



OpenEye
Scientific Software

Spicoli TK – C++
Release 1.1.1

OpenEye Scientific Software, Inc.

January 11, 2012

CONTENTS

1	Front Matter	1
2	Spicoli Theory	3
2.1	Surfaces	3
2.2	Surface Generation	7
2.3	Surface Manipulation	10
2.4	Surface Storage	11
3	API	13
3.1	OESpicoli Classes	13
3.2	OESpicoli Constants	24
3.3	OESpicoli Functions	25
4	Glossary and Bibliography	37
4.1	Glossary	37
4.2	Bibliography	37
5	Release Notes	39
5.1	SpicoliTK 1.1.1	39
5.2	SpicoliTK 1.1.0	39
5.3	SpicoliTK 1.0.2	39
5.4	SpicoliTK 1.0.1	40
5.5	Indices and tables	40
	Bibliography	41
	Index	43

FRONT MATTER

Copyright 1997-2012 OpenEye Scientific Software, Santa Fe, New Mexico. All rights reserved.

All rights reserved. This material contains proprietary information of OpenEye Scientific Software. Use of copyright notice is precautionary only and does not imply publication or disclosure.

The information supplied in this document is believed to be true but no liability is assumed for its use or the infringement of the rights of others resulting from its use. Information in this document is subject to change without notice and does not represent a commitment on the part of OpenEye Scientific Software.

This package is sold/licensed/distributed subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated without OpenEye Scientific Software's prior consent, in any form of packaging or cover other than that in which it was produced. No part of this manual or accompanying documentation, may be reproduced, stored in a retrieval system on optical or magnetic disk, tape, CD, DVD or other medium, or transmitted in any form or by any means, electronic, mechanical, photocopying recording or otherwise for any purpose other than for the purchaser's personal use without a legal agreement or other written permission granted by OpenEye.

This product should not be used in the planning, construction, maintenance, operation or use of any nuclear facility nor the flight, navigation or communication of aircraft or ground support equipment. OpenEye Scientific Software, shall not be liable, in whole or in part, for any claims arising from such use, including death, bankruptcy or outbreak of war.

Windows is a registered trademark of Microsoft Corporation. Apple, OS X, and Macintosh are registered trademarks of Apple Computer, Inc. AIX and IBM are registered trademarks of International Business Machines Corporation. UNIX is a registered trademark of the Open Group. RedHat is a registered trademark of RedHat, Inc. Linux is a registered trademark of Linus Torvalds. SPARC is a registered trademark of SPARC International Inc.

SYBYL is a registered trademark of TRIPOS, Inc. MDL is a registered trademark and ISIS is a trademark of Accelrys, Inc. SMILES, SMARTS, and SMIRKS may be trademarks of Daylight Chemical Information Systems. Macromodel is a trademark of Schrodinger, Inc.

Python is a trademark of the Python Software Foundation. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

Other products and software packages referenced in this document are trademarks and registered trademarks of their respective vendors or manufacturers.

SPICOLI THEORY

The *GRASP* manual states as central to its philosophy “...the use of surfaces both for displaying properties and as objects in their own right”. Spicoli is a logical descendant of this statement. For surfaces to exist as “objects in their own right” they require a support system: creation, manipulation, and storage. Spicoli provides all the necessary functionality to perform these tasks, along with the ability to compare surfaces. Accordingly, Spicoli provides all the functions necessary to analyze surfaces.

2.1 Surfaces

In Spicoli, the surface is presented as a first-class object, i.e., an object in its own right, named `OESurface`. This presentation requires a minimalist definition similar to the way *OEChem* represents molecules as a collection of atoms and bonds. Basically, both `OESurfaces` and `OEMolBases` represent a graph in computer memory.

An `OESurface` provides two ways of retrieving data. The first is a set of methods that will return a copy of the entire underlying data array. The second are a set of methods with the `Element` suffix that allow random access directly into the array of data. Basic usage of both will be shown for vertices and triangles.

2.1.1 Vertices

The simplest datum in a surface is the vertex: a set of (x , y , z) coordinates. Vertices are stored internally as an array of large enough to hold `GetNumVertices() * 3` floats. *Figure 1* shows how equivalent dimensions are stored at every third place in the array:

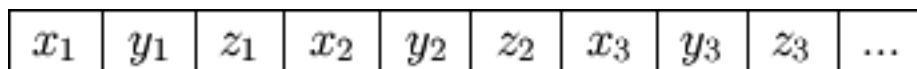


Figure 2.1: **Figure 1**
Arrangement of coordinates in the vertices array

A copy of this array can be obtained using the code fragment in [Listing 1](#), where `surf` is a `OESurface` object.

