height, pix_func, data, mode, oversampling, field)
        int    width, height;
        void (*pix_func)();
        void  *data;
        int    mode;
        int    oversampling;
        int    field;
```

All parameters have the same meaning as in `render_image_func()` except the last one.

`field`

This defines if an odd or even field should be produced. The value should be one of the predefined constants `ODD` or `EVEN`. `ODD` will result in only odd scanlines being rendered, with 0 being the top scanline.

13.3 Rendering to in-core images

To people who want to create images in memory, we provide two image formats similar in kind to the Portable Pixmap (ppm) and Portable Bitmap (pbm). Only very simple operations are defined on them, but the definition of the types are also given here, so those who want to write their own functions operating on the images can do so.

13.3.1 The Sipp_pixmap image data type

To use the pixmap operations you must put the following line into your source file:

```
#include <sipp_pixmap.h>
```

In this include file, the Sipp_pixmap data type is defined as well as all operations operating on it. Only the most basic operations are defined.

A Sipp_pixmap is defined like this:

```
typedef struct {
    int      width;            /* Width of the pixmap */
    int      height;           /* Height of the pixmap */
    unsigned char * buffer;    /* A pointer to the image. */
} Sipp_pixmap;
```

The pointer buffer is a pointer to the image where each pixel is stored as three unsigned chars in the order red, green, blue. Thus, the buffer is 3 * width * height bytes long.

The following functions are defined for a Sipp_pixmap:

```
Sipp_pixmap *
sipp_pixmap_create(width, height)
        int  width;
        int  height;
```

Returns a newly created `Sipp_pixmap` with the given size. The new pixmap is filled with zeros on creation.

```
void
sipp_pixmap_destruct(pm)
        Sipp_pixmap *pm;
```

Frees all memory associated to the `Sipp_pixmap` `pm` and returns it to the heap.

```
void
sipp_pixmap_set_pixel(pm, col, row, red, grn, blu)
        Sipp_pixmap   *pm;
        int            col;
        int            row;
        unsigned char  red;
        unsigned char  grn;
        unsigned char  blu;
```

Set the pixel at (`col`, `row`) in pixmap `pm` to be the color (`red`, `grn`, `blu`). (0, 0) is upper left. Note that this function is directly usable in `render_image_func()` defined in Section 13.2 [Rendering to other devices], page 56, when using the `FLAT`, `GOURAUD` or `PHONG` mode of rendering.

```
void
sipp_pixmap_write(file, pm)
        FILE         *file;
        Sipp_pixmap  *pm;
```

Write the pixmap `pm` to the open file `file`. The image is written in the Portable Pixmap format P6 (raw ppm), the same format SIPP is using when rendering to a file.

13.3.2 The Sipp_bitmap image data type

To use the pixmap operations you must put the following line into your source file:

```
#include <sipp_bitmap.h>
```

In this include file, the `Sipp_bitmap` data type is defined as well as all operations operating on it. Only the most basic operations are defined.

A `Sipp_bitmap` is defined like this:

```
typedef struct {
    int   width;                  /* Width of the bitmap in pixels */
    int   height;                 /* Height of the bitmap in pixels */
    int   width_bytes;            /* Width of the bitmap in bytes. */
    unsigned char * buffer;              /* A pointer to the image. */
} Sipp_bitmap;
```

The pointer `buffer` is a pointer to the image where each pixel is a bit in an unsigned char, eight pixels per char. If the `width` field is not a multiple of 8, the last bits in the last byte of a row are not used. The most significant bit in each byte is the leftmost pixel. The entire buffer is `width_bytes` * `height` bytes long.

The following functions operate on a `Sipp_bitmap`:

```
Sipp_bitmap *
sipp_bitmap_create(width, height)
        int width;
        int height;
```

Returns a new `Sipp_bitmap` with the given size. The new bitmap is filled with zeros on creation.

```
void
sipp_bitmap_destruct(bm)
        Sipp_bitmap  *bm;
```

Frees all memory associated to the `Sipp_bitmap` `bm` and returns it to the heap.

```
void
sipp_bitmap_line(bm, col1, row1, col2, row2)
        Sipp_bitmap  *bm;
        int           col1;
        int           row1;
        int           col2;
```

```
        int           row2;
```

Draw a line from (col1, row1) to (col2, row2) in the bitmap bm. (0, 0) is upper left. Note that this function is directly usable in render_image_func() defined in Section 13.2 [Rendering to other devices], page 56, when using the LINE mode of rendering.

