`aperture`

the generalized aperture program

version 2.2.0

Diab Jerius D. Nguyen M. Tibbetts

August 22, 2013

Contents

Chapter 1

Copying

Chapter 2

Overview

The generalized aperture program (`aperture`) is designed to simulate the effects on the incident ray stream of apertures in physical obstructions (such as the X-ray and thermal baffles). It can handle a wide variety of aperture shapes, and has provisions to allow alteration of the rays by the apertures. The philosophy behind `aperture` is that a geometrically complicated aperture may be modeled by a combination of geometrically simpler apertures. Rather than modeling each complex aperture with a separate program, if a flexible means of specifying the combinations of simple apertures were devised, a single program could model a wide range of shapes.

2.1 Requirements

The generalized aperture program meets the following requirements:

- read and write rays in the `bpipe` format.
- allow the combination of multiple apertures to form a single aperture.
- provide a flexible means of specifying the positions and sizes of the apertures.
- provide for the creation of multiple ensembles of apertures, known as *assemblies*. An assembly could refer to a physically distinct structure, such as a single plate in a multi-plate baffle; in rare circumstances two or more assemblies are used to model a single physical object. Assemblies may optionally be divided into sub-assemblies, which may be nested.
- provide the capability of specifying global rotations and translations of individual apertures as well as assemblies.
- provide for the blocking or passing on of rays, as well as alteration of rays by apertures, such as redirection, generation of new rays, and the modification of ray parameters.
- If a ray is redirected by an aperture, `aperture` shall provide the option of rescanning all of the apertures in the current assembly to ascertain of the redirected ray would interact with them.
- If any aperture accepts a ray, the ray is deemed to have cleared the current assembly, and is moved to the next assembly.
- provide an easy means of adding additional aperture shapes

2.2 Design

The modeling of apertures as combinations of sub-apertures, as well as the recognition that the bookkeeping involved in tracking the rays through the apertures was independent of any particular type of aperture, led to the structuring of the program into three components:

- a *front end*, which parses the description of the openings and generates lists of apertures to check the rays against
- a *central engine*, which reads the rays and checks them against the apertures in the lists
- a *back end*, composed of modules which model each of the apertures and which are called upon by the front end and central engine

The three sections of the program are encapsulated so that a minimum of information is exchanged between them, permitting integration of new modules into the back end without affecting the rest of the program.

2.2.1 Aperture Positions and Orientations

Rather than have the user assign absolute positions and orientations to each aperture, `aperture` uses the concept of a local coordinate system in which each aperture is placed. It keeps track of the transformations required to map between the *external* coordinate system in which the input rays' positions and directions are specified and the apertures' local coordinate systems. It provides for the hierarchical "layering" of coordinate systems:

1. a *global* coordinate system, relative to which assemblies are specified
2. *assembly* coordinate systems, relative to which either sub-assemblies or apertures are specified
3. *sub-assembly* coordinate systems, relative to which either nested sub-assemblies or apertures are specified

At each level, changes to the current coordinate system do *not* affect the higher levels' coordinate systems.

2.2.2 The Aperture Description

Specifying lists of positions and sizes of apertures would suffice to completely describe an opening, but would be tedious and error prone. The route taken here is to provide a language with which to lay out the apertures. The language, `Lua`, is a small language with flow control (`if...then`, `while..do..end`, etc.), functions, floating point arithmetic, multi-dimensional arrays and structures, and implicit dynamical memory allocation and deallocation. The front end provides `Lua` callable routines to translate and rotate the current coordinate systems as well as to create assemblies and sub-assemblies. The back end modules provide functions, callable by `Lua`, which create instances of apertures. Both assemblies and apertures are kept in lists; there is one list of assemblies, but many lists of apertures, one list per assembly. For more information on writing aperture description programs, see §3.2.

2.2.3 The Central Engine

The central engine takes the lists of assemblies and apertures created by the front end, and compares each input ray to the apertures, in turn. It calls functions provided by the back end modules which do the actual checking of the rays. It provides the logic to move the ray to the next assembly should an aperture accept a ray and pass it along.