Listing 1: Example of retrieving the coordinates of all the vertices

```
float *coords = new float[surf.GetNumVertices()*3];  
surf.GetVertices(coords);
```

For direct access to the vertex array use the `Element` version of `GetVertices`, `OESurface::GetVerticesElement`. Listing 2 is a code fragment that shows how to iterate over all the vertex coordinates of a surface.

Listing 2: Example of iterating over all vertices

```
float x, y, z;
for(unsigned int i=0; i < surf.GetNumVertices(); ++i)
{
    x = surf.GetVerticesElement(i * 3);
    y = surf.GetVerticesElement(i * 3 + 1);
    z = surf.GetVerticesElement(i * 3 + 2);
}
```

2.1.2 Triangles

A triangle in Spicoli is a set of three vertices. However, it is important that the order of these vertices be locally consistent. To maintain this consistency vertices of the triangles are defined in a clockwise fashion.

Consider the two triangles in *Figure 2* defined by the vertices *A*, *B*, *C*, *D*. The triangle on the left is defined by the set (*A*, *B*, *C*). The triangle on the right is defined by the set (*D*, *B*, *A*). Also notice how the edge *AB* is defined in both triangles but in opposite order. This reversal of order in the two definitions is a direct consequence of the clockwise ordering rule.

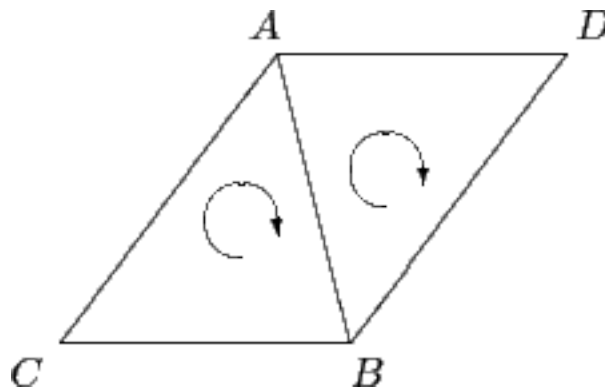


Figure 2.2: **Figure 2**
Two adjacent triangles in a surface

All the methods of construction described in this document obey this rule. A surface that does not follow this rule can still be operated on by Spicoli, but it is not considered a canonical surface and some results may be incorrect or ambiguous.

Spicoli stores triangle vertices as an integer array large enough to hold `GetNumTriangles() * 3` unsigned integers. Each integer is an index into the vertex array explained in *Vertices*. Listing 3 is a code fragment that will retrieve a copy of the entire triangles array.

Listing 3: Example of retrieving the indices of all the triangles

```
unsigned int *triangles = new unsigned int[surf.GetNumTriangles()*3];
surf.GetTriangles(triangles);
```

For direct access to the triangles array, use the Element version of `GetTriangles`, `OESurface::GetTrianglesElement`. Listing 4 is a code fragment that shows how to iterate over all the triangles of a surface.

Listing 4: Example of iterating over all triangles

```
unsigned int v1, v2, v3;
for(unsigned int i=0; i < surf.GetNumTriangles(); ++i)
{
    v1 = surf.GetTrianglesElement(i * 3);
    v2 = surf.GetTrianglesElement(i * 3 + 1);
    v3 = surf.GetTrianglesElement(i * 3 + 2);
}
```

2.1.3 Normal Vectors

The emphasis on triangle vertex ordering is to support the notion of the triangle's front and back. The front of the triangle is the side that the vertices appear ordered in a clockwise fashion. Surface normals can then be calculated to point out from the front of the triangle (and conversely from the back of the triangle).

The `OESurface` also supports vertex normals since vertices are often easier to work with analytically. Vertex normals are calculated by averaging all of the face normals of triangles of which the vertex is a part. The following figures contrast the differences between face and vertex normals respectively.