```
void
sipp_bitmap_write(file, bm)
        FILE         *file;
        Sipp_bitmap  *bm;
```

Write the bitmap bm to the open file file. The image is written in the Portable Bitmap format P4 (pbm), the same format SIPP is using when rendering a line drawing to a file.

13.4 Aborting a rendering

Sometimes there is a need to abort a rendering prematurely. A user may have pressed a "cancel-button" in a GUI or some similar reason. To be able to do this the application must gain control once in a while during a rendering and SIPP provides a callback mechanism for this See Chapter 5 [Initializations], page 19. To abort a rendering gracefully, the callback function should call the following function:

```
void
sipp_render_terminate()
```

After this function has been called, it is *very* important that control is returned to SIPP! Otherwise the rendering is not aborted as it should. sipp_render_terminate() only indicates to SIPP that the rendering should be terminated, the actual cleanup and exit is performed only after control has been returned to SIPP.

14 Shaders

A major feature in SIPP is the very flexible way shading functions are handled. Each surface has a pointer to a function that is called whenever a point on that surface is rendered. The interface to these shading functions is well defined so it is quite easy for a user to write his own. SIPP also provides a number of shaders in the library for various effects.

14.1 Provided shaders

This section describes all the shaders that are provided with SIPP. To use any of them, except `basic_shader()`, the program must contain the following line:

```
#include <shaders.h>
```

The most important thing to know when using a shader is how it represents its surface description and what this description should contain. All provided shaders in SIPP use a normal C struct as surface description.

14.1.1 The basic shader

The basic shader in SIPP, `basic_shader()`, is basically a Phong shader but, with some influence from Blinn, the "shinyness" of the surface is described with a number in the range [0, 1] and the implemented "shinyness" function changes with this constant in a more natural way (at least in our opinion).

Surface description:

```
typedef struct {
        double ambient;
        double specular;
        double c3;
        Color  color;
        Color  opacity;
} Surf_desc;
```

`ambient` is a number in the range [0, 1] specifying how much of the surface color that is visible when the object is not lit by any lightsource.

`specular` is a number in the range [0, 1] specifying how much light that is reflected in a specular highlight on the surface.

`c3` is also a number in the range [0, 1]. It specifies how "shiny" the surface is. 0 means a very shiny surface while 1 indicates a rather dull one.

`color` is simply the color of the surface.

`opacity` specifies how opaque the surface is. This is stored as a color to allow different opacities for the different color bands. The values should be in the range [0, 1] with 1 indicating a completely opaque object and 0 a completely transparent (invisible) one.

14.1.2 The Phong shader

`phong_shader()` implements the well known Phong illumination model.

Surface description:

```
typedef struct {
        double ambient;
        double diffuse;
        double specular;
        int    spec_exp;
        Color  color;
        Color  opacity;
} Phong_desc;
```

`ambient` is a number in the range [0, 1] specifying how much of the surface color that is visible when the object is not lit by any lightsource.

`diffuse` is a number in the range [0, 1] specifying how much light that is reflected diffusely from the surface.

`specular` is a number in the range [0, 1] specifying how much light that is reflected in a specular highlight on the surface.

`spec_exp` is the exponent in the specular highlight calculation. It specifies how "shiny" the surface is. Useful values are about 1 to 200, where 1 is a rather dull surface and 200 is a very shiny one.

`color` is the color of the surface.

`opacity` specifies how opaque the surface is. This is stored as a color to allow different opacities for the different color bands. The values should be in the range [0, 1] with 1 indicating a completely opaque object and 0 a completely transparent (invisible) one.

14.1.3 The Strauss shader

`strauss_shader()` is a shader designed by Paul Strauss at Silicon Graphics Inc. and published in IEEE CG&A Nov. 1990. In his article he explains that most shading models in use today, e.g. Phong, Cook-Torrance, are difficult to use for non-experts, and this for several reasons. The parameters and their effect on a surface are non- intuitive and/or complicated. The shading model Strauss designed has parameters that is easy to grasp and have a reasonably deterministic effect on a surface, but yet produces very realistic results.

Surface description:

```
typedef struct {
        double  ambient;
        double  smoothness;
        double  metalness;
        Color   color;
        Color   opacity;
} Strauss_desc;
```

`ambient` is a number in the range [0, 1] specifying how much of the surface color that is visible when the object is not lit by any lightsource.

`smoothness` is a number in the range [0, 1] that describes how smooth the surface is. This parameter controls both diffuse and specular reflections. 0 means a dull surface while 1 means a very smooth and shiny one.

`metalness` is a number in the range [0, 1]. It describes how metallic the material is. It controls among other things how much of the surface color should be mixed into the specular reflections at different angles. 0 means a non-metal while 1 means a very metallic surface.

`color` is the color of the surface.

`opacity` specifies how opaque the surface is. This is stored as a color to allow different opacities for the different color bands. The values should be in the range [0, 1] with 1 indicating a completely opaque object and 0 a completely transparent (invisible) one.

14.1.4 The marble shader

`marble_shader()` uses a three dimensional texture to create the appearance of marble. The texture is created by mixing distorted strips of one color into another "base" color of the surface.

Surface description:

```
typedef struct {
        double ambient;
        double specular;
        double c3;
        double scale;
        Color  base;
        Color  strip;
        Color  opacity;
} Marble_desc;
```

`ambient`, `specular`, `c3` and `opacity` have the same meaning as in `basic_shader()` (see Section 14.1.1 [The basic shader], page 62).

`scale` describes how much the texture coordinates should be scaled before applying the texture. When scaling get larger, the object will get larger in comparison to the marble pattern.

`base` is the base color of the surface.

`strip` is the color of the strips which is mixed in with the base color.

14.1.5 The granite shader

`granite_shader()` is very similar to `marble_shader()`. It also mixes two colors to create a three dimensional texture, but the mixing is done in a different manner so the result should look like granite.

Surface description:

```
typedef struct {
        double ambient;
        double specular;
        double c3;
        double scale;
        Color  col1;
        Color  col2;
        Color  opacity;
} Granite_desc;
```

`ambient`, `specular`, `c3` and `opacity` have the same meaning as in `basic_shader()` (see Section 14.1.1 [The basic shader], page 62).

`scale` describes how much the texture coordinates should be scaled before applying the texture. When scaling get larger, the object will get larger in comparison to the granite pattern.

`col1` and `col2` are the two colors that are mixed.

14.1.6 The wood shader

wood_shader() creates a simulated wood texture on a surface. It uses two colors, one as the base (often lighter) color of the wood and one as the color of the (often darker) rings in it. The rings are put into the base color about the x-axis and are then distorted slightly. A similar pattern is repeated at regular intervals to create an illusion of logs or boards.

Surface description:

```
typedef struct {
        double ambient;
        double specular;
        double c3;
        double scale;
        Color  base;
        Color  ring;
        Color  opacity;
} Wood_desc;
```

ambient, specular, c3 and opacity have the same meaning as in basic_shader() (see Section 14.1.1 [The basic shader], page 62).

scale describes how much the texture coordinates should be scaled before applying the texture. When scaling get larger, the object will get larger in comparison with the wood texture.

base and ring are the colors in the wood.

14.1.7 The bozo shader

bozo_shader() uses a random number, correlated with the three dimensional texture coordinates, to chose a color from a fixed set. The user supplies an array of colors to choose from.

Surface description:

```
typedef struct {
        Color  colors[];
        int    no_of_cols;
        double ambient;
        double specular;
        double c3;
        double scale;
        Color  opacity;
} Bozo_desc;
```

colors are an array of colors with no_of_cols entries.

ambient, specular, c3 and opacity have the same meaning as in basic_shader() (see Section 14.1.1 [The basic shader], page 62).

scale describes how much the texture coordinates should be scaled before applying the texture.

14.1.8 The mask shader

mask_shader() uses a user provided decision function to mask between two different shaders. The decision function is passed all three texture coordinates and returns TRUE or FALSE.

The decision function should have the following interface:

```
bool
my_masker(mask, u, v, w)
        my_mask_data *mask;
        int           u, v, w;
```

my_mask_data is a pointer to any data structure that the decision function needs. A common use for mask_shader() is to use a bitmap to mask something onto a surface, in this case my_mask_data could point to the bitmap itself.

u, v and w is the interpolated texture coordinates sent to the shader.

Surface description:

```
typedef struct {
        Shader *t_shader;
        void   *t_surface;
        Shader *f_shader;
        void   *f_surface;
        void   *mask_data;
        bool  (*masker)();
} Mask_desc;
```

The shader t_shader and the surface description t_surface is used to shade the surface whenever the decision function returns TRUE.

The shader f_shader and the surface description f_surface is used to shade the surface whenever the decision function returns FALSE.

mask_data points to any data structure the decision function need.

`masker` is a pointer to the decision function.

14.1.9 The bumpy shader

`bumpy_shader()` is a not really a shader. It is a function that changes the surface normal to create the impression of a bumpy surface. The bumps are dependent on the three dimensional texture coordinates. Any other shader can be used to do the final shading calculations.

Surface description:

```
typedef struct {
        Shader *shader;
        void   *surface;
        double scale;
        bool   bumpflag;
        bool   holeflag;
} Bumby_desc;
```

`shader` points to the shader that should be called to do the actual shading calculations.

`surface` is a pointer to the surface description that should be used in `shader`.

`scale` describes how much the texture coordinates should be scaled before applying the texture.

`bumpflag` and `holeflag` make it possible to flatten out half of the bumps. If only bumpflag is TRUE only bumps "standing out" from the surface are visible. The rest of the surface will be smooth. If, on the other hand, only holeflag is TRUE only bumps going "into" the surface will be visible, thus giving the surface an eroded look. If both flags are true, the whole surface will get a bumpy appearance, rather like an orange.

14.1.10 The planet shader

`planet_shader()` is a somewhat specialized shader that produces a texture that resembles a planet surface. The planet is of the Tellus type with a mixture of oceans and continents. Some of the surface is covered by semi-transparent clouds which enhances the effect greatly. On the other hand, no polar caps are provided and this decreases the realism.

The texture is 3-dimensional, so it is possible to create cube planets or even planets with cut-out parts that still have surfaces that resemble the earth surface. The texture is not scalable, and is designed to be used with texture coordinates in the range [-1, 1], e.g. a unit sphere. The world coordinates need not have the same order of magnitude of course .

Surface description: The planet shader uses the same surface description as `basic_shader()`, a `Surf_desc` (see Section 14.1.1 [The basic shader], page 62), but the colors on the surface are hard coded in the shader, so the color entry in the description is ignored.

14.2 Writing your own shaders

As mentioned earlier, SIPP calls a shading function for every point that is rendered. To be able to perform all necessary calculations, the shader needs quite a lot of information of the state the rendering is in. All information are sent to the shader as pointers to the data used internally in SIPP. It is very important that this information is left unchanged. If any processing of the values is needed, e.g. normalization of the surface normal, the result must be stored in local variables in the shader.

The shading functions have the following interface:

```
void
my_shader(world, normal, texture, view_vec, lights, surface, color, opacity)
        Vector        *world;
        Vector        *normal;
        Vector        *texture;
        Vector        *view_vec;
        Lightsource   *lights;
        void          *surface;
        Color         *color;
        Color         *opacity;
```

`world` is the position in world coordinates of the point that is rendered.

`normal` is the surface normal in the point. Note: this vector is NOT normalized.

`texture` contains the interpolated values of the texture coordinates.

`view_vec` is a vector pointing from the rendered point at the viewpoint, i.e. the currently active camera.

`lights` points to the linked list holding all lights.

`surface` is a pointer to a surface description, i.e. a data area holding information specific for the rendered surface and the shader. The implementor of the shader decides what to put in this area. It is the same pointer that was sent to `surface_create()` (See Chapter 6 [Creating objects], page 23).

`color` points to an area where the shader should place the calculated color of the point.

`opacity` points to an area where the shader should place the calculated opacity of the point.

Since shaders are regular C functions they can be "cascaded". If one do not want to implement a complete illumination calculation but want to do some special effect, like texture or bumpmapping, the easiest way is to write a shader that only manipulates the surface color, normal or whatever, and then calls another shader, like `phong_shader()`, to do the actual shading. This is the way most of the shaders provided in SIPP work (see Section 14.1 [Provided shaders], page 62).

If one wants to implement a new shading model, things get slightly more complicated. Lightsources and possible shadows must be considered. The heart of such a shader must contain a loop over all lightsources, which are stored in a linked list. Inside this loop every lightsource is *evaluated* to see how much light from it that reaches the shaded point. Here is an skeleton example of how the code could look:

```
void
my_shader(world, normal, texture, view_vec, lights, surface, color, opacity)
        Vector        *world;
        Vector        *normal;
        Vector        *texture;
        Vector        *view_vec;
        Lightsource   *lights;
        void          *surface;
        Color         *color;
        Color         *opacity;
{
        Lightsource  *lp;            /* Current lightsource */
        Vector        light_vec;     /* Direction to current lightsource */
        double        light_factor;  /* Fraction of light reaching us */
        Color         light_color;   /* Resulting color from lightsource */

        /*
         * Other declarations and various initializations
         * ...
         * ...
         */

        /*
         * Loop over all lightsources
         */

        for (lp = lights; lp != NULL; lp = lp->next)
        {
                /* Find out where the lightsource are and */
                /* how much light from it that reaches us. */

                light_factor = light_eval(lp, world, &light_vec);

                /* Calculate contributed light from the lightsource */
```

```
                light_color.red = light_factor * lp->color.red;
                light_color.grn = light_factor * lp->color.grn;
                light_color.blu = light_factor * lp->color.blu;

                /*
                 * Calculate shading contribution from the
                 * lightsource using whatever model the shader
                 * implements.
                 * ...
                 * ...
                 */
        }