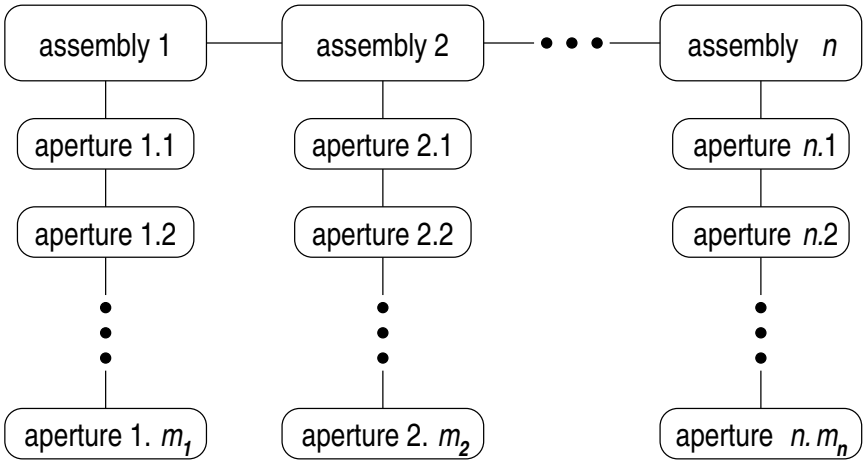

Figure 2.1: Assembly Hierarchy

2.2.4 The Back End Modules

Each back end module consists of two components. The first is called by the user's aperture definition script, and creates an instance of an aperture with a given set of aperture specific parameters, attaching it to the list of apertures for the current assembly. The second component contains the logic necessary to determine if a ray falls within it and is affected by it. Apertures can simply pass or block rays, but, depending upon the complexity of the aperture that they are modeling, can also redirect the rays (e.g. reflection), modify them, or generate new rays (e.g. flourescence). More information on the structure of modules is contained in §3.3.

2.2.5 Detailed Operation

The `Lua` program creates assemblies and sub-assemblies via calls to `begin_assembly`, `end_assembly`, `begin_subassembly` and `end_subassembly`. Assemblies are represented internally as a list of `Assembly` structures. Sub-assemblies are temporary constructs which exist only to allow stacking of the current transformation matrix.

The back end modules called from the `Lua` program create instances of apertures by creating aperture-specific data objects which contain the information necessary to fully describe the aperture. For example, an annulus is described by its inner and outer radius; a rectangle by its height and width. The module itself is described by an `ApertureModule` structure, which contains pointers to standard routines provided by the module (aperture instantiation, initialization, ray processing, cleanup). This structure and the data object are passed to the front end utility routine `new_node`, which encapsulates them inside of an abstract object, `Aperture`, and inserts it at the end of the list of apertures for the current assembly. Conceptually, the hierarchy of assemblies is as shown in Figure 2.1.

The processing of rays begins by reading in a ray. This is usually done from the specified input stream, but may be done from a a special internal ray stack if any rays have been generated by an aperture. It then determines the first valid assembly which the ray will interact with. Usually this is the first assembly, but if the ray is one generated by an aperture, the first valid assembly is determined by the assembly to which the generating aperture belongs.

Beginning with this first valid assembly, the central engine simulates the interactions of the ray with the apertures by an in-order traversal of the assembly's aperture list. At each visit of an aperture, the module associated with the aperture is passed the data object specific to the aperture and a copy of the ray data packet. It uses these data to determine whether it can process the ray, and returns a code to the central engine indicating whether or not it has accepted the ray for processing and the state of the ray after

processing. Normally the module does not alter the contents of the ray data packet; passing it a copy is a safety measure. In the event that it has done so purposefully, it signals (via the return code) the central engine to transfer the contents of the copied ray data packet to the original, so that the remaining apertures interact with the modified ray.

Depending upon the returned code, the central engine may discard the ray and read a new one, begin processing the next assembly in line, or re-start the traversal of the current assembly. The last action is taken if the option *loop-mode* is enabled and rays have been redirected or generated. It is the only means by which a ray can successfully interact with more than one aperture in an assembly.

The defined return codes and the reactions of the central engine are given in Table 2.1.

