OpenGI Programming Guide

Version 2.1

Christian Rau <rauy@users.sourceforge.net>

Contents

List of Tables

List of Figures

1. Introduction

1.1. OpenGI Overview

OpenGI is an open source library written in C for parameterizing triangular meshes and creating geometry images from this parameterization. As you are interested in OpenGI and reading this guide it is assumed that you have a basic understanding of geometry images. If not, the original paper by *Gu* et al.[GGH02] is a good place to start at. Some knowledge in the field of parameterization could be useful but is not neccessary.

The features of OpenGI include:

- Platform-independent and Open Source
- Easy to learn, OpenGL-like syntax and programming paradigms
- Working on 2-manifolds of arbitrary genus with a variable number of boundaries
- Implementing various parameterization algorithms, e.g. Mean Value Coordinates, Stretch Minimization or the original GIM algorithm
- Supporting multi-chart patchifications
- Easy and hardware accelerated creation of Geometry Images
- Creation of images not only for geometry but various generic attributes
- Detailed control and feedback of the whole parameterization and Geometry Image creation process
- Tight integration with OpenGL for easy and efficient data sharing

Figure 1.1 shows the typical OpenGI dataflow from mesh to geometry image with ellipses representing client data and rectangles representing major OpenGI operations. First you create an OpenGI mesh object from your mesh data. After cutting and parameterizing this mesh you can sample its attributes into OpenGI image objects, which operate on your client data. This data can then be used as geometry images to do whatever you want, including letting OpenGI render it with OpenGL. The remaining chapters of this guide will follow the major steps of this usual dataflow. But the dashed lines in figure 1.1 show that this control flow is everything else than strict. You can use OpenGI for parameterization only and then extract the parameterized mesh to use it for texture mapping or other techniques involving parameter coordinates. Or you can specify your own mesh cut and maybe an existing parameterization on this cut and use OpenGI only for sampling.

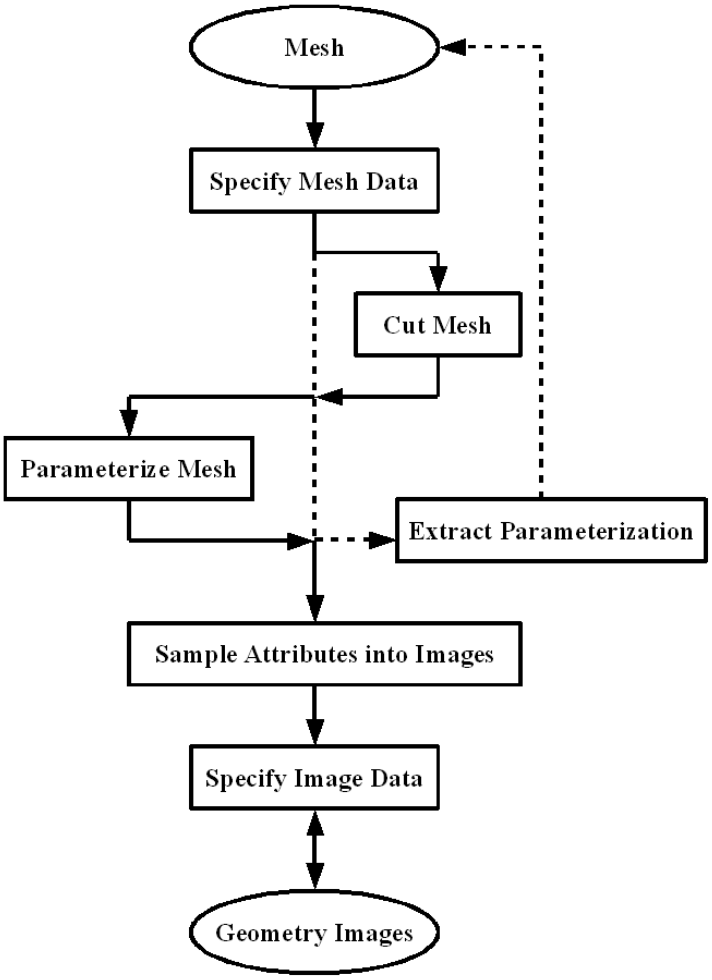

Figure 1.1.: OpenGI dataflow

1.2. General API Design

The command syntax and programming paradigms used in OpenGI are heavily based on OpenGL. Originally this was chosen to give the developer of the library a clear design principle, but a useful side effect is that people understanding OpenGL, which is quite likely as we're moving in the field of computer graphics, have no problems learning OpenGI. Actually OpenGL was not only the design guide but is also tightly integrated into this library making its use within existing graphics software as easy and efficient as possible. So if you are fit in OpenGL, you won't need much time to understand the workings of OpenGI.

Throughout this guide you will find types like `GIuint`, `GIfloat`, etc. These are only `typedef`s of common C datatypes:

```
typedef unsigned int    GIenum
```

```
typedef unsigned char  GIboolean
typedef unsigned int   GIbitfield
typedef char           GIchar
typedef signed char    GIbyte
typedef short          GIshort
typedef int            GIint
typedef int            GIsizei
typedef unsigned char  GIubyte
typedef unsigned short GIushort
typedef unsigned int   GIuint
typedef float          GIfloat
typedef double         GIdouble
typedef unsigned short GIhalf
typedef void           GIvoid
```

For clear use of the `GIboolean` datatype OpenGI defines the two constants `GI_TRUE` and `GI_FALSE` which just evaluate to 1 and 0.

OpenGI also defines some constants for identifying datatypes, e.g. `GI_UNSIGNED_BYTE` or `GI_FLOAT`. These all evaluate to the same values as the corresponding OpenGL constants. However, other OpenGI constants that have the same name as OpenGL constants (but with `GI_` instead of `GL_`) do not have the same value as their OpenGL counterparts.

Like OpenGL, OpenGI was primarily designed as a state machine. Most of the configuration parameters for the various functions (set by `gi...Parameter`) and other settings are represented as global state variables and modifying them has effect on all following calls to the functions using them. So, when calling a function always be sure in what state OpenGI currently is. In the appendix you can find the default values for all state variables. Note, that these values can **only** be changed by you calling the respective functions and **not** by any side effects of other functions (except variables giving feedback of the success of an algorithm, like `GI_SAMPLED_ATTRIBS`).

One of the main design principles of OpenGI was the object model known from OpenGL's texture or buffer objects. They are a little decoupling of the global state, representing all the data, the algorithms work on and modify in distinct objects. They are created by calling `giGen...` and deleted by `giDelete...`. To the user they are represented by a numeric object handle. There is allways one object bound for a given object type (or none, represented by the NULL-handle), using `giBind...`, and all algorithms working on that object type refer to this bound object. So, to do a bunch of things with an object you only need to bind it once and call the functions for modifying it without always passing the object explicitly. The two object types used in OpenGI are triangular meshes and rectangular images.

2. Basic Functionality

2.1. Context Management

The current OpenGI context encapsulates all the OpenGI state of an application. There can be more than one context in an application but only one can be the active one. By

```
GIcontext giCreateContext()
```

you can create a new context. The function returns the new context and automatically makes it the active context. To change or query the active context use

```
void      giMakeCurrent(GIcontext context)
GIcontext giGetCurrent().
```

Finally you can delete a context and free its resources by calling

```
void giDestroyContext(GIcontext context)
```

If the deleted context is the current active context it is unbound automatically. Note that calling any other OpenGI functions except these when there is no active context results in undefined behaviour, but most likely an access violation.

2.2. State Management

Nearly every value of the current OpenGI state can be queried by one of the functions

```
void giGetBooleanv(GIenum pname, GIboolean *params)
void giGetIntegerv(GIenum pname, GIint *params)
void giGetFloatv  (GIenum pname, GIfloat *params)
```

specifying the state to query and the address to store its value at. The possible values for *pname* are listed in tables D.1 to D.3 in appendix D. Note that every boolean state can also be queried with `giGetIntegerv` and `giGetFloatv` and every integer state can be queried with `giGetFloatv`, but not vice versa.

Certain features of OpenGI can be activated, deactivated and queried with

```
void giEnable (GIenum pname)
void giDisable(GIenum pname)
GIboolean giIsEnabled(GIenum pname).
```

The value of every OpenGI constant can be queried given its name using

```
GIenum giGetEnumValue(const GIchar *name).
```

2.3. Error Handling

When calling OpenGI functions there can occur different kinds of errors caused by inappropriate use of its functionality. OpenGI only stores the last error it encountered. The error code of this error can be queried by

```
GIenum giGetError().
```

Table 2.1 shows the possible errors.

error code	description
GI_NO_ERROR	everything's fine
GI_INVALID_ENUM	constant not accepted by the function
GI_INVALID_OPERATION	operation illegal in current state
GI_INVALID_VALUE	numeric value out of range
GI_INVALID_ID	object with specified id does not exist
GI_INVALID_MESH	input mesh is not a connected 2-manifold
GI_INVALID_CUT	mesh cut incomplete
GI_NUMERICAL_ERROR	numerical stability problem
GI_UNSUPPORTED_OPERATION	attempted operation not supported by the system
GI_INVALID_-PARAMETERIZATION	no parameterization onto the unit square

Table 2.1.: Possible errors and their meaning

When querying the last error it will be reset so that the next call of this function returns `GI_NO_ERROR`. For a given error code you can retrieve a descriptive error string using

```
const GIchar* giErrorString(GIenum error).
```

For a better control over the thrown errors you can register a callback function, called whenever an error occurs with the function

```
void giErrorCallback(GIerrorcb fn, GIvoid *data)
```

where *GIerrorcb* is just a

```
typedef void (GICALLBACK *GIerrorcb)(GIenum, GIvoid*)
```

and *data* a pointer to custom user data. The callback function is then called with the error code and the user data as arguments. When no error callback is set, which is the default, and OpenGI was built with `OPENGI_DEBUG_OUTPUT` defined, a descriptive error string is written to the console.

2.4. Multithreading

If OpenGI was compiled with multithreading support, some functions may profit from using multiple threads, especially on a multi-core architecture. Multithreading can be enabled or disabled by calling `giEnable`/`giDisable` with the constant `GI_MULTITHREADING`.

It is enabled by default but it only takes effect if OpenGI was compiled with multithreading support, which is true if it was built either on a Win32 platform or on a platform containing the *pthreads* library. Functions that use multiple threads are `giParameterize` and `giSample` with software sampling.

Although it may use mutlithreading internally OpenGI is absolutely **not thread-safe**. You should not use OpenGI functions in two or more different threads simultaneously. It is not enough to just work on different meshes, as the OpenGI context and the current mesh binding etc. are application-global data. Of course you can use OpenGI in different threads, but then you should guard the OpenGI calls by critical sections, so that no two threads are in an OpenGI function at the same time.