        /*
         * Store the final calculated color and opacity
         * for the point where SIPP can find it and return.
         */

        color->red = ....
        color->grn = ....
        color->blu = ....

        opacity->red = ....
        opacity->grn = ....
        opacity->blu = ....
}
```

The function `light_eval()` and its parameters are described in more detail in the chapter on lightsources (see Section 10.2 [Manipulating lights], page 46).

15 Object primitives

As mentioned before, SIPP only renders surfaces built up of polygons. Sometimes this is too low a level for the user to program in, so some higher level of abstraction is needed. In the SIPP library a number of functions are provided that generate higher level objects from ordinary SIPP surfaces. Most of them are simple geometric primitives, but some are more sophisticated such as Bezier surfaces. If other types of objects are needed the user has to build them by him/herself (See Chapter 6 [Creating objects], page 23).

Each object primitive which can be created in SIPP has an argument that describes what kind of texture coordinates should be assigned to the surface of the object. This parameter can have one of the following predefined values:

- `NATURAL`

  This value tell SIPP to use a two dimensional mapping which is "natural" for this particular object. It might be one of the other available mappings or it might be something unique for the object. The description of the functions for creating the individual objects specifies how this mapping is done.
- `CYLINDRICAL`

  A two dimensional mapping. The coordinates are assigned as if the object were projected on a cylinder surrounding the object and centered on the z-axis object. The coordinates are mapped so that `x` goes from 0 to 1 around the base of the cylinder and `y` goes from 0 to 1 from bottom to top on it.
- `SPHERICAL`

  Same as `CYLINDRICAL`, but the object are projected on a sphere surrounding it instead.
- `WORLD`

  A three dimensional mapping. The texture coordinates are the same three dimensional coordinates as the world coordinates of the object at creation time.

The following objects are provided in the standard SIPP distribution. To use them, you must put the line

```
#include <primitives.h>
```

into your `C` source file.

15.1 The cube object

This function creates a cube centered about the origin.

The `NATURAL` texture mapping is similar to `CYLINDRICAL` but the `x` coordinate is not taken from projection on a cylinder but is evenly distributed around the perimeter. An odd thing in all the 2D mappings (all except `WORLD`) for the cube is that the top face will have texture coordinates (0.0, 1.0) while the bottom will get (0.0, 0.0).

```
Object *
sipp_cube(size, surface, shader, texture)
        double   size;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`size`

Size of the sides on the cube.

`surface`

Pointer to the surface description to use when shading the cube.

`shader`

Shader to use when shading the cube.

`texture`

Choice of texture mapping.

15.2 The block object

This function creates a rectangular block centered about the origin.

The `NATURAL` texture mapping is similar to `CYLINDRICAL` but the `x` coordinate is not taken from projection on a cylinder but is evenly distributed around the perimeter. An odd thing in all the 2D mappings (all except `WORLD`) for the block is that the top face will have texture coordinates (0.0, 1.0) while the bottom will get (0.0, 0.0).

```
Object *
sipp_block(xsize, ysize, zsize, surface, shader, texture)
        double   xsize, ysize, zsize;
        void    *surface;
```

```
        Shader  *shader;
        int      texture;
```

`xsize, ysize, zsize`

Size of the sides on the block.

`surface`

Pointer to the surface description to use when shading the block.

`shader`

Shader to use when shading the block.

`texture`

Choice of texture mapping.

15.3 The prism object

This function creates a prism, i.e. a polygon in the x,y-plane which is extruded along the z-axis.

The `NATURAL` texture mapping is similar to `CYLINDRICAL` but the `x` coordinate is not taken from projection on a cylinder but is evenly distributed around the perimeter. An odd thing in all the 2D mappings (all except `WORLD`) for the prism is that the top face will have texture coordinates (0.0, 1.0) while the bottom will get (0.0, 0.0).

```
Object *
sipp_prism(num_points, points, zsize, surface, shader, texture)
        int      num_points;
        Vector   points[];
        double   zsize;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`num_points` Number of points defining the prism.

`points` Array of `num_points` points defining the prism. The points should be given counter-clockwise when looking at the prism from above (positive Z). Only the `x` and `y` members in the vectors are significant, the `z` member is ignored.

`zsize`

Size of the prism along the z-axis.

`surface`

Pointer to the surface description to use when shading the prism.

`shader`

Shader to use when shading the prism.

`texture`

Choice of texture mapping.

15.4 The sphere object

This function creates a sphere centered around the origin.

The `NATURAL` texture mapping is `SPHERICAL`.

```
Object *
sipp_sphere(radius, resol, surface, shader, texture)
        double   radius;
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`radius`

The radius of the sphere.