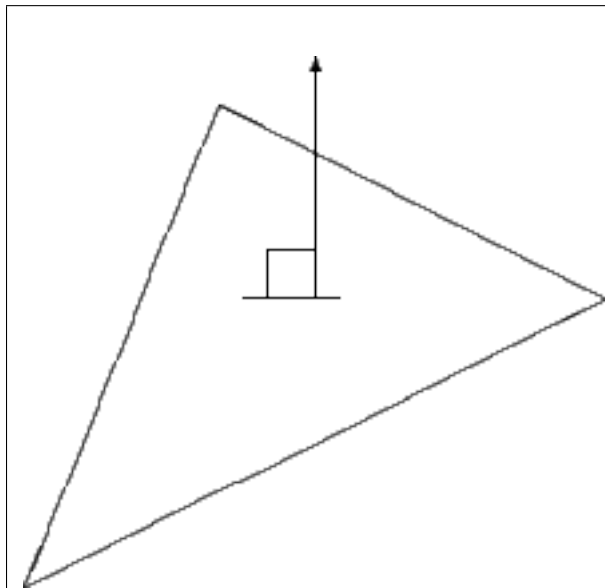


Figure 3
Face normals point out of perpendicular to the triangle

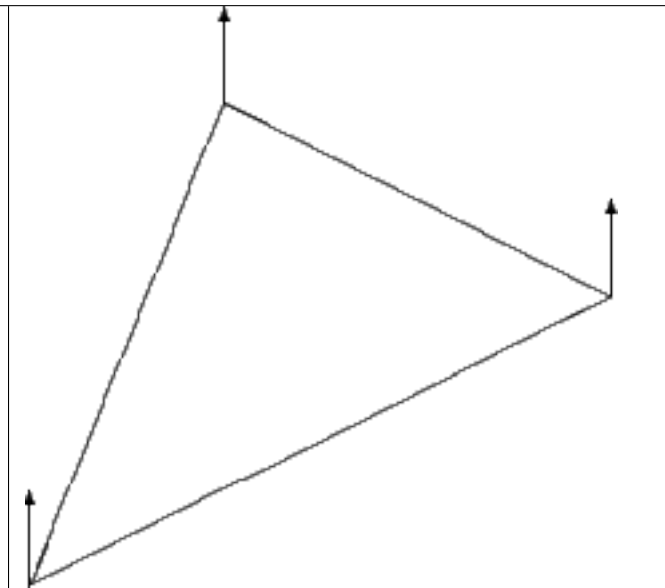


Figure 4
Vertex normals are the average of the vertices face normals

Vectors in Spicoli are represented by three floating point values. Normal vectors are always of unit length. They can be calculated and stored on a `OESurface` object by invoking the free functions `OECalculateNormals` and

OECalculateFaceNormals for vertex and face normals, respectively.

It is important to remember that the size of a normal array is determined by whether it is a face or vertex normal. For example the size of the vertex normal array is `GetNumVertices() * 3 floats`. While the size of the face normal array is `GetNumTriangles() * 3 floats`.

2.1.4 Specific Data

Specific data is any information that can be derived from the surface alone but not from any one triangle or vertex. Currently these properties include the following:

Curvature (float) Solvent accessibility of the vertex

Distance (float) The Euclidean distance to the vertex from another portion of the surface

Curvature follows a pragmatic computational chemistry definition. It is a property of solvent molecules, represented as spheres, packed onto the surface. *Figures 5, 6, and 7* demonstrate the two dimensional case of how solvent molecules are packed onto a surface. The first sphere is mapped adjacent to the vertex using the vertex's normal. Two more spheres are then packed adjacent to the surface and the starting sphere. The angle between these two spheres is used to calculate the accessibility to solvent of the initial sphere using the following formula:

$$100 * \frac{\theta - \pi}{\pi}$$

Where θ is in radians. For this simple case the angle is also a measure of the surface curvature, hence the name. In three dimensions steradians are required to accomplish the same functional form:

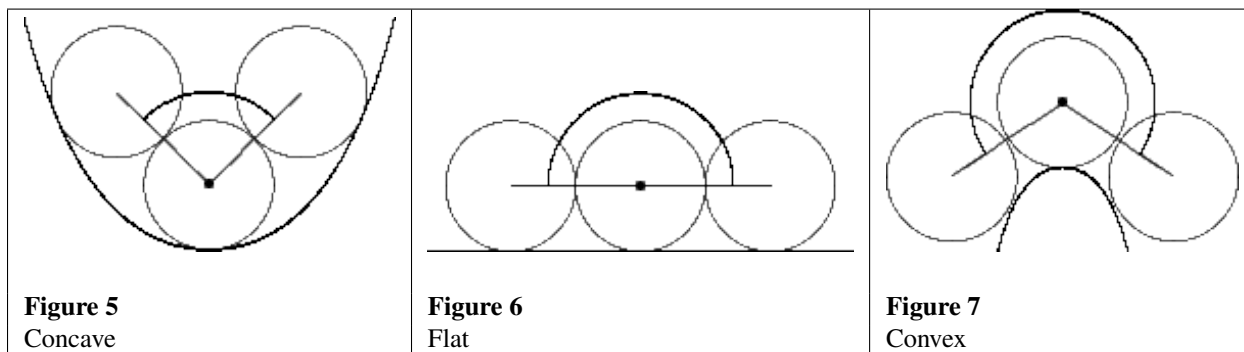
$$100 * \frac{\theta - 2\pi}{2\pi}$$

Where θ is in steradians. Therefore, a vertex's "curvature" falls into a range with the following bounds:

-100.0 Solvent that is completely secluded within the surface

0.0 Flat portion of the surface

100.0 Solvent that is completely detached from the surface



Distance is listed as specific data because it can be derived as the distance between different portions of the surface. This can be useful for measuring the thickness of a volume that is enclosed by a surface. Distances can also be measured to other arbitrary objects as well, such as molecules and other surfaces.