Table 2.1: Return Codes from Module Ray Processing Functions

`NOTAPPLICABLE`	Pass the ray to the next aperture in the assembly, or begin the next assembly.
`BLOCKED`	Discard the current ray, and process the next ray.
`PASSED`	The current ray is done with the current assembly and will be processed by the next assembly. That is, if aperture 2 of assembly 2 returns `PASSED`, then the next aperture to be processed is aperture 1 of assembly 3, skipping the remaining apertures in assembly 2.
`REDIRECTED`	If the *loop-mode* option is on, then reprocess all of the apertures in the current assembly. That is, if aperture 2 of assembly 2 returns `REDIRECTED`, the next aperture processed will be aperture 1 of assembly 2. If the *loop-mode* option is off, then the behavior is the same as if `MODIFIED` were returned.
`GENERATE`	The module function has created new rays. It does this by passing the rays to the `push_photon` function, which pushes copies onto the system ray stack. The newly generated rays are only valid for assemblies subsequent to the current one, unless *loop-mode* is on, in which case they are valid for the current assembly as well. The central engine discards the current ray, reads the next ray (which has been generated by the module), advances to the valid assembly, and begins processing the ray through that assembly.
`MODIFIED`	The module has modified its copy of the ray data packet, and the central engine transfers the contents of the modified copy to its original data packet. It then continues processing the ray as if `PASSED` had been returned.

Chapter 3

User's Guide

3.1 Program Parameters

`aperture` uses the standard parameter interface. The parameters it recognizes are:

`aperture` the filename of the `Lua` program which defines the aperture

`override` `Lua` code that will be made available to the `Lua` definition program as the `Lua` function `override` (see Appendix A for more information). This can either by actual code, for example

```
aperture override='energy=1.49; shell=3'
```

or it may be the name of a file containing the `Lua` code. In the latter case, the first character should be the `@` character:

```
aperture override=@code.lua
```

`input` the name of the input ray stream. If it is the string `stdin`, `aperture` will read from the standard input stream.

`output` the name of the output ray stream. If it is the string `stdout`, `aperture` will write to the standard output stream.

`statfname` the filename to output the accumulated statistics of the components. The output file is an rdb table containing a column for the name of the component and columns for all possible outcome of the interaction of the ray with the component. The prefix n and w (for example nblocked and wblocked) stands for the number and weight respectively.

`loop` a boolean variable indicating that rays that have been redirected by an assembly should be checked against that assembly first, rather than being checked against the next assembly.

`help` a boolean variable indicating that `aperture` should print out usage information should be printed and exit.

`version` a boolean variable which, if true, indicates that `aperture` should print out its version information and exit.

3.2 Constructing an Aperture

Aperture construction consists of placing geometric components (such as circles, polygons, wedges) into assemblies.

3.2.1 Components

The currently available components include:

`annulus`	an annulus
`circle`	a circle
`ellipse`	an ellipse
`polygon`	a polygon
`rect`	a rectangle
`strut`	an infinitely tall rectangle
`wedge`	an angular wedge of infinite extent
`wedger`	an angular wedge with an maximum outer radius, i.e. a pie slice
`wedger2`	an angular wedge with an inner and outer radii

Some components are mostly of diagnostic aid:

`ell`	an component that looks like the letter 'L'
`block`	block all rays
`pass`	pass all rays
`print_ray`	print the rays' coordinates and directions

All of the components are described more fully in Appendix B.

3.2.2 Placement

Placement of an component is done by specifying a set of coordinate transformations which define the location and orientation of components' intrinsic coordinate system (components are centered at the origin in their $X-Y$ plane). Each transformation updates the *current transformation matrix* (CTM), which maps rays in the global coordinate system to the current transformed system. Components are assigned their position and orientation from the CTM existent when they are instantiated.

Coordinate transformations may be specified as either *intrinsic* or *extrinsic* transformations. Intrinsic transformations are performed *relative to the current coordinate system.* For example, an intrinsic rotation about the X axis followed by one about the Y axis will be about the *transformed* Y axis, not that existing prior to the rotation about the X axis. In intrinsic transformations the world is transformed around you; your coordinate system is always fixed. Extrinsic transformations are always relative to a fixed external system independent of the orientation in space of the current coordinate system.