`resol`

The sphere is tessellated into polygons. `resol` tells SIPP how many polygons there should be around the "equator" of the sphere.

`surface`

Pointer to the surface description to use when shading the sphere.

`shader`

Shader to use when shading the sphere.

`texture`

Choice of texture mapping.

15.5 The ellipsoid object

This function creates an ellipsoid centered around the origin.

The `NATURAL` texture mapping is `SPHERICAL`.

```
Object *
sipp_ellipsoid(xradius, yradius, zradius, resol, surface, shader, texture)
        double   xradius;
        double   yradius;
        double   zradius;
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`xradius, yradius, zradius`

The radii of the ellipsoid in the principal axes directions.

`resol`

The ellipsoid is tessellated into polygons. `resol` tells SIPP how many polygons to generate around the "equator" of the ellipsoid.

`surface`

Pointer to the surface description to use when shading the ellipsoid.

`shader`

Shader to use when shading the ellipsoid.

`texture`

Choice of texture mapping.

15.6 The cylinder object

This function creates a cylinder centered around the z-axis and the origin.

The `NATURAL` texture mapping is `CYLINDRICAL`.

```
Object *
sipp_cylinder(radius, resol, surface, shader, texture)
        double   radius;
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`radius`

Radius of the cylinder.

`resol`

The cylinder is tessellated into polygons, `resol` tells SIPP how many polygons there should be around it.

`surface`

Pointer to the surface description to use when shading the cylinder.

`shader`

Shader to use when shading the cylinder.

`texture`

Choice of texture mapping.

15.7 The cone object

This function creates a, possibly truncated, cone centered around the z-axis and the origin.

The `NATURAL` texture mapping is `CYLINDRICAL`.

```
Object *
sipp_cone(topradius, bottomradius, resol, surface, shader, texture)
        double   topradius;
        double   bottomradius;
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`topradius, bottomradius`

Radius of the cone at the top and bottom. If the cone should be pointed at one of the end, specify 0 as radius.

`resol`

The cone is tessellated into polygons, `resol` tells SIPP how many polygons there should be around it.

`surface`

Pointer to the surface description to use when shading the cone.

`shader`

Shader to use when shading the cone.

`texture`

Choice of texture mapping.

15.8 The torus object

This function creates a torus centered around the z-axis and the origin.

The `NATURAL` texture mapping is a two dimensional mapping with the `x` coordinate going around the "small" circle and the `y` coordinate going around the "large" circle.

```
Object *
sipp_torus(bigradius, smallradius, res1, res2, surface, shader, texture)
        double   bigradius;
        double   smallradius;
        int      res1;
        int      res2;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`bigradius, smallradius`

Radius of the big and small circle defining the torus, the small circle is swept along the big one to sweep out the torus.

`res1, res2`

The torus will be tessellated into `res1 x res2` polygons. `res1` is the number of vertices around the big circle and `rad2` is the number of vertices around the small one.

`surface`

Pointer to the surface description to use when shading the torus.

`shader`

Shader to use when shading the torus.

`texture`

Choice of texture mapping.

15.9 The Bezier patch

This function creates one or more Bezier patches. All created patches in a call will belong to the same surface.

The texture coordinates are a bit special for the Bezier patches. `CYLINDRICAL` and `SPHERICAL` coordinates are not applicable, if they are specified, SIPP will use `NATURAL` anyway. The `NATURAL` mapping is a two dimensional mapping using the surface parameters *u* and *v*, see figure below. Note that these parameters range from 0 to 1 within each patch!

The patches are defined with a list of vertex coordinates and a set of 16 indices into that list for each patch. The following figure show in which order the indices to vertices corresponding to control points for the patch should be given (and how *u* and *v* varies over the patch):

```
v=1  13____14____15____16
      |     |     |     |
      |     |     |     |
      9____10____11____12
      |     |     |     |
      |     |     |     |
      5_____6_____7_____8
      |     |     |     |
      |     |     |     |
v=0   1_____2_____3_____4

     u=0               u=1
```

```
Object *
sipp_bezier_patch(num_vertex, vertex, num_patch, vx_index, resol,
                  surface, shader, texture)
        int      num_vertex;
        Vector   vertex[];
        int      num_patch;
        int      vx_index[];
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`num_vertex, vertex`

The array `vertex` contains a list of `num_vertex` vertices.

`num_patch`

The number of patches that should be defined.

`vx_index`

A list of `16 * num_patch` indices into `vertex` defining the control mesh of the patches. The vertices for each patch should be specified in the order indicated in the figure above.

`resol`

Each patch will be tessellated into `resol x resol` polygons.

`surface`

Pointer to the surface description to use when shading the patches.

`shader`

Shader to use when shading the patches.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.10 The Bezier rotation curve

This function creates a surface by rotating one or more Bezier curves about the world z-axis.

The texture coordinates are a bit special for these surfaces. `SPHERICAL` and `CYLINDRICAL` mappings are not applicable, and `NATURAL` mapping will apply to the piece of surface created by each Bezier curve separately. The `NATURAL` mapping uses the curve parameter *u* along each curve as `x` coordinate and goes from 0 to 1 around the perimeter of the rotational surface on the other axis

The curves are defined with a list of vertex coordinates and a set of 4 indices into that list for each curve. The following figure show in which order the indices to vertices corresponding to control points for the curve should be given.