3. Triangular Meshes

3.1. Basic Mesh Management

The triangular mesh is the central datastructure on which OpenGI operates. Note that OpenGI only supports **2-manifolds** but these can have an arbitrary number of boundaries and an arbitrary genus. The mesh should also be connected, meaning it does not consist of more than one unconnected parts, as such a "multi-mesh" cannot be parameterized. Since version 2.0 of the library a mesh can consist of more than one parameterized patch. But for now you do not have to worry about patches, as OpenGI's patch semantics are thoroughly explained in chapter 4 and are never needed when the mesh consists of only one patch.

In the client application meshes are represented by mesh IDs which are actually just numbers. To create a new mesh object and get its ID you can use the functions

```
GIuint giGenMesh()
void   giGenMeshes(GIsizei n, GIuint *meshes)
```

returning either one ID or storing n IDs in *meshes*. With

```
void giBindMesh(GIuint mesh)
```

you can then bind this mesh as the active one so that all operations work on this mesh. OpenGI generates a `GI_INVALID_ID` error if the mesh with the specified ID does not exist. To query the currently bound mesh ID use `giGetIntegerv` with the parameter `GI_MESH_BINDING`. To delete one or more meshes and free their resources just call

```
void giDeleteMesh  (GIuint mesh)
void giDeleteMeshes(GIsizei n, const GIuint *meshes)
```

which unbinds the mesh if it was the active one. Note that calling any OpenGI functions working on meshes will generate a `GI_INVALID_OPERATION` error if no mesh is bound (actually mesh 0 is bound).

3.2. Vertex Attributes

Before creating the mesh we have to take a look at OpenGI's attribute system. Since version 2.0 the attribute system has been made more generic. Each vertex of a triangular mesh can have a number of indexed generic vertex attributes. These attributes can be anything from positions to normals or colors or whatever, OpenGI does not care about their semantics. The maximum number of vertex attribute channels can be queried by calling `giGetIntegerv` with `GI_MAX_ATTRIBS` and is at least 16 at the moment.

semantic	description	default
GI_POSITION_ATTRIB	vertex position	0
GI_PARAM_ATTRIB	parameter coordinates	14
GI_PARAM_STRETCH_ATTRIB	parameterization stretch	15

Table 3.1.: Special vertex attribute semantics

Although this attribute system is quite generic there have to be some attributes which are stored and treated specially and thus have to have a special semantic. These are the vertex positions, their parameter coordinates and parameterization stretch (more about this later). This semantic can be established for a given attribute channel by calling

```
void giBindAttrib(GIenum semantic, GIuint attrib)
```

with *semantic* being one of the constants from table 3.1 and *attrib* between 0 and GI_MAX_ATTRIBS−1 specifying the attribute channel to use for this semantic. Only one attribute channel can be bound to a certain semantic at a time and table 3.1 also lists the initial bindings. It is important to note, that these semantic bindings are only relevant for mesh creation. When the mesh data is created (see next section) the current semantic bindings are **fixed** for the mesh and every function working with attributes will assume the semantic bindings of the current mesh and **not** the global ones if different.

Various global per-attribute state can be queried by

```
void giGetAttribbv(GIuint attrib, GIenum pname, GIboolean *params)
void giGetAttribiv(GIuint attrib, GIenum pname, GIint *params)
void giGetAttribfv(GIuint attrib, GIenum pname, GIfloat *params)
```

with the constants from tables D.4 to D.6 for *pname* (see appendix D).

3.3. Mesh Creation

To create the actual data for a mesh several steps have to be taken. Meshes are created from vertex arrays either indexed or non-indexed, just as you would draw them in OpenGL or whatever 3D-API you like. Note that the only primitive type supported is a triangle set. Other primitives as quads or triangle strips known to 3D-APIs are not supported. To set the vertex arrays to use for mesh creation just call

```
void giAttribPointer(GIuint attrib, GIint size,
                     GIboolean normalized, GIsizei stride,
                     GIfloat *pointer)
```

In this function *attrib* specifies the attribute channel, *size* stands for the number of components in [1,4] and *normalized* says if the attribute should be renormalized after interpolation. The *stride* argument specifies the number of values between the start of two consecutive elements. If the elements are tightly packed (corresponding to *stride* equaling *size*) you may also set *stride* to 0. Finally *pointer* is the pointer to a user controlled array of single precision floating point values containing the data. As this

data is not accessed until the actual mesh creation the vertex arrays are global state and no per mesh state. The current pointer for each array can be obtained by calling

```
void GetAttribPointerv(GIuint attrib, GIfloat **params).
```

After (or before) the data pointers for the neccessary attribute channels are set, they have to be enabled or disabled with

```
void giEnableAttribArray(GIuint attrib)
void giDisableAttribArray(GIuint attrib)
```

Before mesh creation some requirements have to be met. Of course every enabled attribute channel should have a valid data pointer. Moreover the attribute channel bound to `GI_POSITION_ATTRIB` always has to be enabled and have at least 3 components, as every mesh needs at least vertex positions in 3-space. The mesh can also be given an existing patchification and parameterization by enabling and setting the attribute bound to `GI_PARAM_ATTRIB`, but this attribute has to have at least 2 components and has to meet some additional requirements (see 4 and 5 for more information). The attribute bound to `GI_PARAM_STRETCH_ATTRIB` always has to be disabled, as the parameterization stretch is computed internally and cannot be specified by the user.

Now everything is ready for mesh creation. To create a mesh from the current vertex attribute arrays, call one of the functions

```
void giIndexedMesh(GIuint start, GIuint end,
                   GIsizei count, const GIuint *indices)
void giNonIndexedMesh(GIint first, GIsizei count).
```

The first function creates the mesh by indexing into the arrays with the *count* indices in the *indices* array. The arguments *start* and *end* are just hints to keep the memory consumption of the function low. They should contain the minimal and maximal index used or can just be set to 0 and *vertex_array_size*-1 when you do not know the range of vertices or use the whole array. With the second function you just walk through the arrays taking every tuple of three consecutive elements as a triangle. Here *first* stands for the vertex to start at and *count* is the number of vertices processed. If attempting to create a **non-manifold** mesh, e.g. a mesh with edges adjacent to more than two triangles, vertices belonging to more than one boundary, or mesh parts only connected by a vertex, a `GI_INVALID_MESH` error is thrown and the mesh creation fails.

You can also use the function

```
void giCopyMesh(GIuint mesh)
```

to create the current mesh as an exact copy of the mesh with the ID specified in *mesh*. Note that OpenGI generates a `GI_INVALID_ID` error if the source mesh does not exist.

There have to be made some remarks on the memory consumption. Due to the rather complex mesh processing done during parameterization, the mesh datastructure is quite complex and memory consuming. On 32-bit platforms the mesh can take about 400-600 bytes per vertex and about 700-900 bytes on 64-bit platforms, depending on the number of different attributes used and their size (usually around 450 or 750). Additionally the parameterization can create about 250-350 bytes per vertex of temprorary data,

depending on the algorithm used. So parameterizing a mesh with 1 million vertices requires you to have about 1 GB of free memory.

3.4. Vertex Subsets

When creating meshes it is sometimes necessary to define some vertices (usually very few) that have a certain property. Because it would be quite wasteful to do this by a per vertex flag this is accomplished by vertex subsets. A vertex subset is an index array containing the indices of all vertices in the vertex array with a certain property.

The semantics of vertex subsets are nearly like those of attribute arrays. A certain subset used for mesh creation is set by

```
void giVertexSubset(GIenum subset, GIsizei count,
                    GIboolean sorted, const GIuint *indices)
```

where *indices* contains the *count* indices of the vertices in the subset. The *sorted* argument specifies if the index array is sorted. If the array is sorted or you have the ability to sort it you should do this, otherwise in every call to `gi[Non]IndexedMesh` a local copy of the subset is made and then sorted if activated. To activate a subset for use during mesh creation just call `giEnable` with the same parameter used for the *subset* argument of `giVertexSubset`. These arguments can be one of the following constants:

GI_EXACT_MAPPING_SUBSET: During parameterization vertices in this subset are parameterized to exact texels if they lie on the parameterization boundary, so their exact value stands in the geometry image. Usually this is not required as OpenGI takes care of such vertices where it is necessary (at cut intersections). But if you want to separate a mesh into different patches and you do not want to use OpenGI's patch functionality, the exact mapping of certain vertices can aid the stitching of adjacent images.

GI_PARAM_CORNER_SUBSET: vertices in this subset are mapped to the corners of the parameterization domain if they lie on the parameterization boundary, but this works only for patches that have **exactly four** such vertices on their boundary. This subset is useful if you want to give the mesh an existing patchification (but no parameterization) and you know it is a quadrilateral patchification, so the patch intersections are mapped to the image corners and OpenGI only has to parameterize the interior.

The currently set pointer for a vertex subset can be retrieved by calling

```
void giGetPointerv(GIenum pname, GIvoid **params)
```

with *pname* being one of the constants from the above list.

Some remarks on the indices in the subset: If a vertex, that belongs to the subset is duplicated because of different normals or something like this, you should put all index synonyms of that vertex into the subset or at least the index that is processed first during mesh creation, which is the smallest for `giNonIndexedMesh` or the first in the index array for `giIndexedMesh`.

3.5. Mesh Retrieval

Every mesh property can be retrieved by one of these functions

```
void giGetMeshbv(GIenum pname, GIboolean *param)
void giGetMeshiv(GIenum pname, GIint *param)
void giGetMeshfv(GIenum pname, GIfloat *param)
```

specifying the property to query and the address to store its value at. Tables 3.2 to 3.4 list the properties that can be queried. The last column shows if the respective property can also be retrieved per patch (see 4 for more information on patches). The per-patch value may vary from the per-mesh value.

In addition to the global mesh properties there are some per-attribute mesh properties, which are queried by

```
void giGetMeshAttribbv(GIuint attrib, GIenum pname, GIboolean *params)
void giGetMeshAttribiv(GIuint attrib, GIenum pname, GIint *params)
```

given the attribute channel to query the property for. The allowed values for *pname* are listed in table 3.5. It makes no difference if you call these functions for the whole mesh or just one patch, the results are the same.

There are two functions to extract the mesh data back to a vertex/index array representation. They can also be used to retrieve a single patch's data only. With

```
void giGetNonIndexedMesh(GIuint *vcount)
```

you can extract the mesh into vertex arrays with three consecutive vertices representing a triangle. The number of vertices extracted will be stored in *vcount*. The vertex attributes are written to the arrays specified by `giAttribPointer` and only the currently enabled attributes are extracted (using `gi[Enable/Disable]AttribArray`). This way only a subset of the mesh attributes can be extracted. But to prevent errors, the number of components and bound semantic of every enabled attribute channel has to match the corresponding per-mesh properties. If a single attribute channel does not match, the whole extraction fails and a `GI_INVALID_OPERATION` error is thrown.