2.1.5 Associative Data

Associative data is not inherent from the first-class object definition of a surface. Instead, they are properties mapped onto the surface. Spicoli provides the following associative data arrays:

Atoms (unsigned int) Atom index for the vertex

Color (4 floats or 4 unsigned chars) Color of the vertex

Potential (float) Electrostatic potential at the vertex

To map chemical properties onto a surface it is useful to know which atoms are responsible for portions of the surface. When a surface is constructed from a molecule, a data array containing the corresponding atom index for each vertex is created. An atom's index can be obtained from the `OEAtomBase::GetIdx` method and is unique over the molecule. Refer to the *OEChem* manual for more information about atom indices.

To display chemical properties it is often useful to render them as either discrete colors or a spectrum of colors. The color data array allows the user to set a color for every vertex in the surface. When the surface is then read into a visualizer, such as *Vida*, the properties can easily be interpreted. Every vertex has the associated values: red, green, blue, and alpha. Alpha is the transparency of the vertex. If any value is retrieved as a `float`, then that value will range from 0 to 1 inclusive. If retrieved as an `unsigned char`, the value will range from 0 to 255 inclusive.

Electrostatic potentials can be calculated and displayed on a surface. This can be done with current *OpenEye* tools such as *Zap*.

The potentials array is essentially an array of floats the user can use to record analytical data for each vertex.

2.2 Surface Generation

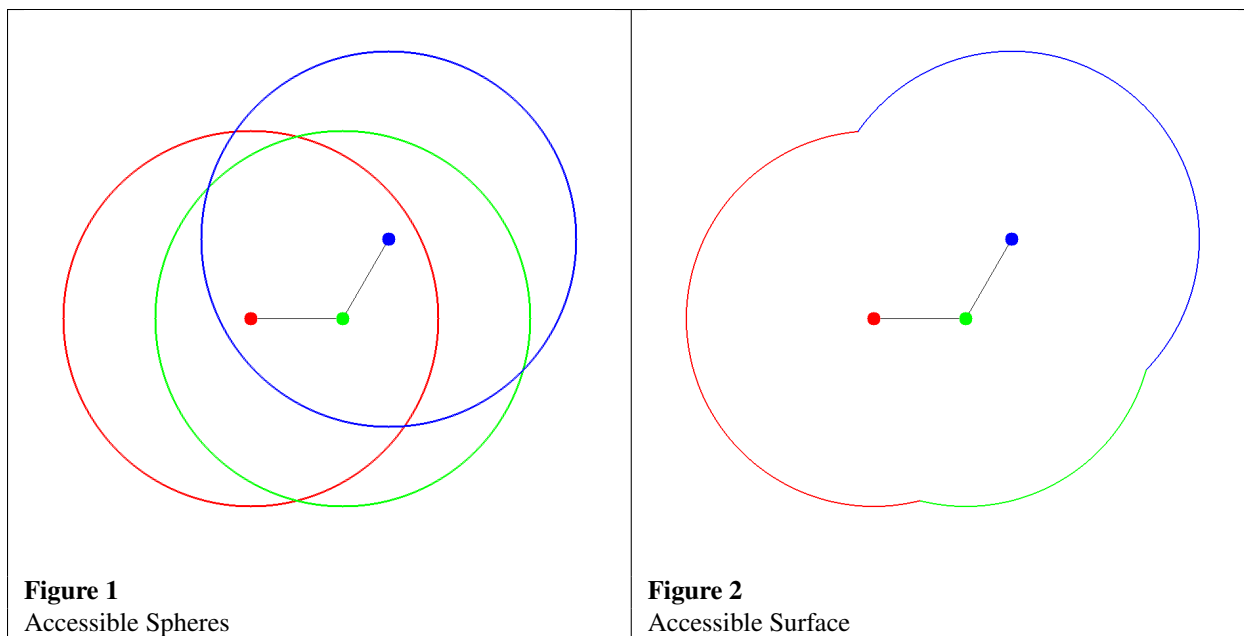
Spicoli can generate surfaces from the following other types of objects:

- *Molecules*
- *Grids*
- *Other Surfaces*
- *Geometric Primitives*

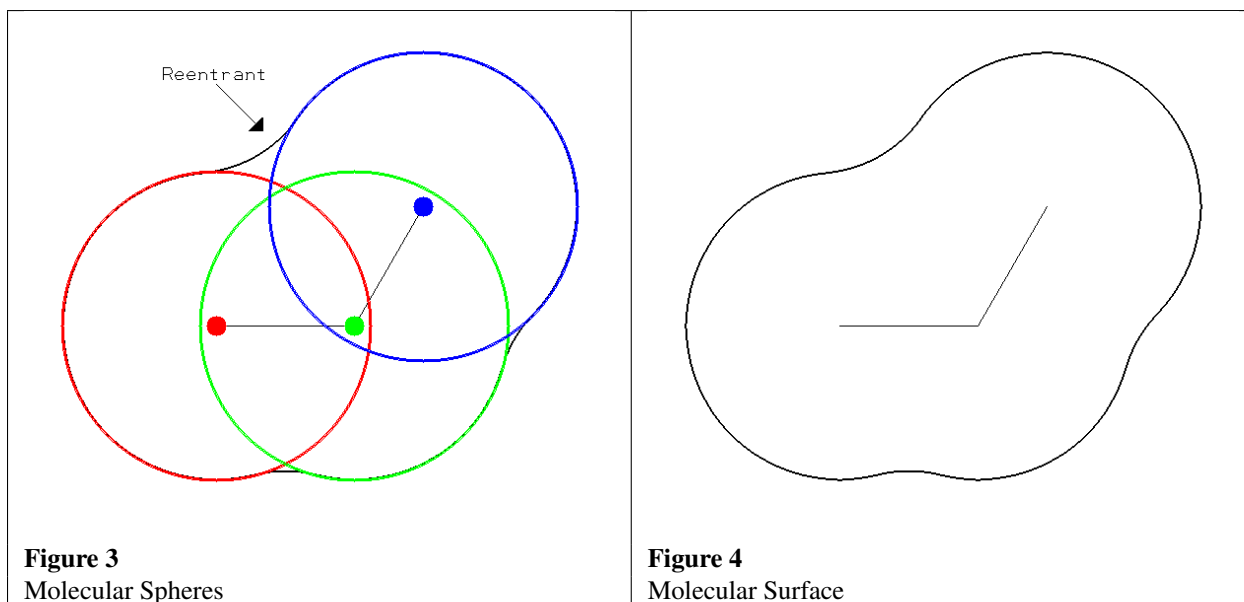
2.2.1 Molecules: Accessible vs Molecular

OESpicoli provides two functions for generating surfaces directly from molecules: `OEMakeAccessibleSurface` and `OEMakeMolecularSurface`. Both functions require the definition of a solvent molecule's probe radius. The default solvent is water with a probe radius of 1.4 Ångströms.

The accessible surface is created by representing each atom as a hard sphere [Lee-Richards-1971]. The radius of each sphere is the radius of the atom plus the probe radius. Figures 1 and 2 demonstrate how the spheres are packed together to form the surface. In the figures, portions of the surface are colored based upon each atom's contribution to the final accessible surface.



The molecular surface is composed of atom centered spheres plus reentrants [Connolly-1983]. Each sphere's radius is the atomic radii of the atom it is associated with. The defining characteristic of the molecular surface is the reentrant as shown in Figures 3 and 4. The reentrant models the portion of the molecule inaccessible to solvent. For this reason the volume enclosed by the molecular surface is sometimes referred to as the “solvent-excluded” volume.



Note: The atoms associated with every surface vertex can be obtained by using the `OESurface::GetAtoms` and `OESurface::GetAtomsElement` methods. For an accessible surface this is simple to define, it is the atom closest to the vertex. However, for the molecular surface it is not so clear because of the re-entrants. Typically, it is the closest atom to the vertex, but that is not guaranteed. If this guarantee is required you should call the `OESurfaceToMoleculeDistance` function on the molecule and surface.