When `aperture` begins, the CTM is set to the identity matrix. The top-level coordinate system mapped by this CTM (which exists outside of the assemblies) is referred to as the the *global* coordinate system (see §2.2.1), and is the initial CTM for an assembly. Changes made to the CTM inside of an assembly do *not* affect the global coordinate system. Inside of an assembly, extrinsic transformations are performed relative to the coordinate system existent at the time the assembly was created.

Within an assembly, `aperture` uses the concept of logical (possibly nested) *sub-assemblies* to manage complex transformations. The initial CTM of a sub-assembly is the CTM of its parent at the time the sub-assembly is created. As with assemblies, transformations within a sub-assembly do not affect the parent assembly or sub-assembly. Inside of a sub-assembly extrinsic transformations are performed relative to the coordinate system existent at the time the sub-assembly was created. Unlike in assemblies, there is no intrinsic meaning to the collection of components in a sub-assembly. They are merely constructs to manipulate coordinate systems.

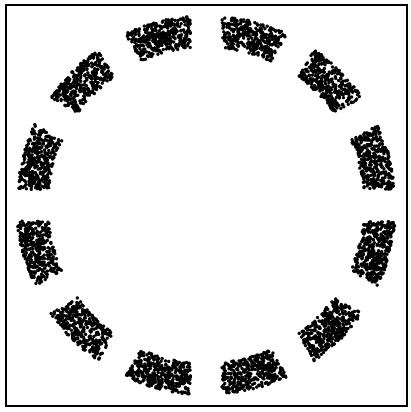

Figure 3.1: An example aperture

Internally, `aperture` keeps a stack of transformation matrices, one for each assembly or sub-assembly which is currently active (e.g., those begun but not ended). Each matrix specifies the transformation which will be applied to the previous entry in the stack. For design purposes, the CTM (which is the inverse of the product of the matrices in the stack), the stack top, or the full stack may be output (see Appendix A for more information).

3.2.3 How to approach aperture construction

When building a complicated opening, components are layered on top of each other, allowing one to mask or "cut out" sections to block rays or let rays through. Components are collected into assemblies which in practice usually refer to physical objects (such as a plate with holes cut into it). It is sometimes useful to model obstructions composed of a single piece with multiple assemblies. Assemblies should be ordered in increasing axial location.

When building assemblies, it is important to understand how `aperture` guides a ray through them. A ray is checked against all components in an assembly in the order that they are specified by the user. If a component accepts responsibility for a ray (e.g. it falls within its geometric area), it can either block the ray or pass it on. If the component cannot handle the ray, it notifies the program that it is not applicable, and the ray is checked against the next component in the assembly. In the event that no component claims responsibility for the ray, it can either be considered blocked or passed, at the user's discretion. If a ray is blocked, `aperture` fetches the next ray and starts over. If it is passed, the program jumps directly to the next assembly and begins the checking process there.

It is somewhat more expensive to jump to the next assembly than it is to jump to the next component within an assembly. It behooves one to model an aperture in a single assembly if possible. If there are many components in a single assembly, it may take time to check rays against all of them. The trick is thus to get rid of as many rays as possible early on in the assembly, and only pass good ones on to the next assembly. In order to make the rejection process more efficient, most of the provided components can act as reverse masks. For example, the `annulus` component can act upon rays which fall within the annulus, and either block or accept those, or it can act upon those which fall outside of the annulus, and act on those. In addition, while assemblies normally block rays which are ignored by all of their components, an assembly can be instructed to pass on the ignored rays. This feature can be used instead of the `pass` component, which, because it is an component, adds overhead.