To extract the mesh into vertex and index arrays use

```
void giGetIndexedMesh(GIuint *vcount, GIuint *icount, GIuint *indices)
```

with the user allocated array *indices* which stores the index values. The number of extracted indices is written into *icount*. Due to attribute discontinuities the number of vertices you retrieve depends on the enabled attributes and is determined on the fly and not nesseccarily equal to the number of vertices you created the mesh from. Because of this, you may not know the required size of the attribute arrays. To retrieve this size you can call `giGetIndexedMesh` with an *indices* argument of `NULL`. This will only compute the number of extracted vertices and indices based on the enabled attribute channels and will not write any data, but keep in mind, that the attribute specifications (number of components and semantic) still have to match. After this you can allocate your vertex and index arrays and call `giGetIndexedMesh` a second time. But you can also just allocate the arrays to a size of $3\times\#\text{faces}$, which will always suffice.

constant	description	patch
GI_HAS_PARAMS	mesh has valid parameterization	x
GI_HAS_CUT	mesh has valid cut	x

Table 3.2.: Boolean mesh properties

constant	description	patch
GI_PATCH_COUNT	number of patches	
GI_FACE_COUNT	number of faces	x
GI_EDGE_COUNT	number of edges	x
GI_VERTEX_COUNT	number of vertices	x
GI_POSITION_ATTRIB	attribute channel used for vertex positions	x
GI_PARAM_ATTRIB	attribute channel used for parameter coords	x
GI_PARAM_STRETCH_ATTRIB	attribute channel used for param stretch	x
GI_PARAM_STRETCH_METRIC	last metric used for computing per vertex stretches or 0 if no stretches computed	x
GI_PARAM_RESOLUTION	resolution of parameter domain or 0 if no parameterization	x
GI_TOPOLOGICAL_SIDEBAND_LENGTH	length of topological sideband	
GI_TOPOLOGICAL_SIDEBAND	topological sideband data	
GI_ACTIVE_PATCH	ID of active patch or GI_ALL_PATCHES if none	

Table 3.3.: Integer mesh properties

constant	description	patch
GI_AABB_MIN	minimal vertex of axis aligned bounding box	
GI_AABB_MAX	maximal vertex of axis aligned bounding box	
GI_RADIUS	radius of bounding sphere around origin	
GI_MIN_PARAM_STRETCH	minimum per vertex stretch value or 0 if no per vertex stretch values existing	x
GI_MAX_PARAM_STRETCH	maximum per vertex stretch value or 0 if no per vertex stretch values existing	x

Table 3.4.: Floating point mesh properties

constant	description
GI_HAS_ATTRIB	mesh has data for this attribute channel
GI_ATTRIB_NORMALIZED	attribute should be renormalized after interpolation
GI_ATTRIB_SIZE	number of components
GI_ATTRIB_SEMANTIC	semantic attribute is bound to or GI_NONE

Table 3.5.: Boolean and integer per-attribute mesh properties

4. Mesh Cutting

4.1. Patch Handling

Before the mesh can be parameterized it has to be cut into a topological disk, as a sphere, for example, cannot be unrolled to the plane. You have to cut it open before you can fold it into the parameter plane. This cut is then mapped to the boundary of the parameter domain, in our case the unit square. Since OpenGI 2.0 the cut graph may contain circles, meaning the mesh can consist of a number of disjoint patches that can be parameterized and sampled individually. But this is an all or nothing contract. **Either** the mesh consists of one patch only and therefore the cut graph is a tree **or** the cut graph does not contain any degree-1 nodes (open ends) and consists of a single connected component, meaning the patches already represent topological disks.

The number of patches a mesh has can be queried by calling `giGetMeshiv` with `GI_PATCH_COUNT` and the function

```
void giMeshActivePatch(GIint patch)
```

can be used to select a patch as the active one, by its 0-based index. After this call every function working on mesh data (except for the mesh creation and copy) refers to **this patch only**. To work on the whole mesh again, call `giMeshActivePatch` with a *patch* argument of `GI_ALL_PATCHES`. The active patch is a per-mesh state and can be queried with `giGetMeshiv` and a *pname* of `GI_ACTIVE_PATCH`, which is `GI_ALL_PATCHES` by default. If the mesh consists of a single patch only, it makes no difference if patch 0 or no patch is selected. Therefore, as long as no multi-chart patchification is needed, you do not have to bother with these patch semantics.

To give the mesh an existing cut or patchification you can either supply the parameter coordinates on creation or copy them from another attribute (see 5). If you do not know the parameterization and only want to set the cut you can do this by using the patch ID as parameter coordinates (exptended to 2 components), for example. The mesh will not get a valid parameterization but the cut will be extracted from the attribute discontinuities (edges with different attribute values on either side). After this you can parameterize the patches with OpenGI, as cutting and parameterization are seperated since version 2.0. When specifying the cut yourself and using OpenGI for parameterization, also consider using the `GI_PARAM_CORNER_SUBSET` if you know the corners of the parameterization boundaries (e.g. when using a quadrilateral patchification).

4.2. Cutting Algorithms

The actual mesh cutting is realized by calling

```
void giCut()
```

and the algorithm used for cutting can be selected with

```
void giCutterParameteri(GIenum pname, GIint param)
```

with *pname* being `GI_CUTTER` and *param* being one of the following methods:

GI_INITIAL_GIM: This algorithm works on meshes of arbitrary genus and with a variable number of boundaries. It actually represents the initial cut computed by *Gu* et al. in their original GIM parameterization algorithm [GGH02], therefore its name. When the mesh consists of more than one unconnected parts the parameterization fails with a `GI_INVALID_CUT` error, as a connected cut cannot be computed. On a genus-0-mesh with one boundary the resulting cut is actually that boundary.

GI_CATMULL_CLARK_SUBDIVISION: This method realizes standard Catmull-Clark subdivision presented in [CC78] and creates a single patch for every quadrilateral after the first iteration, resulting in a completely quadrilateral patchification. The overall number of iterations can be controlled by calling `giCutterParameteri` with `GI_SUBDIVISION_ITERATIONS`. Every patch already comes with a valid uniform parameterization, such that a geometry image of size $(2^i+1)\times(2^i+1)$ exactly captures the patch's geometry after $i+1$ iterations of Catmull-Clark subdivision. Unlike other cutting algorithms the subdivision process changes the geometry of the mesh, but it is completely restored when deleting (or overwriting) the cut. But note, that although the vertex positions are adapted to realize a smooth surface, every other attribute is just interpolated linearly, meaning the avegare of the incident vertices for edges and faces respectively. You should also keep an eye on the memory comsumption, as the internal data for a patch takes about 350-450 bytes and 3 patches are created for every triangle of the input mesh. Figure 4.1 shows the results of subdividing an octahedron.

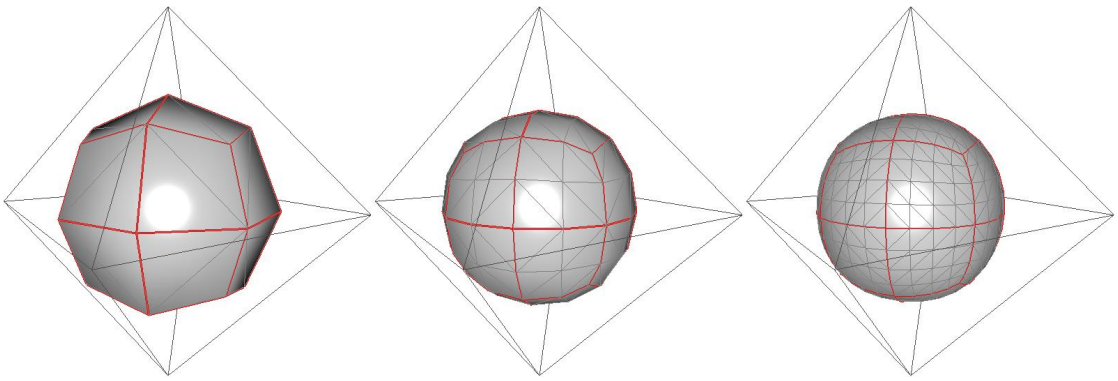

Figure 4.1.: Catmull-Clark subdivision of an octahedron after 1, 2 and 3 iterations. Cut rendered in red.

5. Parameterization

5.1. General Parameterization Issues

After the mesh has been cut (and possibly patchified) it can be parameterized. Parameterization will usually be the most time consuming task during the whole process of geometry image creation. It could be useful if you have a little background in parameterization but it is not necessary for using the parameterization functionality of OpenGI. Nevertheless the SIGGRAPH course by *Hormann* et al.[HLS07] is a good starting point. Parameterization can be invoked on the current bound mesh by calling

```
void giParameterize()
```

This function can be used to parameterize a single patch individually or all patches of the mesh at once, by selecting the whole mesh. If this function was successful, the mesh will have valid per vertex parameter coordinates. OpenGI only supports the unit square as parameter domain, as we mainly want to create rectangular geometry images. There is no possibility to make parameterizations with an arbitrary border. The actual parameterization process can be controlled in various ways by the functions

```
void giParameterizerParameterb(GIenum pname, GIboolean param)
void giParameterizerParameteri(GIenum pname, GIint param)
void giParameterizerParameterf(GIenum pname, GIfloat param)
```

During the parameterization there may be certain vertices that need to be mapped to exact texels, to prevent small gaps in the reconstructed geometry images. To define what an exact texel is, the resolution of the parameter domain needs to be set. This can be achieved by calling `giParameterizerParameteri` with `GI_PARAM_RESOLUTION`. This defaults to 33 and should be set to the desired resolution of the geometry (position) image. When intending to sample or render the geometry image in different resolutions, e.g. for doing LOD, the parameter resolution should be set to the lowest intended geometry image resolution. In this case you should consider using resolutions of the form $(2^m+1)\times(2^m+1)$, so that all texels of the lowest resolution are also part of all higher resolutions. When taking exactly mapped vertices into account you should also think about using the software mode for sampling (see A for more information). Note, that too low a param resolution may affect the quality of the boundary parameterization, when there are many cut intersections.

After a successful parameterization you can also retrieve the active patchs's topological sideband (see [GGH02]). This is a list of integers representing the cut paths along the boundary of the parameter domain, starting at the lower left corner and going counterclockwise. For each cut path an ID is stored and the path's length in texels. If the mesh has only one patch, the ID identifies the cut path. So, if a cut path ID is in

the list twice, the path is in the internal of the mesh and traversed twice in different directions. This way you can identify the regions on the geometry image border that are adjacent in the reconstructed mesh. If the mesh has multiple patches, the ID identifies the neighbouring patch along this path, with -1 for no patch. The sum of the cut path lengths is allways equal to $4(r-1)$, with r being the resolution of the parameter domain during parameterization and **not** the resolution of the actual geometry image (there need not yet be any image), although often these values are the same. The length of this list can be obtained by calling `giGetMeshiv` with `GI_TOPOLOGICAL_SIDEBAND_LENGTH` and the list itself with `GI_TOPOLOGICAL_SIDEBAND`. Note, that, when the length is n, the array storing the list should at least have the length $2n$, as with each cut path id its length in texels is stored.