```
        4  u=1
z-axis   \
    ^     \
    |      3
    |      |
    |      |
    |      2
    |       \
    |        \
    |         1  u=0
```

```
Object *
sipp_bezier_rotcurve(num_vertex, vertex, num_curve, vx_index, resol,
                  surface, shader, texture)
        int       num_vertex;
        Vector    vertex[];
        int       num_curve;
        int       vx_index[];
        int       resol;
```

```
void    *surface;
Shader  *shader;
int      texture;
```

`num_vertex, vertex`

The array `vertex` contains a list of `num_vertex` vertices.

`num_patch`

The number of curves that should be defined.

`vx_index`

A list of `4 * num_patch` indices into `vertex` defining the control polygon for the curves. The vertices for each curve should be specified in the order indicated in the figure above.

`resol`

Each rotational surface will be tessellated into `resol x 4*resol` polygons, `resol` vertices along the curve and `4*resol` vertices around the perimeter.

`surface`

Pointer to the surface description to use when shading the surface.

`shader`

Shader to use when shading the surface.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.11 The Bezier file

This functions reads descriptions of Bezier patches or Bezier curves in a predefined format from a file and creates objects out of them. The file can contain a description of patches or curves, but not both. If curves are defined, a surface will be created by rotating them about the world z-axis. The file contain basically the same information as the parameters to a call to `sipp_bezier_patch()` or `sipp_bezier_rotcurve()` and texture mapping is applied in the same way as in these functions too.

The format of the file is very simple. Please note however, that the format differs slightly from the way the data were specified in the previous two functions. This is for compatibility with older versions. The differences are noted in detail at the spots marked *Diff:* below.

First in the file is a keyword defining the type of description in the file, `bezier_curves:` or `bezier_patches:`. Then follows a description of the vertices (control points). First the word `vertices:` followed by an integer number that tells how many vertices there are in the description, then the word `vertex_list:` followed by the x, y and z coordinates for each vertex. The number of vertices must be same as the number given above. This is, however, not checked for.

If the file contains curves, the keyword `curves:` followed by the number of Bezier curves in the file is on the next line. After this line, a line with the single keyword `curve_list:` follows. Lastly, the Bezier curves themselves follow as numbers in groups of four by four.
Diff: Each number is an index into the vertex list with the first index having number 1.
Diff: The indices are given in the opposit order compared to `sipp_bezier_rotcurve()`.

If the file contains patches, the format is the same with the following exceptions: The word `patches:` is substituted for `curves:`, the word `patch_list:` is substituted for `curve_list:` and the indices into the vertex list are grouped 16 by 16 instead of 4 by 4.
Diff: Each number is an index into the vertex list with the first index having number 1.

Comments can be inserted anywhere in a Bezier curve/patch description file by using the hashmark character, `#`. The comment lasts to the end of the line.

As an example of a Bezier file is here the body of a standard Newell teapot:

```
# Bezier curves (rotational body) for teapot body.

bezier_curves:

vertices: 10
vertex_list:
    3.500000E-01    0.000000E+00    5.625000E-01
    3.343750E-01    0.000000E+00    5.953125E-01
    3.593750E-01    0.000000E+00    5.953125E-01
    3.750000E-01    0.000000E+00    5.625000E-01
    4.375000E-01    0.000000E+00    4.312500E-01
    5.000000E-01    0.000000E+00    3.000000E-01
    5.000000E-01    0.000000E+00    1.875000E-01
    5.000000E-01    0.000000E+00    7.500000E-02
    3.750000E-01    0.000000E+00    1.875000E-02
    3.750000E-01    0.000000E+00    0.000000E+00

curves:    3
curve_list:

  1 2 3 4

  4 5 6 7
```

```
  7 8 9 10

#End of teapot bezier file
```

```
Object *
sipp_bezier_file(file, resol, surface, shader, texture)
        FILE    *file;
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`file`

An open filepointer to the file containing the descriptions.