As an illustration of how to built an aperture from components, imagine trying to model an aperture which looks like an annulus with twelve opaque struts crossing the annulus. Figure 3.1 shows the clear aperture (as seen by the rays). Assume that most of the rays that are lost at this aperture are clipped by the annulus, not the struts. Here are a few of ways of constructing it:

```
deg2rad = 3.14159265358979/180.0
strut_width = 2
r_i = 10
r_o = 12

begin_assembly( )

  -- twirl the struts
  begin_subassembly()
    theta = 0
    dtheta = 30
    while theta < 180 do
      strut( strut_width, 1, 0 )
      rotate_z( dtheta * deg2rad )
      theta = theta + dtheta
    end
  end_subassembly( )

  annulus( r_i, r_o, 1, 1 )

end_assembly( )
```

Figure 3.2: A first cut at the example aperture

1. Check the struts first, blocking out all rays which fall within the struts, then check the annulus, passing all those which falls inside of it. Since most of the rays that are blocked fall outside of the annulus, rather than within the struts, this is inefficient.

2. Check the annulus first, passing the rays that fall within it. However, because passed rays move on to the next assembly, the struts will have to be modeled as a separate assembly, which introduces extra overhead that isn't necessary. In addition, the rays which aren't claimed by the struts in the second assembly will need to be claimed, either by the `pass` component, or by the assembly.

3. Check against the *outside* of the annulus, blocking the rays which fall there. In this case the `annulus` component will not deal with those rays which fall within it, and they will be passed on to the next component (not the next assembly). Check against the struts next, blocking those that fall within, and, finally, pass on the unclaimed rays.

It's important to get the sequence of components correct. Since every aperture is different, experimentation is the key to success. There is always more than one way to create it, but some are more efficient than others. Checking a ray against an component has an invocation cost as well as the cost of transforming the ray into the component's reference frame and the cost of whatever component specific processing is required. The first two costs are the same for each component. The exceptions to this rule are the `pass` and `block` components, which have only an invocation cost, as they don't care where the ray is and do no processing of the ray.

3.2.4 The nitty gritty details

Aperture descriptions are written the language `Lua`. `Lua` is a lightweight language which provides dynamic variable typing, automatic allocation (and freeing) of storage space, functions, flow control and decision branches. `aperture` provides some special functions that translate and rotate the coordinate system and place and size the components.

```
deg2rad = 3.14159265358979/180.0
strut_width = 2

function do_struts( )
  local theta = 0
  local dtheta = 30

  begin_subassembly( )

  while theta < 180 do
    strut( strut_width, 1, 0 )
    rotate_z ( dtheta * deg2rad )
    theta = theta + dtheta
  end

  end_subassembly( )

end
```

Figure 3.3: A Lua function to draw struts

Figure 3.2 is a Lua program which implements the first example in the previous section. It illustrates most of the required parts of an aperture description. Before any components can be placed within an assembly, the assembly must be initialized with the `begin_assembly` function. It must always finish off an assembly with `end_assembly`. Sub-assemblies are treated similarly. Here the script accepts an assembly's default behavior of blocking rays which are not claimed by the assembly's components. Had the initialization call been `begin_assembly("pass")`, the assembly would have passed unclaimed rays. The function can also take the argument `"block"` to specify the default behavior. The user will be alerted if an assembly is ended prematurely (or not ended at all).

The struts are placed at even angular increments. Since they are infinitely tall, they are only distributed in the upper two quadrants. Because coordinate transformations are cumulative, the struts are created inside of a sub-assembly to isolate their positioning from that of the annulus. Since the struts' coordinate systems are only rotated with respect to the global system, it really doesn't matter to the annulus component, but in general, its wise to encapsulate things into sub-assemblies when mucking about with the coordinate system. The program can manipulate the transformation matrix inside a sub-assembly with impunity.

The `strut` and `annulus` functions take a few more parameters than might be intuitively obvious. The first extra parameter instructs the component whether to deal with rays that are interior or exterior to the component (1 means interior), the second specifies whether the component is transparent (1) or opaque (0). Most components will have similar parameters.

The second example from the previous section required multiple assemblies. In order to keep things a bit clearer, we define a function (Figure 3.3) that will be used in subsequent examples to place the struts. Note that in Lua, just as in Pascal, all functions must be defined before use. Figure 3.4 shows the remainder of the code. Here the ability of an assembly to pass unclaimed rays is used. Figure 3.5 contains the code for the last, most efficient approach.