Internally the parameterization algorithms result in solving systems of linear equations. These systems are by default solved by CG in the symmetric case and BiCGStab in the unsymmetric case. Although, due to preconditioning, it should never happen that BiCGStab crashes for normal meshes, it could possibly happen some day, resulting in a `GI_NUMERICAL_ERROR`. In this case the advanced user can set the solving algorithm to the slower GMRES(25) by calling `giParameterizerParameteri` with `GI_UNSYMMETRIC_SOLVER` and setting it to `GI_GMRES` instead of `GI_BICGSTAB`.

5.2. Parameterization Algorithms

After the cut has been parameterized we come to the core of the whole work, the computation of UV-coordinates for the interior vertices based on the values on the cut. For the actual parameterization you have the choice between different algorithms, set by calling `giParameterizerParameteri` with `GI_PARAMETERIZER`:

GI_FROM_ATTRIB: This copies the parameter coordinates from another attribute. The attribute channel to use can be selected with `giParameterizerParameteri` and a *pname* of `GI_PARAM_SOURCE_ATTRIB`. But some restrictions are imposed on this attribute: The mesh has to have data for it and it has to have at least two components. Of course it cannot be the attribute bound to `GI_PARAM_ATTRIB` or `GI_PARAM_STRETCH_ATTRIB`. As this discards the current cut and patchification and recomputes it completely, it can only be used on the whole mesh and not on individual patches, but it can also be used on meshes that do not have a valid cut. It computes a cut via the attribute discontinuities and these have to meet the same requirements stated in the previous chapter. Only if its parameter coordinates build a bijective mapping onto the unit square will a patch be considered as parameterized, but you can still use it for creating the cut.

GI_TUTTE_BARYCENTRIC: This is the simplest convex-combination parameterization, developed by *Tutte* [Tut63]. It uses uniform edge weights that do not adapt to the mesh's geometry, so it produces quite unsmooth results.

GI_SHAPE_PRESERVING: This parameterization was presented by *Michael Floater* in [Flo97] and produces much smoother results than the previous one.

GI_DISCRETE_HARMONIC: This parameterization from *Eck* et al.[EDD+95], uses Discrete Harmonic coordinates. Like the previous one it is conformal (angle-preserving).

GI_MEAN_VALUE: This other parameterization by *Floater*, using Mean Value coordinates [Flo03], looks quite the same as the other angle-preserving parameterizations. It should at least produce results as good as that from the Shape Preserving parameterization, but the weight computation should be faster, although not very significantly.

GI_DISCRETE_AUTHALIC: This parameterization from *Desbrun* et al.[DMA02] uses Wachspress coordinates. In theory it should account for the area distortion but in practice it looks like the other conformal parameterizations.

GI_INTRINSIC: The Intrinsic parameterization by *Desbrun* et al.[DMA02] is the generalization of the Discrete Harmonic and Discrete Authalic parameterizations, trying to account for angle and area distortion. It is actually a mix of both methods. The mixing weights of the two parameterizations can be set by calling `giParameterizerParameterf` with the constants `GI_CONFORMAL_WEIGHT` and `GI_AUTHALIC_WEIGHT`, which both default to 1.

GI_STRETCH_MINIMIZING: When you want to create geometry images you should actually use this parameterization or the next, as these try to minimize parameterization stretch and not only angle distortion. This is a very simple but also fast stretch minimizing parameterization, developed by *Yoshizawa* et al.[YBS04]. It starts with a normal convex-combination map and iteratively changes the linear system based on the per vertex stretch values. Call `giParameterizerParameteri` with `GI_INITIAL_PARAMETERIZATION` and the constants 3 to 6 of this list to set the starting parameterization, although this does not make much difference. The stretch metric to use is set with the same function, using `GI_STRETCH_METRIC` and one of the constants from table 5.1, defaulting to `GI_RMS_GEOMETRIC_STRETCH`. Finally you can adjust how the stretch values influence the linear system per adaption step. When setting `GI_STRETCH_WEIGHT` with `giParameterizerParameterf` to a value lower than 1, the parameterization will take longer but might produce better results. This weight defaults to 1 and has to lie in [0,1]. When using `GI_COMBINED_STRETCH` as stretch metric, you can control how much the area distortion influences the stretch energy by setting `GI_AREA_WEIGHT`, which defaults to 1 and should be larger than 0.

GI_GIM: This finally is the original GIM algorithm developed by *Gu* et al.[GGH02]. It works by iteratively improving the cut and computing stretch minimizing parameterizations in each iteration. As it internally uses the previous algorithm, all parameters set for this will also take effect here. As it changes the cut, it cannot be used on a mesh consisting of more than one patch.

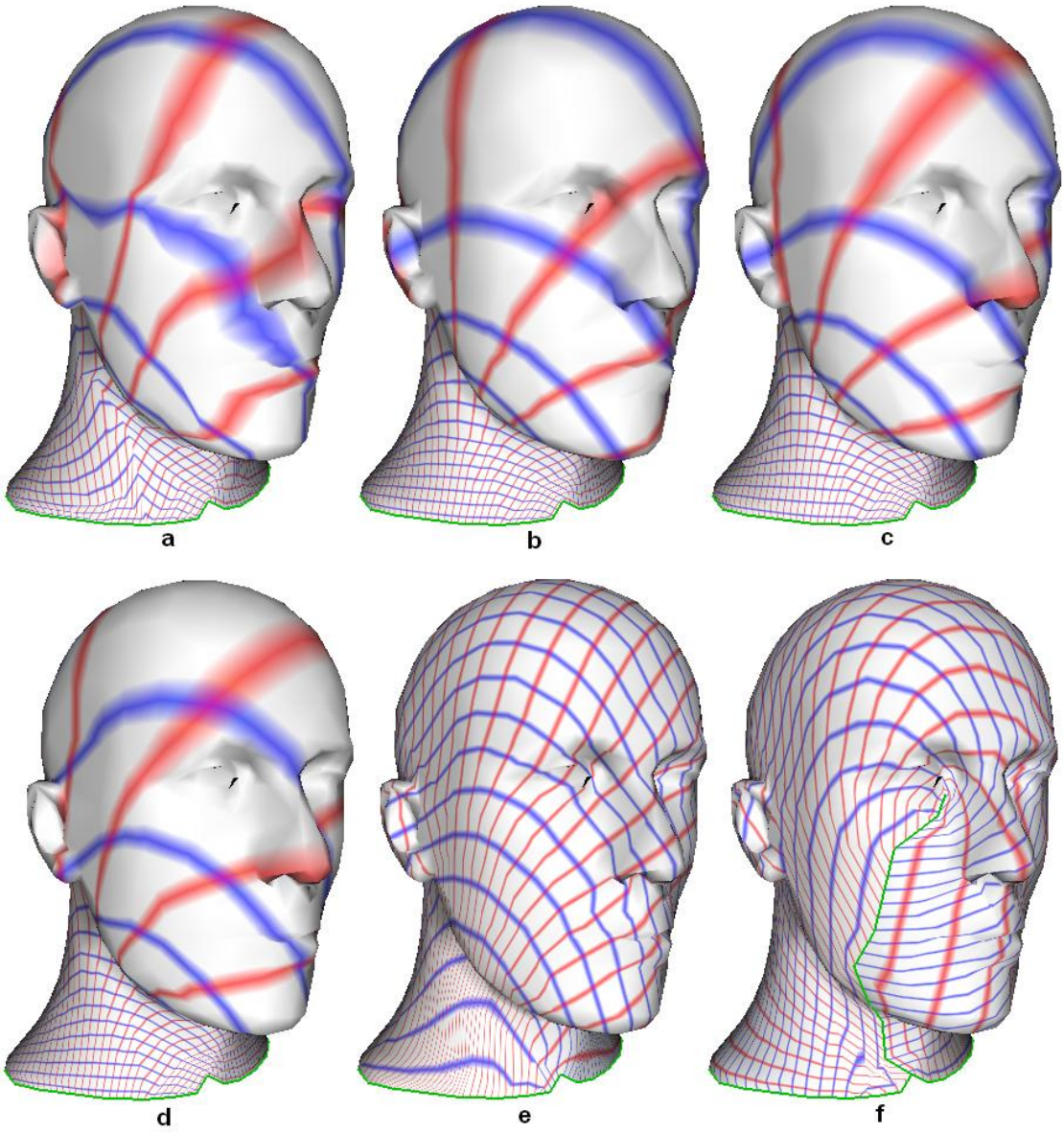

Figure 5.1.: UV lines for head model parameterized by Uniform (a), Discrete Harmonic (b), Mean Value (c), Discrete Authalic (d), Stretch Minimizing (e) and GIM parameterization (f). The green curve is the parameterization border (cut).

Note, that the Discrete Harmonic and Discrete Authalic (and therefore also the Intrinsic) parameterization may produce negative weights on certain extreme triangle constellations. This would result in incorrect triangle flips in parameter space. Furthermore can the Discrete Authalic parameterization produce a linear system that may not be solvable. But with the Shape Preserving or Mean Value parameterization you are always on the safe side. Most times you will be well advised with just using the default values for parameterization, but for more complex meshes it could be a good idea to try the GIM parameterization or change some other parameters. In figure 5.1 you can see the head model parameterized by different algorithms.

5.3. Parameterization Stretch

The stretch is a measure for the quality of the parameterization as it gives information about the stretch (of what kind ever) of a reconstruction when sampling the parameter domain uniformly. There exist various metrics to measure the stretch, which have different goals. The metrics that can be used for stretch minimization are listed in table 5.1. The stretch of a mesh's or patch's parameterization can be retrieved by calling `giGetMeshfv` with one of these constants. But note, that the value is undefined if called on the whole mesh and not all patches have a valid parameterization.

constant	**description**
`GI_MAX_GEOMETRIC_STRETCH`	L_∞ stretch as defined in [SSGH01]
`GI_RMS_GEOMETRIC_STRETCH`	L_2 stretch as defined in [SSGH01]
`GI_COMBINED_STRETCH`	combined stretch metric introduced in [DMK03]

Table 5.1.: Metrics for measuring parameterization stretch

The parameterization stretch can also be measured locally, as a scalar vertex attribute (guess, it is the one bound to the `GI_PARAM_STRETCH_ATTRIB` semantic), but in order to work with this attribute, the per-vertex stretch values have to be computed by calling

```
void giComputeParamStretch(GLenum metric)
```

which computes the values with the specified metric for the currently bound mesh or only its active patch. There can only be stretch values for one metric at a time and calling this function with a different metric overwrites the previous values. Moreover the values are only computed if neccessary. If the stretch values are still valid or have been computed during parameterization, there is no need to compute them again. The currently active metric can be queried with `giGetMeshiv` and a *pname* of `GI_PARAM_STRETCH_METRIC`, returning 0 if there are no stretch values or different ones for different patches (if called for the whole mesh). Finally the mesh's or patch's minimal and maximal per-vertex stretch values can be retrieved by calling `giGetMeshfv` with either `GI_MIN_PARAM_STRETCH` or `GI_MAX_PARAM_STRETCH`.