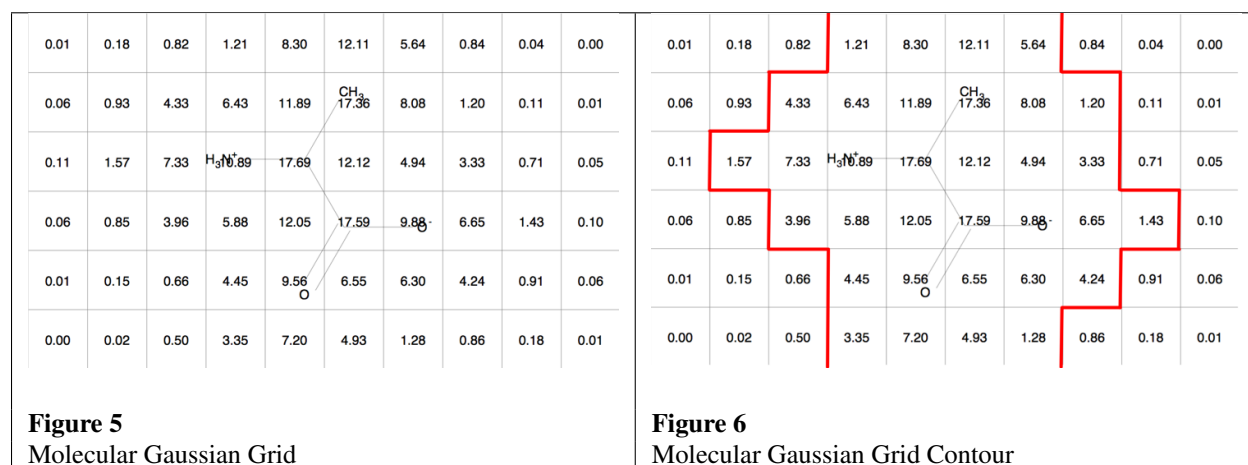
2.2.2 Grids

In Spicoli the construction of surfaces from a molecule proceeds through a grid intermediate. The space between grid points determines the resolution of the surface, i.e., how many triangles there are and the size of each triangle. Grids usually consist of equidistant points aligned along orthogonal axes, but this need not always be the case (for instance, electron density grids from crystallography).

Scalar values are placed at every grid point. Surfaces are constructed by tracing out a contour through the grid points. A contour is a separator of points based on whether they are greater than or less than a given value. The separator is a line in two dimensions and a surface in three dimensions. This is similar to how topographic maps use lines to convey elevation.

When dealing with surfaces the points on the grid with a value less than the chosen contour value are inside the surface and vice versa. `OEMakeSurfaceFromGrid` will generate a surface from a grid using a variation of the *marching cubes* algorithm.

Figure 5 is an example of a two dimensional molecular gaussian grid for a simple molecule arbitrarily oriented in the grid. Figure 6 shows a contour at 1.0 of that same molecular gaussian grid.



Hint: Grids and surfaces are fairly interchangeable. A grid can be constructed from a surface using the `OEMakeGridFromSurface` function. Then that grid can be used to recreate the original surface by using the `OEMakeSurfaceFromGrid` function.

2.2.3 Surface Subsets

Surface subsetting is done through the use of *Cliques*. The triangle, not the vertex, is the most physically relevant intrinsic property of the surface. To maintain a constant surface area for the sum of all partitions Spicoli will not duplicate triangles across surface partitions. However, this does not restrict the duplication of vertices across partitions.

Therefore, the sum of the surface area of every partition will equal the surface area of the whole surface. The sum of `OESurface::GetNumTriangles` over every partition will equal the total number of triangles in the whole surface. However, the sum of `OESurface::GetNumVertices` over every partition will not equal the total number of vertices in the whole surface.

See Also:

- `OESurfaceCropToClique`
- `OEMakeCliqueSurface`

2.2.4 Geometric Primitives

Spicoli provides the following free functions to construct surfaces from primitive geometric shapes:

- Boxes** `OEMakeBoxSurface`
- Spheres** `OEMakeSphericalSurface`
- Ellipsoids** `OEMakeEllipsoidSurface`

2.3 Surface Manipulation

Spicoli manipulates surfaces through the following two primary mechanisms:

- *Cliques*
- *Grids*

2.3.1 Cliques

A clique is an integral value associated with portions of the surface. They can be arbitrarily assigned in order to create groupings of vertices. Cliques in Spicoli are unsigned ints that can be associated with every vertex of the surface. Spicoli treats the zero clique as a special NULL clique. Therefore, the user should not use zero for performing any clique operations on the surface.

Cliques in Spicoli are collections of vertices. However, an ambiguity occurs when performing triangle-based operations on these cliques. Spicoli will automatically resolve vertex cliques into triangle cliques using the following rule: if two or more vertices in the triangle have the same clique value, the triangle is assigned that clique value; otherwise, if every vertex of the triangle has a different clique value then the triangle is arbitrarily assigned the lowest clique value of the three vertices.

See Also:

- `OESurfaceCropToClique`
- `OEMakeCliqueSurface`
- `OEMakeConnectedSurfaceCliques`
- `OESurfaceCliqueArea`
- `OESurfaceCliqueVolume`

2.3.2 Grids

Spicoli can construct grids from surfaces. Grid values are assigned using the distance to the nearest surface vertex to the grid point. This makes grid construction very expensive because every grid point must be compared to every surface vertex. The `OEVoxelizeMethod` namespace describes the available methods.

See Also:

- `OEMakeGridFromSurface`
- `OEMakeSurfaceFromGrid`

2.4 Surface Storage

2.4.1 File Formats

Spicoli supports the following storage formats:

- **.srf** Old GRASP format
- **.oesrf** OpenEye format based upon tagged binary files

The *GRASP* format is provided for backwards compatibility with older visualization programs. There were byte ordering issues with the format so OpenEye developed a more flexible format based upon the tagged binary IO available in OESystem.

See Also:

- `OESurface`
- `OESurfaceFileType`

2.4.2 Attached to Molecules

The OpenEye format allows surfaces to be attached to molecules and then written out to OEBinary (.oeb) files. A visualizer can then read in the molecule and surface without any other means of making the association. [Listing 1](#) demonstrates how to properly attach surfaces to molecules.