`resol`

Each rotational surface will be tessellated into `resol x 4*resol` polygons, `resol` vertices along the curve and `4*resol` vertices around the perimeter.

Patches will be tessellated into `resol x resol` polygons.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.12 The teapot

This function creates a model of the famous "Utah Teapot" as a SIPP object. The model is built as a combination of four subobjects: the body, the lid, the handle and the spout. These subobjects are also available as separate primitive objects.

The body and the lid are created as Bezier rotation curves while the handle and the spout are created as sets of four Bezier patches each. Texture coordinates and resolution are assigned in the same way as for these primitives (See Section 15.10 [The Bezier rotation curve], page 80 and See Section 15.9 [The Bezier patch], page 78).

```
Object *
```

```
sipp_teapot(resol, surface, shader, texture)
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`resol`

Each rotational surface will be tessellated into `resol x 4*resol` polygons, `resol` vertices along the curve and `4*resol` vertices around the perimeter. The teapot body is built from three rotational surfaces and the lid from two.

Patches will be tessellated into `resol x resol` polygons. Both the handle and the spout are built from four patches.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.12.1 The teapot body

This function creates the body of the "Utah Teapot" from three Bezier rotational curves.

```
Object *
sipp_teapot_body(resol, surface, shader, texture)
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`resol`

Each rotational surface will be tessellated into `resol x 4*resol` polygons, `resol` vertices along the curve and `4*resol` vertices around the perimeter. The teapot body is built from three rotational surfaces.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.12.2 The teapot lid

This function creates the lid of the "Utah Teapot" from two Bezier rotational curves.

```
Object *
sipp_teapot_lid(resol, surface, shader, texture)
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`resol`

Each rotational surface will be tessellated into `resol x 4*resol` polygons, `resol` vertices along the curve and `4*resol` vertices around the perimeter. The teapot lid is built from two rotational surfaces.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.12.3 The teapot handle

This function creates the handle of the "Utah Teapot" from four Bezier patches.

```
Object *
sipp_teapot_handle(resol, surface, shader, texture)
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`resol`

Each patch will be tessellated into `resol x resol` polygons.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

15.12.4 The teapot spout

This function creates the spout of the "Utah Teapot" from four Bezier patches.

```
Object *
sipp_teapot_spout(resol, surface, shader, texture)
        int      resol;
        void    *surface;
        Shader  *shader;
        int      texture;
```

`resol`

Each patch will be tessellated into `resol x resol` polygons.

`surface`

Pointer to the surface description to use when shading the surfaces.

`shader`

Shader to use when shading the surfaces.

`texture`

Choice of texture mapping (only `NATURAL` and `WORLD` is applicable.

16 Future enhancements

SIPP is constantly under development and we often run into new interesting things that we would like to see included. Here is a small list of such things, some more realistic than others. If you feel like adding to this list, please do! Check out the chapter on bug reports (Chapter 17 [Reporting bugs], page 88) for information on how to get in touch with us.

- More sophisticated anti-aliasing. This should not be so difficult with the new pixel buffer.
- Generalized interface to lightsources, much in the same way as the shader interface. This would allow users to design "lightsource shaders"
- Better support for animation.
- Support for some more advanced object primitives, especially patches (Hermite, NURBS, etc.)
- Four channel output. Write out an alpha channel together with RGB. This is troublesome if we want to stick with the ppm-format which does not support this. Possible solutions include switching to Utah Raster format or writing the alpha channel in a separate pgm-file.
- Curved surface rendering (this would mean a name change I guess... :-))
- Use some sort of "virtual memory" by swapping things to disk. This would make it possible to run SIPP on machines with brain-damaged OS and/or hardware which doesn't support real VM.
- Front-end for reading RIB-files (RenderMan Interface Bytestream). This might not be as impossible as it may sound.
- Store objects in a "higher order" format, and tessellate to polygons at rendering time. This could allow generalizing the object interface too so users could supply their own objects, with tessellation functions. (Yes, I have been reading the RenderMan specs...)

16.1 Contributions

We are grateful for all donations of code that we can receive. We are especially looking for new primitive objects and interesting shaders.

17 Reporting bugs

We have tried to test SIPP thoroughly, but since it is constantly being developed, there are probably numerous bugs remaining, both in the source code and in the documentation. If you find a bug in either, please send a bug report to either `jonas-y@isy.liu.se` or `ingwa@isy.liu.se`. We will try to be as quick as possible in fixing the bugs and redistributing the fixes.

/Jonas Yngvesson & Inge Wallin

Concept index

Function index

Table of Contents