These examples have only rotated coordinates about the z axis. There are additional rotation functions for the other axes, as well as a coordinate translate function. There are also some utility functions to print out the assembly and the transformation stack. These are documented in Appendix A.

As a final more complicated example, the following is the code used to create Figure 3.6.

```
deg2rad = 3.14159265358979/180.0

-- ears are drawn as reflections across the x=0 axis.
```

```
r_i = 10
r_o = 12

begin_assembly( "block" )
  annulus( r_i, r_o, 1, 1 )
end_assembly( )

begin_assembly( "pass" )
  do_struts( )
end_assembly( )
```

Figure 3.4: A second cut at the example aperture

```
r_i = 10
r_o = 12

begin_assembly( "pass" )
  annulus( r_i, r_o, 0, 0 )
  do_struts( )
end_assembly( )
```

Figure 3.5: A third cut at the example aperture

```
--  they just touch the face.
function draw_ear ( angle, face, ear )
  begin_subassembly()
    rotate_z ( angle * deg2rad )
    translate( face + ear, 0, 0 )
    circle( ear, 1, 1)
  end_subassembly()
end

function draw_ears(angle, face, ear )
  draw_ear ( 90 - angle / 2, face, ear )
  draw_ear ( 90 + angle / 2, face, ear )
end

-- eyes are also reflected across the x=0 axis
function draw_eyes ( y, sep, eye )
  begin_subassembly()
    translate(-sep/2, y, 0)
    circle( eye, 1, 0 )
    translate( sep, 0, 0)
    circle( eye, 1, 0 )
  end_subassembly()
end

-- the smile is drawn as a transparent ellipse almost covering
-- an opaque ellipse.  the transparent must be drawn first.
function draw_mouth (y, a, b, sep )
  begin_subassembly()
  translate(0, y, 0)
  ellipse( a , b , 1, 1)
```

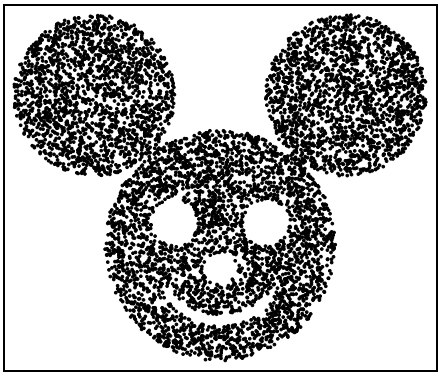

Figure 3.6: A famous aperture

```
  translate(0, -sep, 0)
  ellipse( a , b , 1, 0)
  end_subassembly()
end

face = 5
ear = 3.5

-- the overall scheme is to draw the opaque "cut outs" first:
-- the eyes, the nose, and the mouth.  then we accept all of the
-- rays through the face and ear apertures
begin_assembly()

draw_eyes( 1, 4, 1 )

draw_ears( 80, face, ear )

-- nose
begin_subassembly()
  translate(0, -1, 0)
  circle( 0.7, 1, 0 )
end_subassembly()

draw_mouth( -1, 2.5, 2, 0.5 )

circle( face, 1, 1 )

end_assembly()
```

3.3 Writing a module

Writing modules is much too complicated for mere mortals.

Appendix A

Lua Accessible utility functions

These functions are provided by the front end of `aperture` to create assemblies and sub-assemblies, manipulate the local coordinate system, and provide diagnostic output. They are called from the user's `Lua` aperture definition program.

`begin_assembly( [ "pass" | "block" ] )`
Begin an assembly. The current transformation matrix is set equal to the global transformation matrix. The argument is optional, and is one of the specified strings. If the argument is `"pass"`, rays which are not claimed by any components in an assembly are passed on to the next assembly. If the argument is `"block"`, rays which are not claimed by any components in an assembly are discarded. This is the default behavior if no argument is given.

`begin_subassembly( )`
Begin a sub-assembly. The current transformation matrix is set equal to that of the parent assembly or sub-assembly.

`end_assembly( )`
Signify the end of an assembly. The current transformation matrix is now the global transformation matrix.

`end_subassembly( )`
Signify the end of a sub-assembly. The current transformation matrix is now that of the parent assembly or sub-assembly.