Listing 1: Attaching a surface as generic data to be written to OEB

```
OESurface surf;
OEMakeMolecularSurface(surf, mol);
mol.SetData<OESurface>("surface", surf);

oemolostream ofs("foo.oeb");
OEWriteMolecule(ofs, mol);
```

[Listing 2](#) demonstrates how to then read that surface back out of the OEB file.

Listing 2: Retrieving a surface attached as generic data from an OEB file

```
oemolistream ifs("foo.oeb");
OEReadMolecule(ifs, mol);

OESurface msrf = mol.GetData<OESurface>("surface");
```

See Also:

- `OEBase::SetData`
- `OEBase::GetData`

Note: Versions of Spicoli prior to 1.0.2 required calling `OEInitSurfaceHandlers` in order to have surfaces attached to molecules be read and written to and from OEB.

3.1 OESpicoli Classes

3.1.1 OESurface

```
class OESurface : public OESystem::OEBase
```

Constructors

```
OESurface()  
OESurface(const OESurface &rhs)  
OESurface(const OESurfaceImpl &rhs)
```

Default and copy constructors.

operator=

```
OESurface &operator=(const OESurface &rhs)  
OESurface &operator=(const OESurfaceImpl &rhs)
```

Assignment operator for a surface. Will copy all data from rhs to the surface.

operator+=

```
OESurface &operator+=(const OESurface &rhs)  
OESurface &operator+=(const OESurfaceImpl &rhs)
```

In-place addition of one surface to another. The titles are concatenated together with an underscore. The vertex indices in the triangles array of rhs are increased by `OESurface::GetNumVertices`. All other data is copied from rhs as well. In the case where one of the surfaces does not have data (Potentials, Atoms, etc) and the other does, the data array for the surface that does not will be zeroed out.

operator bool

```
operator bool() const
```

Whether the surface contains any data (vertices).

Clear

```
void Clear()
```

Delete and deallocate all data associated with this surface. This includes all `OEBase` data as well.

ClearVertexClique

```
bool ClearVertexClique()
```

Sets every vertex clique value to zero. Afterwards, `OESurface::IsVertexCliqueSet` will return `false`. This will prevent vertex clique values from being output by `OEWwriteSurface`.

CreateCopy

```
OESystem::OEBase *CreateCopy() const
```

GetAtoms

```
bool GetAtoms(unsigned int *atoms) const
```

Fills the memory pointed to by `atoms` with the atom index associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices()` unsigned ints.

GetAtomsElement

```
unsigned int GetAtomsElement(unsigned int n) const
```

Retrieves an element from the internal atoms array. The index `n` should be less than `GetNumVertices()`.

GetColor

```
bool GetColor(float *color) const  
bool GetColor(unsigned char *color) const
```

Fills the memory pointed to by `color` with the color associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices() * 4` floats or `GetNumVertices() * 4` unsigned chars, depending on which overload is used.

GetColorElement

```
void GetColorElement(unsigned int n, float &r, float &g, float &b, float &a) const
void GetColorElement(unsigned int n, unsigned char &r, unsigned char &g,
                    unsigned char &b, unsigned char &a) const
```

Retrieves an element from the internal color array. The index `n` should be less than `GetNumVertices()`.

GetCurvature

```
bool GetCurvature(float *curvature) const
```

Fills the memory pointed to by `curvature` with the curvature associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices()` floats.

GetCurvatureElement

```
float GetCurvatureElement(unsigned int n) const
```

Retrieves an element from the internal curvature array. The index `n` should be less than `GetNumVertices()`.

GetDataType

```
const void *GetDataType() const
```

GetDistance

```
bool GetDistance(float *distance) const
```

Fills the memory pointed to by `distance` with the distance each vertex is from another object. The array passed in should be large enough to hold `GetNumVertices()` floats.

GetDistanceElement

```
float GetDistanceElement(unsigned int n) const
```

Retrieves an element from the internal distance array. The index `n` should be less than `GetNumVertices()`.

GetFaceNormals

```
bool GetFaceNormals(float *faceNormals) const
```

Fills the memory pointed to by `faceNormals` with a normal vector associated with every triangle of the surface. The array passed in should be large enough to hold `GetNumTriangles() * 3` floats.

GetFaceNormalsElement

```
float GetFaceNormalsElement(unsigned int n) const
```

Retrieves an element from the internal face normals array. The index `n` should be less than `GetNumTriangles()` * 3. Note that each individual vector is composed of three floats: a float for every component of the vector. The user will normally need to call this function three times incrementing `n` by one each time to obtain the full 3-dimensional vector.

GetNormals

```
bool GetNormals(float *normals) const
```

Fills the memory pointed to by `normals` with a normal vector associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices() * 3` floats. A vertex normal is calculated by averaging the face normals of all the triangles in which the vertex resides.