5.4. Callback Functions

As parameterization can be quite time consuming depending on the mesh size and the algorithm used, OpenGI provides callback hooks for you to get feedback during parameterization. These are set by

```
void giParameterizerCallback(GIenum which,
                             GIparamcb fn, GIvoid *data)
```

specifying the function to call and optional user data, where `GIparamcb` is just a

```
typedef GIboolean (GICALLBACK *GIparamcb)(GIvoid*).
```

The *which* argument can be one of the following:

GI_PARAM_STARTED: This is called once per call to `giParameterize` before starting the parameterization.

GI_PARAM_CHANGED: This is called everytime a single patch's parameterization changes and therefore once per patch for the simple parameterizations and once per iteration for the stretch minimization and GIM parameterization. When it is called, the mesh's active patch is set to the currently parameterized patch.

GI_PARAM_FINISHED: This is called once per call to `giParameterize` after all of the patches (or only the active one) have been parameterized.

The callback's return value tells OpenGI if it should continue parameterizing. This way you can abort the parameterization by returning `GI_FALSE`. When aborting, the current (aborted) patch's parameter coordinates are undefined. If parameterizing the whole mesh, all patches parameterized before keep their new parameterization and all patches still to parameterize keep their previous parameterization, if any.

In these callback routines you can then just render the current cut or the current mesh using the parameter coordinates as texture coordinates to visualize the parameterization, or fetch the current stretch value, or visualize the per-vertex stretch, or anything else. But this can only be static, as you have to proceed with the parameterization. A more advanced technique used by my GUI geometry image creation program is to use multiple threads. When starting parameterization, `giParameterize` is called in a new thread and the callbacks just tell the main thread, which has control over OpenGL, to rebuild the mesh's display list. The main thread then draws the mesh, using `giGLDrawMesh` with the parameter coordinates as texture coordinates for use with a checkerboard texture, into a display list. After this display list has been created, the parameterization thread may proceed and the callback function looks if the user has tried to cancel the parameterization since the last callback and proceeds or aborts the parameterization. This way the user can always interactively examine the current parameterization iteration while the next one is being computed. The display list is necessary because OpenGI is everything else than thread save. you should not draw a mesh, that is currently changed by another function.

6. Images and Sampling

6.1. Image Management

Image objects are used to represent the attribute images into which a mesh's attributes are sampled. They generally behave like mesh objects. They are represented in client space by IDs, which are created by calling

```
GIuint giGenImage()
void   giGenImages(GIsizei n, GIuint *images)
```

and deleted by calling

```
void giDeleteImage (GIuint image)
void giDeleteImages(GIsizei n, const GIuint *images)
```

Like meshes an image is bound as the active one by calling

```
void giBindImage(GIuint image)
```

and the currently bound image can be retrieved by calling `giGetIntegerv` with `GI_IMAGE_BINDING`.

Images do not contain any data, instead they are just pointers to the actual image data in user memory. This way unnecessary copying operations can be avoided. You just tell OpenGI what client data is used for the active image and this data is written to when sampling into this image. You can set the data for an image by calling one of these functions:

```
void giImageExternalData (GIsizei width, GIsizei height,
                          GIsizei components,
                          GIenum type, GIvoid *data)
void giImageGLTextureData(GIsizei width, GIsizei height,
                          GIsizei components,
                          GIenum type, GIuint texture)
void giImageGLBufferData (GIsizei width, GIsizei height,
                          GIsizei components,
                          GIenum type, GIuint buffer)
```

Width, *height* and *components* specify the image's width, height (≥ 2) and number of components (in $[1, 4]$) respectively. The *type* argument gives the type of the individual image components, which can be one of the constants from table 6.1. The constants have the same values as the corresponding OpenGL constants, making communication easier. The first function tells OpenGI that the image's data is located in the user-controlled memory pointed to by *data*. The user is responsible for allocating and destroying this data. By the second function you tell OpenGI that the data for the image is located in

constant	description
GI_UNSIGNED_BYTE	8-bit unsigned bytes
GI_HALF_FLOAT	16-bit floating point numbers (1.5.11)
GI_FLOAT	32-bit floating point numbers (1.8.24)

Table 6.1.: Image types

the user-controlled OpenGL texture object specified by *texture*. Again it is the user's responsibility to create and delete this texture, OpenGI only works on it. Finally by the third function the image's data can lie in a user-controlled OpenGL buffer object. It is important to know that these three functions exclude each other, meaning an image's data is **either** located in user RAM **or** in an OpenGL texture **or** in an OpenGL buffer, but not in two or three locations at the same time as this would create data synchronization problems (but you can still manage multiple copies and synchronize them yourself). So a call to `giImage...Data` overwrites a previous call to any `giImage...Data` function.

By this data redirection method the image can easily be suited to your needs without inefficient data copies. For example when working with the image on the CPU you let its data reside in RAM. When using it as textures (e.g. normal mapping) or working with it on the GPU its data can reside in a texture object. Or when you want to render it as vertex array its data can reside in a buffer object. And all this can be achieved without copying the data yourself and with the sampling suited to the particular storage mode. For example when using OpenGL accelerated sampling with framebuffer objects supported and you want to use the image as texture there is no need for the image to ever leave the graphics card.

Since version 2.1 you can specify a sub-rectangle of the whole image by calling

```
void giSubImage(GIuint x, GIuint y, GIsizei width, GIsizei height)
```

with (x,y) being the offset of a $width \times height$ sub-image of the currently bound image. After this call every algorithm working on the image, like sampling or geometry image rendering will only use this sub-image. This way you can for example pack geometry images of multiple patches into a single large image. The default sub-image is the whole image, that can be selected by calling the function with all zeros. Calling any of `giImage...Data` will reset the selected sub-image to this default. When trying to select a sub-image that does not fit into the overall image bounds or that is smaller than 2×2, a `GI_INVALID_VALUE` error is generated.

Nearly every property of an image can be retrieved by calling

```
void giGetImageiv(GIenum pname, GIint *param)
```

specifying the property to query and the address to store its value at. Table 6.2 lists the possible values for *pname*. Note that querying `GI_GL_IMAGE_TEXTURE` or `GI_GL_IMAGE_BUFFER` returns 0 for an image stored in RAM data. The storage mode of the image is either `GI_EXTERNAL_DATA`, `GI_GL_TEXTURE_DATA`, `GI_GL_BUFFER_DATA` or `GI_NO_IMAGE_DATA` for a new image, not yet having `giImage...Data` called for. The image's

constant	description
GI_IMAGE_WIDTH	width of the image
GI_IMAGE_HEIGHT	height of the image
GI_IMAGE_COMPONENTS	number of components per pixel
GI_IMAGE_TYPE	type of image components
GI_GL_IMAGE_TEXTURE	OpenGL texture object if any
GI_GL_IMAGE_BUFFER	OpenGL buffer object if any
GI_IMAGE_STORAGE	storage mode
GI_SUBIMAGE_X	x-offset of selected sub-image
GI_SUBIMAGE_Y	y-offset of selected sub-image
GI_SUBIMAGE_WIDTH	width of selected sub-image
GI_SUBIMAGE_HEIGHT	height of selected sub-image
GI_SUBIMAGE	sub-image region as 4 integers

Table 6.2.: Image properties

RAM data address, which is NULL if the image is not stored in RAM, can be queried by means of giGetPointerv with the argument GI_IMAGE_DATA.

6.2. Sampling

To sample into an image, it has to be bound to a specific attribute channel by calling

```
void giAttribImage(GIuint attrib, GIuint image).
```

This uses *image* as target when sampling the attribute specified by *attrib*.

Finally, after specifying and binding the necessary images, the currently bound mesh's attributes, or rather those of the mesh's active patch, can be sampled into the corresponding attribute images by calling

```
void giSample()
```

This function samples all vertex attributes of the currently active patch that have a valid image bound to their channel. Bind image 0 to disable an attribute. If there is no active patch selected in the current mesh, a GI_INVALID_OPERATION error is thrown and if the patch has not been parameterized or the parameterization boundary is not the unit square — which can only happen if you specified the parameter coordinates yourself or took them from an attribute — a GI_INVALID_PARAMETERIZATION error will be thrown.

If any of the bound attribute images has no data associated to it, a GI_INVALID_OPERATION error will be generated and this attribute will not be sampled. However, the operation will proceed sampling the other attributes. After sampling is finished the number and combination of successfully sampled attributes can be queried by calling giGetIntegerv with either GI_SAMPLED_ATTRIB_COUNT or GI_SAMPLED_ATTRIBS, which has the ith bit set if the ith attribute has been sampled successfully.

The actual sampling can either be done in software using a software rasterizer or hardware accelerated by using OpenGL for rasterization. Both samplers have their advantages and disadvantages. Most times you will use the software sampler because of its higher precision but in certain cases you may trade off precision for the higher speed of hardware accelerated sampling. The sampler to use can be specified by calling

```
void giSamplerParameteri(GIenum pname, GIint param)
```

with *pname* being `GI_SAMPLER` and *param* being one of the following values:

GI_SAMPLER_SOFTWARE: The software sampler is the default and also the recommended sampler as it is the most exact sampling method. It may be slower than the hardware sampler, but it profits from mutlithreading, if supported and enabled. Since version 2.0 the software sampler supports resampling of textures and is thus equal to the OpenGL sampler regarding the supported features.

GI_SAMPLER_OPENGL: This sampler uses OpenGL for rasterizing triangles and can therefore be faster, depending on your graphics hardware, especially when the image data is stored in VRAM. The sampler works with any hardware supporting at least OpenGL 1.1, but can take advantage of newer hardware capabilities, as GLSL, FBOs, etc. The use of GLSL and FBOs can be enabled or disabled by calling

```
void giSamplerParameterb(GIenum pname, GIboolean param)
```

with `GI_SAMPLER_USE_SHADER` and `GI_SAMPLER_USE_RENDER_TO_TEXTURE`. These are enabled by default, but are only hints, as these features also have to be supported by the OpenGL context. If not using FBOs, you can only sample to images of type `GI_UNSIGNED_BYTE`. Also think of the OpenGL requirements that depend on the image configuration, as e.g. floating point textures, half-precision floating point values, or two component textures. It is not reccommended to use this sampler as it does not provide the precision needed in many cases (see A for the cases when hardware sampling may suffice).

Note that it is no problem to call `giSample` more than once or even sample one attribute with the software sampler and another one with hardware sampling but for performance reasons it is recommended to sample all attributes in one call and for precision reasons it is recommended to sample all attributes with the same sampler as the results of different sampling modes may vary slightly and different attributes may not fit onto each other exactly, although these artifacts may not be noticable.