`get_TM_stack_top( inverse )`
This function returns the top transformation matrix on the stack as a set of nested tables. Transformation matrices are homogenous 4×4 matrices encoding both rotation and translation. If the `inverse` argument is present and true, the *inverse* of the CTM is returned.

`override( )`
In some cases it's nice to be able to provide values to the `Lua` program from the `aperture` parameter file (and thus the command line which invoked `aperture`). The `aperture` program parameter `override` is used to specify `Lua` code to be passed to the program. `override` may have three 'values':

- it may contain `Lua` code, which will be inserted into a `Lua` function called `override()`. For example, if the value of `override` is `"a=1;b=2;"`, the following code would be generated:

```
function override( )
  a = 1;
  b = 2;
end
```

- it may be empty, in which case an empty Lua function `override()` will be generated:

```
function override( )
end
```

- it may contain a filename (signified by the first character of the value being an "@" character, i.e., `@override.lua`). The file should contain a complete definition of a Lua function called `override()`. For instance, to duplicate the effects of the first item in this list, the file should contain

```
function override( )
  a = 1;
  b = 2;
end
```

It is guaranteed that there will always be an `override()` function, even if it is empty, so that a call to `override()` can be made with impunity. The following snippet of code illustrates how to use this functionality:

```
a = 2;
b = 3;
override( );
```

If the `override` parameter is empty, then the call to `override()` does nothing. However, suppose that the `override` parameter contains the string `"a=1;b=2;"`. In this case, the values of `a` and `b` will be changed as specified.

`print_assembly( )`
This function will print to the standard error stream the identification number, type and forward and backward transformation matrices for the current assembly.

`print_CTM( )`
Print the current transformation matrix to the error stream

`print_TM_stack( )`
Print the full transformation matrix stack to the error stream

`print_TM_stack_top( )`
Print the transformation matrix at the top of the stack to the error stream

`rotate( matrix, transform_type )`
Rotate the local coordinate system using the provided 3×3 matrix. The matrix is specified as a table, with each row in the matrix is specified as a subtable. For example, to rotate about the X axis by 90°:

```
rotate( { {  1,  0,  0 }, {  0,  0,  1 }, {  0, -1,  0 } } )
```

`transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

`rotate_x( delta_theta, transform_type )`
Rotate the local coordinate system about the X by the given angle (in radians). `transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

`rotate_y( delta_theta, transform_type )`
Rotate the local coordinate system about the Y axis by the given angle (in radians). `transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

`rotate_z( delta_theta, transform_type )`

Rotate the local coordinate system about the Z axis by the given angle (in radians). `transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

`transform( matrix, transform_type )`

Transform (rotate and translate) the local coordinate system using the provided 4×4 homogenous matrix. The matrix is specified as a table, with each row in the matrix is specified as a subtable. For example, to rotate around the point $(1, 0, 1)$ about the Z axis by $90°$:

```
transform( {
  {  0,  1,  0,  0},
  { -1,  0,  0,  0},
  {  0,  0,  1,  0},
  {  1, -1,  0,  1},
})
```

`transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

`translate( delta_x, delta_y, delta_z, transform_type )`

Translate the local coordinate system by the given amount. `transform_type` is an optional string (either `"intrinsic"` or `"intrinsic"`) indicating whether the transformation is *intrinsic* or *extrinsic*. It defaults to `"intrinsic"`.

Appendix B

Lua Aperture Instantiation Functions

These functions are provided by the back end modules to instantiate a component in the current local coordinate system. They are called from the user's `Lua` aperture definition program. Most of the functions optionally take a final argument, which is a `Lua` table.

`annulus( inner_radius, outer_radius, interior, transparent, [{label=name}])`
Insert an annulus lying in the $x - y$ plane, centered at the origin of the local coordinate system. The annulus has the specified inner and outer radii. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays, otherwise it will block them. The optional argument `label` may be used to specify the name of the component.

`block( )`
Block all rays. Use this as a diagnostic aid to stop the processing of any rays which have made it thus far in the current assembly.