When sampling an attribute the per-vertex values are first expanded to 4 components by $(0, 0, 0, 1)$, if smaller. Then these values are transformed by a user specified 4×4-matrix that can be set by calling

```
void giAttribSamplerParameterfv(GIuint attrib, GIenum pname,
                                const GIfloat *params)
```

with *pname* being `GI_SAMPLING_TRANSFORM` and *params* being the matrix to use in **column-major** format, defaulting to the identity matrix. It can be used to fit the mesh into the 0-1-cube when sampling vertex postions into a ubyte-image, for example.

After expansion and transformation the per-vertex attribute values are interpolated across the triangles and rasterized into the parameter domain. The specific sampling mode to use for an attribute can be specified with

```
void giAttribSamplerParameteri(GIuint attrib, GIenum pname,
                               GIint param)
```

and a *pname* of `GI_SAMPLING_MODE`. The following modes are supported:

GI_SAMPLE_DEFAULT: As the name suggests this is the default sampling mode. The attribute is just interpolated across the triangle.

GI_SAMPLE_NORMALIZED: The attribute is renormalized after interpolation. If the type of the associated image is `GI_UNSIGNED_BYTE`, the values are also linearly transformed from $[-1, 1]$ to $[0, 1]$ (or exactly $[0, 255]$). The attribute is always normalized as a 3D vector and the fourth component set to 1.

GI_SAMPLE_TEXTURED: The attribute is used as a texture coordinate to resample a texture. The actual OpenGL texture object to resample is specified by `giAttribSamplerParameteri` with *pname* being `GI_GL_SAMPLE_TEXTURE` and its dimension with `GI_TEXTURE_DIMENSION`. The texture dimension has to lie between 1 and 4, with 4 meaning a cube map texture.

7. OpenGL Utility Functions

7.1. General OpenGL Issues

OpenGI is designed for easy use with OpenGL. All of the render functions explained in this chapter assume an appropriately initialized OpenGL context to be active. As OpenGI may create OpenGL context local data as e.g. shader programs or VBOs, the same GL context should be active when calling functions using OpenGL during the whole lifetime of the OpenGI context. Before another OpenGL context is made current or when the GL context is destroyed before the GI context you should call

```
void giGLCleanUp()
```

otherwise the results are undefined next time calling an OpenGI function that makes use of OpenGL. When the GI context is destroyed before the GL context it is not necessary to call this function. To be sure this list contains all functions that use OpenGL:

- `giSample` (the software sampler may also use some OpenGL)
- `giGLDraw[Mesh/Cut/GIM]`
- `giGLCleanUp`

The non-rendering methods are designed to preserve every OpenGL state they modify. Furthermore they also mostly set every state they need explicitly. So these functions may be called at every point in your application, even in the middle of your rendering routine, although this is quite unlikely.

7.2. Rendering Meshes

The currently bound mesh (or only its active patch) can be rendered with

```
void giGLDrawMesh()
```

which renders the mesh's faces as triangles immediately (using `glBegin/glEnd`). This routine is mainly intended for debugging purposes, e.g. when you want feedback during parameterization, as it is quite inefficient to convert the whole mesh into a vertex/index array representation every time the parameterization changes. Because of its intensive use of conditional blocks it could be a good idea to put this function into an OpenGL display list when you render the mesh more than once, e.g. every frame.

Similarly to the mesh itself you can also render its current cut by calling

```
void giGLDrawCut()
```

constant	description
GI_NONE	do not render attribute
GI_GL_VERTEX	render as glVertex
GI_GL_NORMAL	render as glNormal
GI_GL_COLOR	render as glColor
GI_GL_SECONDARY_COLOR	render as glSecondaryColor
GI_GL_FOG_COORD	render as glFogCoord
GI_GL_EVAL_COORD	render as glEvalCoord
GI_GL_TEXTURE_COORD	render as gl[Multi]TexCoord
GI_GL_VERTEX_ATTRIB	render as glVertexAttrib

Table 7.1.: OpenGL render semantics for attribute channels

which draws the mesh's cut immediately as a line strip (actually a line loop). These rendering functions do not modify any OpenGL state and require the usual state assumed when calling `glBegin/glEnd`. Both functions only draw the active patch's triangles and cut edges, unless the whole mesh is selected.

The mapping from OpenGI attribute channels to OpenGL vertex attributes for rendering is established by calling

```
void giGLAttribRenderParameteri(GIuint attrib, GIenum pname,
                                GIint param)
```

for a specific attribute, with *pname* being `GI_GL_RENDER_SEMANTIC` and one of the constants from table 7.1 as *param*. Of course the used attributes also have to be supported by the OpenGL context. When using the `GI_GL_TEXTURE_COORD` or `GI_GL_VERTEX_ATTRIB` semantics, the texture unit or attrib location to use can be set by means of `giGLAttribRenderParameteri` with a *pname* of `GI_GL_RENDER_CHANNEL`. If multitexturing is not supported, only the attribute that is bound to texture unit 0 is rendered. When more than one OpenGI attribute channel is bound to a single OpenGL vertex attribute, only one will be rendered. The same holds for `glVertexAttrib(0, ...)` and `glVertex`, which are actually aliases.

This way the mesh can be rendered fully multitextured, the current parameterization can be visualized, an attribute can be transmitted as vertex attrib for use in a shader, or the stretch of the parameterization can be visualized using a 1D texture as color map. When doing so, note that the param stretch values are highly mesh and parameterization dependent and usually larger than 1, so it could be a good idea to transform them to the [0,1]-range by using the texture matrix and its minimum and maximum values.

7.3. Rendering Geometry Images

OpenGI can also render the currently bound attribute images as a colored, normal mapped and fully multitextured triangular mesh. This rendering currently only supports the first light source and only directional or point light without attenuation. When

normals are rendered, colors are ignored. The textures are applied by modulation. Two sided materials and lighting are supported.

The geometry image is rendered by calling

```
void giGLDrawGIM()
```

This function draws the attributes bound to the `GI_GL_VERTEX`, `GI_GL_NORMAL` and `GI_GL_COLOR` semantics and the first four `GI_GL_TEXTURE_COORD` units, using the images bound to these attribute channels, if any. It requires the active OpenGL context to support GLSL and the rendering shaders to be compiled successfully, which they should normally do, given GLSL support, otherwise a `GI_UNSUPPORTED_OPERATION` error will occur. There are some more OpenGL requirements depending on the image configuration which are quite self-evident, e.g. floating point textures, NPOT textures, half-precision vertex data, etc. If any error occurs for the attribute bound to the `GI_GL_VERTEX` render semantic, the rendering will fail. For errors caused by other attributes the rendering will continue just ignoring the attribute image that caused the error.

The rendering can be controlled with the functions

```
void giGLRenderParameterb(GIenum pname, GIboolean param)
void giGLRenderParameteri(GIenum pname, GIint param)
```

By calling `giGLRenderParameteri` with either `GI_RENDER_RESOLUTION_U` or with `GI_RENDER_RESOLUTION_V` you can specify a lower resolution for the geometry image than its size. This way the geometry data is subsampled and discrete Levels of Detail can be implemented. But some restrictions have to be taken care of: when the geometry image size is $n \times n$ and the render resolution $m \times m$ then $n - 1$ has to be an integer multiple of $m - 1$, e.g. 10 and 4 or 513 and 257. Usually you will use image sizes and resolutions of the form $(2^m + 1) \times (2^m + 1)$, which suffice this restriction (and have some other advantages). By setting a resolution to 0, which is the default for both dimensions, the geometry will always be sampled at the highest resolution (that of the image) in the corresponding dimension. When rendering multiple resolutions of a geometry image, see A for the steps to take for getting a watertight reconstruction.

By default the domains of the texture coordinates for accessing the images are indented by half a texel resulting in the images being sampled at texel centers at least at the border. This is recommended for images created by OpenGI as `giSample` maps the parameterization border to exact texels and OpenGL returns the exact texel value when filtering at the texel center. Sometimes you may want to override this default behavior, for example when you have an external texture that needs texture coordinates in the range [0,1]. Just call `giGLAttribRenderParameteri` with a *pname* of `GI_TEXTURE_COORD_DOMAIN` for the required attributes, setting it to `GI_UNIT_SQUARE` instead of `GI_HALF_TEXEL_INDENT`.

The following features can be enabled with `giGLRenderParameterb`:

GI_USE_VERTEX_TEXTURE: This will enable/disable the use of vertex texture fetches (VTF), which might be faster when the position data is stored in a texture object. But you should evaluate it yourself for your specific hardware. It is turned off by default.

GI_USE_GEOMETRY_SHADER: This will enable/disable the use of the geometry shader (GS). When using geometry shaders the quads of the regular quad mesh resulting from a geometry image are splitted along their shorter diagonal to produce the two triangles. This should look better for highly stretched quads, compared to the normally used triangle strips which always split along the same diagonal. Figure 7.1 shows the difference, especially at sharper creases, as the nose or the chin. Note that this rendered triangular mesh is not regular anymore at the triangle level, but it is of course still regular in the sense of the quads defining the triangle pairs. Also, when using GS and VTF together there is no index data needed anymore, which might reduce the memory consumption and increase the rendering throughput.

Note that these flags are only hints to the renderer, as these techniques have to be supported by the OpenGL context, of course.

If VBOs are supported the rendering performance might profit from caching texture coordinate and index data, which depend only on the geometry image size and can therefore be used more than once. The size of this render cache, consisting of a texCoord and an index cache, can be set by calling `giGLRenderParameteri` with `GI_RENDER_CACHE_SIZE`. This caching can speed up rendering significantly as texCoord and index data can be stored in VRAM and reused for many render calls. But the size of the cache should not be too small as this could result in reallocating data every frame when rendering too much differently sized geometry images per frame. As an orientation: one texCoord cache entry for one image size and one index cache entry for one combination of image size and rendering resolution. So for example when rendering only one geometry image (or only images with the same size) every frame, a cache size of 1 should be sufficient. But when rendering the geometry data in 3 different resolutions **every frame** the cache size should be at least 3 because of the index data. But when switching the resolution for **all** geometry images only a few times the cache does not need to be so large. Note that for the render cache only the size of the geometry image matters and not that of other attribute images. The default cache size is 8 and setting it to 0 disables render caching.

As textures are considered highly object dependent and this is a render function, it does not save the current texture bindings before changing them, but as shaders are used, it does not enable any texture units. When rendering n different attributes (including positions), the texture bindings (for the `GL_TEXTURE_2D`-target) of the first n-1 texture units are changed, the first n if vertex texturing is used.

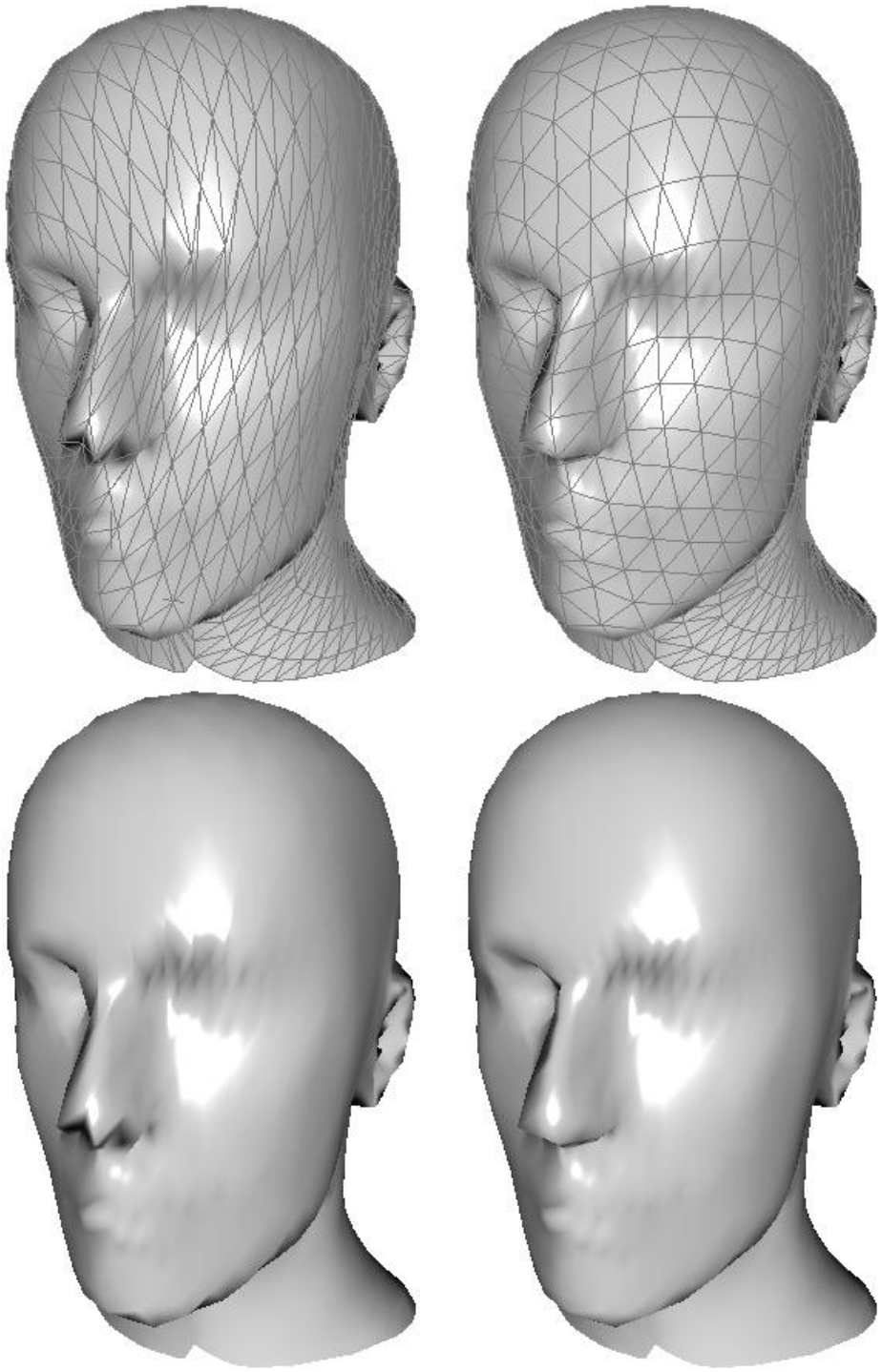

Figure 7.1.: The left images show regular triangle strip rendering and the right ones are rendered using the geometry shader, splitting each quad along its shorter diagonal. This reduces artifacts at sharper creases, like the nose or chin.

A. Programming Tips

When using OpenGI there are some best practices regarding the performance and, more important, the precision of the operations, that might not be known to someone not too deep into the field of Geometry Images. These tips might have already been stated elsewhere but are drawn together here for reference purposes. Take these advices from someone who really knows the internals of OpenGI.

A.1. Precision Issues

Whenever you use `GI_INITIAL_GIM` for cutting or `GI_GIM` for parameterizing, it can (or must) happen, that the cut cuts into the mesh, meaning there are cut edges not lying on the mesh boundary. These edges are mapped to two different border segments of the geometry image, as the mesh is ripped open along the cut. For the rendered geometry image not to show any gaps along this cut, both border segments have to contain the exact same values. To achieve this, some vertices have to be mapped to exact texels during parameterization, namely the vertices incident to only one or more than two cut edges. For this to happen, the resolution of the parameter domain has to be set appropriately.

- Before parameterizing set `GI_PARAM_RESOLUTION` to the size of the geometry (position) image when sampling.

You might want to sample or render the geometry image in different resolutions to do LOD or whatever and do not want to change the parameterization. Then, what should be the resolution of the parameter domain?

- Use resolutions where each grid contains all the texels of the next lower resolution and border texels are also border texels in the next higher resolution. A good choice are resolutions of the form $(2^m + 1) \times (2^m + 1)$. These should also be used for the other images, although precision deviations in these might not be noticeable. (This does not mean the other images should have the same size as the geometry image, they only should suffice the power-of-two-plus-1 condition).

- Set `GI_PARAM_RESOLUTION` to the lowest resolution you want to sample or render the geometry image with.

After the needed vertices have been parameterized to exact texels, they also have to be sampled exactly. At the moment this can only be achieved by the software sampler.

- Before sampling set `GI_SAMPLER` to `GI_SAMPLER_SOFTWARE` or leave it as is, as this is the default sampling mode.

Of course these restrictions also apply if you have used `GI_EXACT_MAPPING_SUBSET` during mesh creation or if the mesh has multiple patches. If you have a genus-0 mesh with only one boundary and there are no other vertices that you want to be mapped exactly, then you may want to use the hardware sampler, because of its higher performance, especially for images stored in VRAM.

A.2. Performance Tips

Here are some performance rules you should have in mind when sampling the mesh. These are anything else than strict and may not have too much impact on the overall performance, but when they can be applied, they should. Do not underestimate the cost of copying a large image between the graphics card and the system memory.

- Try to sample all needed attributes in one call to `giSample`.
- The software sampler is faster if more images have the same size.
- Set the storage mode of the image to fit its intended use, e.g. buffers to use for vertex array rendering or pixel operations, textures to use for texturing or GPU processing and system memory to use for CPU processing.
- If sampling in software mode, and the image is needed as a texture **and** as a RAM copy, store it in RAM and copy it into the texture after sampling, as the software sampler works in system memory anyway.
- If sampling hardware-accelerated, and the image is needed as a texture **and** as a RAM copy, store it in the texture and read it into RAM after sampling, as the hardware sampler works on the graphics card anyway.

When you want to render the images as triangular meshes using `giGLDrawGIM` you should have in mind, that

- Rendering with `giGLDrawGIM` works fastest if the geometry image is stored in a buffer object and the other images in texture objects.

B. Usage Example

At this point some sourcecode from the *gim* example is presented, demonstrating the use of OpenGI in a simple Geometry Image creator/viewer. I will only show the OpenGI related parts in the sequence they are executed. The whole project can be found together with the source distribution of OpenGI.

First we need some global variables storing our OpenGI and OpenGL objects.

```
unsigned int uiMesh;              // mesh object
unsigned int uiGIM[3];            // image objects
unsigned int uiPattern;           // checkerboard texture
void GICALLBACK errorCB(unsigned int error, void *data);
```

In the `main` function we first create an OpenGI context and set an error callback function. This error callback just prints the error string to the standard output, which would OpenGI do by default, when compiled with verbosity enabled.

```
int main(int argc, char *argv[]) {
...
GIcontext pContext = giCreateContext();
giMakeCurrent(pContext);
giErrorCallback(errorCB, NULL);
```

Next, our mesh has to be created from a set of arrays, which are actually read from a file, but this is unimportant here. First we will tell OpenGI where it can find some special attributes. We will use attribute 0 for vertex positions (default anyway) and 2 for the parameter coordinates (and 1 for the normals, but that does not care OpenGI). Then we set and enable the arrays for our attributes. We only want to set positions and normals, params are created during parameterization.

```
giBindAttrib(GI_POSITION_ATTRIB, 0);
giBindAttrib(GI_PARAM_ATTRIB, 2);
giAttribPointer(0, 3, GI_FALSE, 0, pVertices);
giAttribPointer(1, 3, GI_TRUE, 0, pNormals);
giEnableAttribArray(0);
giEnableAttribArray(1);
```

Now we can create our mesh from these arrays with an appropriate index array.

```
uiMesh = giGenMesh();
giBindMesh(uiMesh);
giIndexedMesh(0, iNumVertices-1, iNumIndices, pIndices);
```

First we have to cut the mesh into a topological disc.

```
giCutterParameteri(GI_CUTTER, GI_INITIAL_GIM);
giCut();
```

When this is done, we parameterize the mesh. The parameters set here (except for the resolution) are actually the default ones but are set here for the sake of completeness.

```
giParameterizerParameteri(GI_PARAMETERIZER, GI_STRETCH_MINIMIZING);
giParameterizerParameteri(GI_INITIAL_PARAMETERIZATION, GI_MEAN_VALUE);
giParameterizerParameterf(GI_STRETCH_WEIGHT, 1.0f);
giParameterizerParameteri(GI_PARAM_RESOLUTION, res);
giParameterize();
```

When the mesh is parameterized we can sample its attributes into images. But first we have to create the storage needed for the images. All images will be stored in textures, so we need to create three textures, one for the geometry and the normal data each and a third one, which is not sampled into, containing a checkerboard pattern for visualizing the parameterization.

```
glGenTextures(2, uiTex);
glBindTexture(GL_TEXTURE_2D, uiTex[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, res, res, 0,
   GL_RGBA, GL_FLOAT, NULL);

int iNRes = (res&1) ? (res<<1)-1 : (res<<1);
glBindTexture(GL_TEXTURE_2D, uiTex[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, iNRes, iNRes, 0,
   GL_RGB, GL_UNSIGNED_BYTE, NULL);

glGenTextures(1, &uiPattern);
glBindTexture(GL_TEXTURE_2D, uiPattern);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 256, 256, 0,
   GL_LUMINANCE, GL_UNSIGNED_BYTE, checkerboard);
```

Now we can create the images for the three attributes and tell them that they are stored in OpenGL textures.

```
giGenImages(3, gim);
giBindImage(gim[0]);
giImageGLTextureData(res, res, 4, GI_FLOAT, uiTex[0]);
giBindImage(gim[1]);
giImageGLTextureData(iNRes, iNRes, 3, GI_UNSIGNED_BYTE, uiTex[1]);
giBindImage(gim[2]);
giImageGLTextureData(256, 256, 1, GI_UNSIGNED_BYTE, uiPattern);
```