`circle( radius, interior, transparent, [{label=name}] )`
Insert a circle lying in the $x - y$ plane, centered at the origin of the local coordinate system. The circle has the specified radius. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays, otherwise it will block them. The optional argument `label` may be used to specify the name of the component.

`ellipse( a, b, interior, transparent, [{label=name}] )`
Insert an ellipse lying in the $x - y$ plane, centered at the origin of the local coordinate system. The ellipse has the specified semi-major and semi-minor axes. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays, otherwise it will block them. The optional argument `label` may be used to specify the name of the component.

`ell( height, width, thickness, transparent, [{label=name}] )`
Insert an 'L' shaped component lying in the $x - y$ plane, centered at the origin of the local coordinate system. The 'L' has the specified height and width. The thickness refers to the width of the bars which make up the legs. If `transparent` is non-zero, the component will pass rays, otherwise it will block them. The optional argument `label` may be used to specify the name of the component.

`pass( )`
Pass all rays. Use this as a diagnostic aid to ignore any components which follow the current component in the current assembly.

`polygon( vertices, interior, transparent, [{label=name}] )`

Insert a polygon lying in the $x-y$ plane, centered at the origin of the local coordinate system. The polygon is specified as a list of x,y coordinates of the vertices ordered in either a clockwise or counterclockwise direction. Vertices must be specified as a `Lua` table, with each vertex a table with two elements, e.g. {*X,* Y}. For example:

```
vertices = { {0,0}, {1,0}, {0.5,1} }
polygon( vertices, 1, 1 )
```

If `interior` is non-zero, the component will operate on ray found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays, otherwise it will block them. The optional argument `label` may be used to specify the name of the component.

`print_ray( "xfrm" | "raw" | "raw_project" | "project" | "on" | "off" )`

Print the position and direction cosines of the ray (in that order, with the components in the order x, y, z) to the standard error stream. The argument controls what information to print or whether to shut off all printing:

`"xfrm"`	print out the ray position and direction cosines after being transformed to the current coordinate system
`"raw"`	print out the ray position and direction cosines in the external coordinate system
`"raw_project"`	project the ray to the component and print its position and direction cosines in the external coordinate system
`"project"`	print out the ray position and direction cosines after having been transformed to the current coordinate system and projected to the $x-y$ plane.
`"on" | "off"`	Turn on or off output from *all* calls to `print_ray`. It defaults to "on".

Previously `print_ray` was called `print_photon`. This usage is deprecated.

`rect( width, height, interior, transparent, [{label=name}] )`

Insert a rectangle lying in the $x-y$ plane, centered at the origin of the local coordinate system. The rectangle has the specified height and width. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays. The optional argument `label` may be used to specify the name of the component.

`strut( width, interior, transparent, [{label=name}] )`

Insert a strut lying in the $x-y$ plane, centered at the origin of the local coordinate system. The strut has infinite height and the specified width. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays. The optional argument `label` may be used to specify the name of the component.

`wedge( theta_min, theta_max, interior, transparent, [{label=name}] )`

This module creates an angular slice starting at `theta_min` and ending at `theta_max`, extending to infinity. The angles are specified in radians, and must be specified in a counter-clockwise direction. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays. The optional argument `label` may be used to specify the name of the component.

`wedger( radius, theta_min, theta_max, interior, transparent, [{label=name}] )`

This module creates an angular slice starting at `theta_min` and ending at `theta_max`, extending to the specified radius. The angles are specified in radians. The angles are specified in radians, and must be specified in a counter-clockwise direction. If `interior` is non-zero, the component will operate on

rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays. The optional argument `label` may be used to specify the name of the component.

`wedger2( radius_inner, radius_outer, theta_min, theta_max, interior, transparent, [{label=name}] )`
This module creates an angular slice starting at `theta_min` and ending at `theta_max`, extending form the specified inner radius to the specified outer radius. The angles are specified in radians. The angles are specified in radians, and must be specified in a counter-clockwise direction. If `interior` is non-zero, the component will operate on rays found within it, otherwise it operates on those that are outside. If `transparent` is non-zero, the component will pass rays. The optional argument `label` may be used to specify the name of the component.