We bind the images to the attribute we want to get sampled into them. We only want to sample the positions and normals (0 and 1) as image 2 already contains useful data. Then we tell the sampler to renormalize attribute 1 after interpolation, as it contains the normals.

```
giAttribImage(0, gim[0]);
giAttribImage(1, gim[1]);
giAttribImage(2, 0);
giSamplerParameteri(GI_SAMPLER, GI_SAMPLER_SOFTWARE);
giAttribSamplerParameteri(0, GI_SAMPLING_MODE, GI_SAMPLE_DEFAULT);
giAttribSamplerParameteri(1, GI_SAMPLING_MODE, GI_SAMPLE_NORMALIZED);
giSample();
```

Now we can look at our `display` function, that is executed every frame. With *iDrawMesh* in {0,1} we can decide if we draw the mesh or the geometry image and with *iParam* in {0,1} if we want to visualize the parameterization. We will use attribute 0 as positions, 1 as normals and 2 as texture coordinates when rendering the mesh and its image as texture when rendering the geometry image (but only if we want to visualize the parameter coordinates). Note that the mesh and image bindings are not neccessary here, as they should still be bound. Also in practice it would be a good idea to build the mesh rendering into a display list.

```
void display() {
  ...
  giGLAttribRenderParameteri(0, GI_GL_RENDER_SEMANTIC, GI_GL_VERTEX);
  giGLAttribRenderParameteri(1, GI_GL_RENDER_SEMANTIC, GI_GL_NORMAL);
  giGLAttribRenderParameteri(2, GI_GL_RENDER_SEMANTIC,
     iParam ? GI_GL_TEXTURE_COORD : GI_NONE);
  giGLAttribRenderParameteri(2, GI_GL_RENDER_CHANNEL, 0);
  if(iDrawMesh) {
     if(iParam) {
        glBindTexture(GL_TEXTURE_2D, uiPattern);
        glEnable(GL_TEXTURE_2D);
     }
     giBindMesh(uiMesh);
     giGLDrawMesh();
  } else {
     giAttribImage(0, uiGIM[0]);
     giAttribImage(1, uiGIM[1]);
     if(iParam) {
        giAttribImage(2, uiGIM[2]);
        giGLAttribRenderParameteri(2,
           GI_TEXTURE_COORD_DOMAIN, GI_UNIT_SQUARE);
     }
     giGLDrawGIM();
  }
  ...
}
```

C. Porting from Version 1.X

Since some parts of the API have changed significantly in version 2.0, this chapter summarizes the major changes and presents adaption strategies for older applications. These should be ported, as not only the API has changed since version 1.2 but also many internals have been improved. For some code samples it is also a good idea to compare the usage examples of both versions with each other.

The most significant change, effecting nearly all aspects of OpenGI, is the new generic attribute system, replacing the old fixed-semantic attributes. So the setting of arrays has become simpler, with only `giAttribPointer` instead of a different function for every attribute. But on the other hand you have to manage the attribute semantics yourself, the OpenGI semantics with `giBindAttrib` as well as your own, like normals or texture coordinates. This means, you also have to set up the correct render semantics before calling `giDraw[Mesh/Cut/GIM]` and the correct sampling mode. Moreover there is now per-attribute state and corresponding getters and setters. The mesh retrieval API has been simplified, too, but now you have to allocate the target arrays yourself.

A feature that has been removed are the mesh modification functions, like `giFlipMesh` or `giTransformMesh`, as these were more of a debugging purpose and do not make much sense with the generic attributes. To make these modifications to the mesh data you will have to retrieve its data and create the mesh again, or you make these modifications before creating the mesh. If you just want to transform the mesh before sampling, you can use the new per-attribute sampling matrix (but remember to transform the normals appropriately when transforming the vertex positions). This matrix is also needed when fitting the geometry to $[0, 1]$ or normalizing the param stretch for sampling into a ubyte-image, which is not done automatically anymore.

Fortunately you do not have to bother with the new patch functionality if the mesh consists of one patch only. Many cutting algorithms have been removed. As `GI_SINGLE_BOUNDARY` gives the same results as `GI_INITIAL_GIM` on the allowed meshes, it is not needed and a relict of old times, when the more advaced cutter was not implemented yet. As the cut is now decoupled from the parameterization, `GI_REPARAMETERIZE_ONLY` is not needed anymore and as parameterizing the cut makes no real performance difference, `GI_USE_EXISTING` is dropped, too.

There is now only one image bound for modification by `giBindImage` and not one per attribute. The attribute images are bound separately by `giAttribImage` and this binding is used to determine the attributes selected for sampling and rendering, instead of a bitfield. Another good news is, you can now use the software sampler for texture resampling and do not have to switch to hardware mode for this. But on the other hand the hardware sampler has become much more accurate (but still cannot beat software mode).

D. State Variables

Tables D.1 to D.3 list all state values that can be retrieved by calling `giGetBooleanv`, `giGetIntegerv` or `giGetFloatv`. Additionally tables D.4 to D.6 list the per-attribute global state, that can be retrieved with `GetAttrib[bif]v`. For the sake of compactness the leading `GI_` has been left off in all constants.

constant	default	description
MULTITHREADING	TRUE	multithreading enabled
EXACT_MAPPING_SUBSET	FALSE	exact mapping subset enabled
PARAM_CORNER_SUBSET	FALSE	parameterization corner subset enabled
EXACT_MAPPING_SUBSET_SORTED	FALSE	sorted flag for exact mapping subset
PARAM_CORNER_SUBSET_SORTED	FALSE	sorted flag for param corner subset
SAMPLER_USE_SHADER	TRUE	use GLSL in hardware sampling
SAMPLER_USE_RENDER_TO_-TEXTURE	TRUE	use FBOs in hardware sampling
GL_USE_VERTEX_TEXTURE	FALSE	use vertex texture in GIM rendering
GL_USE_GEOMETRY_SHADER	FALSE	use geometry shader in GIM rendering

Table D.1.: Boolean state variables

constant	default	description
AUTHALIC_WEIGHT	1.0	authalic weight for Intrinsic parameterization
CONFORMAL_WEIGHT	1.0	conformal weight for Intrinsic parameterization
STRETCH_WEIGHT	1.0	stretch weight for Stretch Minimization
AREA_WEIGHT	1.0	area weight for combined stretch metric

Table D.2.: Floating point state variables

constant	default	description
VERSION	(2,1,X)	OpenGI version as array of size 3
MAX_ATTRIBS	16	available attribute channels
MESH_BINDING	0	currently bound mesh
IMAGE_BINDING	0	currently bound image
EXACT_MAPPING_SUBSET_COUNT	0	number of elements in exact mapping subset
PARAM_CORNER_SUBSET_COUNT	0	number of elements in parameterization corner subset
POSITION_ATTRIB	0	attribute used for vertex positions
PARAM_ATTRIB	0	attribute used for parameter coordinates
PARAM_STRETCH_ATTRIB	0	attribute used for parameterization stretch
CUTTER	INITIAL_GIM	cutting algorithm
SUBDIVISION_ITERATIONS	1	number of iterations for subdivision
PARAMETERIZER	STRETCH_MINIMIZING	parameterization algorithm
INITIAL_PARAMETERIZATION	MEAN_VALUE	starting parameterization for Stretch Minimization
STRETCH_METRIC	RMS_GEOMETRIC_STRETCH	stretch metric for Stretch Minimization
PARAM_RESOLUTION	33	resolution of parameter domain
UNSYMMETRIC_SOLVER	BICGSTAB	solver for unsymmetric systems
PARAM_SOURCE_ATTRIB	0	attribute to source param coords from
SAMPLER	SAMPLER_SOFTWARE	sampling mode
SAMPLED_ATTRIB_COUNT	0	number of sampled attributes
SAMPLED_ATTRIBS	0	bitwise combination of successfully sampled attributes
RENDER_RESOLUTION_U	0	U-resolution for GIM rendering
RENDER_RESOLUTION_V	0	V-resolution for GIM rendering
RENDER_CACHE_SIZE	8	size of cache for GIM rendering

Table D.3.: Integer state variables

constant	default	description
ATTRIB_ARRAY_ENABLED	FALSE	attribute array enabled
ATTRIB_ARRAY_NORMALIZED	FALSE	normalized flag of attribute array

Table D.4.: Boolean per-attribute state variables

constant	default	description
ATTRIB_ARRAY_SIZE	0	size of attribute array
ATTRIB_ARRAY_STRIDE	0	stride value of attribute array
ATTRIB_ARRAY_SEMANTIC	-	semantic attribute is bound to
ATTRIB_IMAGE	0	image bound to attribute channel
SAMPLING_MODE	SAMPLE_-DEFAULT	attribute sampling mode
TEXTURE_DIMENSION	2	dimension of texture to resample
GL_SAMPLE_TEXTURE	0	texture object to resample
GL_RENDER_SEMANTIC	-	OpenGL attribute used for rendering
GL_RENDER_CHANNEL	0	texture unit or vertex attrib location used for rendering
TEXTURE_COORD_DOMAIN	HALF_TEXEL_-INDENT	texture coord domain for GIM rendering

Table D.5.: Integer per-attribute state variables

constant	default	description
SAMPLING_TRANSFORM	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	4×4 transformation matrix to be applied during sampling (column-major)

Table D.6.: Floating point per-attribute state variables

E. GNU Free Documentation License

Version 1.2, November 2002

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgments", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section

"Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time

you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgments" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgments”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgments", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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.
See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under

> the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[CC78] Edwin Catmull and Jim Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978.

[DMA02] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum*, 21(3):209–218, 2002.

[DMK03] Patrick Degener, Jan Meseth, and Reinhard Klein. An adaptable suface parameterization method. In *Proceedings of the 12th International Meshing Roundtable (IMR 2003)*, pages 201–213. Sandia National Laboratories, 2003.

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH 1995*, pages 173–182. ACM Press, 1995.

[Flo97] Michael Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(3):231–250, 1997.

[Flo03] Michael Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[GGH02] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *Proceedings of SIGGRAPH 2002*, pages 355–361. ACM Press, 2002.

[HLS07] Kai Hormann, Bruno Lévy, and Alla Sheffer. Mesh parameterization: Theory and practice. In *ACM SIGGRAPH Course Notes*, 2007.

[SSGH01] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of SIGGRAPH 2001*, pages 409–416. ACM Press, 2001.

[Tut63] W. T. Tutte. How to draw a graph. In *Proceedings of the London Mathematical Society*, pages 743–767, 1963.

[YBS04] Shin Yoshizawa, Alexander Belyaev, and Hans-Peter Seidel. A fast and simple stretch-minimizing mesh parameterization. In *Proceedings of the 2004 International Conference on Shape Modeling and Applications (SMI '04)*, pages 200–208. IEEE Computer Society, 2004.

Function Index

M

N

P

S

V

Enumeration Index

I

M

N

P

R

S

T

U

V