GetNormalsElement

```
float GetNormalsElement(unsigned int n) const
```

Retrieves an element from the internal vertex normals array. The index `n` should be less than `GetNumVertices()` * 3. Note that each individual vector is composed of three floats: a float for every component of the vector. The user will normally need to call this function three times incrementing `n` by one each time to obtain the full 3-dimensional vector.

GetNumTriangles

```
unsigned int GetNumTriangles() const
```

Returns the number of triangles in the surface.

GetNumVertices

```
unsigned int GetNumVertices() const
```

Returns the number of vertices in the surface.

GetPotential

```
bool GetPotential(float *potential) const
```

Fills the memory pointed to by `potential` with the potential associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices()` floats.

GetPotentialElement

```
float GetPotentialElement(unsigned int n) const
```

Retrieves an element from the internal potentials array. The index `n` should be less than `GetNumVertices()`.

GetPotentialName

```
const char *GetPotentialName() const
```

Returns name of the potentials stored in the surface's potential array.

GetResolution

```
float GetResolution() const
```

Returns the grid spacing used in the surface construction.

GetTitle

```
const char *GetTitle() const
```

Returns the title of the surface.

GetTriangles

```
bool GetTriangles(unsigned int *triangles) const
```

Fills the memory pointed to by `triangles` with the vertex indices that compose every triangle in the surface. The vertex indices will obey the clockwise ordering rule described in the *Triangles* section. The array passed in should be large enough to hold `GetNumTriangles() * 3` unsigned ints.

GetTrianglesElement

```
unsigned int GetTrianglesElement(unsigned int n) const
```

Retrieves an element from the internal triangles array. The index `n` should be less than `GetNumTriangles() * 3`. Note that each individual triangle is composed of three integers. The user will normally need to call this function three times incrementing `n` by one each time to obtain all three vertices of the triangle.

GetVertexClique

```
bool GetVertexClique(unsigned int *vertexClique) const
```

Fills the memory pointed to by `vertexClique` with the clique values associated with every vertex of the surface. The array passed in should be large enough to hold `GetNumVertices()` unsigned ints.

GetVertexCliqueElement

unsigned int GetVertexCliqueElement(**unsigned int** n) **const**

Retrieves an element from the internal cliques array. The index *n* should be less than `GetNumVertices()`.

GetVertices

bool GetVertices(**float** *vertices) **const**

Fills the memory pointed to by *vertices* with the vertex coordinates of the surface. Vertices are aligned every three places in the array as described in the *Vertices* section. The array passed in should be large enough to hold `GetNumVertices() * 3` floats.

GetVerticesElement

float GetVerticesElement(**unsigned int** n) **const**

Retrieves an element from the internal vertex array. The index *n* should be less than `GetNumVertices() * 3`. Note that each individual vertex is composed of three floats. The user will normally need to call this function three times, incrementing *n* by one each time, to obtain all three dimensions of the vertex.

IsAtomsSet

bool IsAtomsSet() **const**

Determine whether the surface has atom indices associated with each vertex. Surfaces generated from `OEMakeAccessibleSurface` and `OEMakeAccessibleSurface` automatically set this data.

IsColorSet

bool IsColorSet() **const**

Determine whether the surface has color values associated with each vertex.

IsCurvatureSet

bool IsCurvatureSet() **const**

Determine whether the surface has curvature value calculated for each vertex.

IsDataType

bool IsDataType(**const void** *type) **const**

IsDistanceSet

```
bool IsDistanceSet() const
```

Determine whether the surface has a distance value associated with each vertex.

IsFaceNormalsSet

```
bool IsFaceNormalsSet() const
```

Determine whether the surface has face normals calculated for each triangle.

IsNormalsSet

```
bool IsNormalsSet() const
```

Determine whether the surface has vertex normals calculated for each vertex. Surfaces generated from `OEMakeAccessibleSurface` and `OEMakeAccessibleSurface` automatically set this data.

IsPotentialSet

```
bool IsPotentialSet() const
```

Determine whether the surface has a potential value associated with each vertex.

IsVertexCliqueSet

```
bool IsVertexCliqueSet() const
```

Determine whether the surface has a clique value associated with each vertex.

SetAtoms

```
bool SetAtoms(const unsigned int *atoms)
```

Sets the internal atoms array data to the values pointed to by `atoms`. This should be a pointer to an array of unsigned ints of length `GetNumVertices()`. Returns `true` upon success.

SetAtomsElement

```
bool SetAtomsElement(unsigned int n, unsigned int value)
```

Sets an element in the internal atoms array. The index `n` should be less than `GetNumVertices()`. Returns `true` upon success.

SetColor

```
bool SetColor(const float *color)
bool SetColor(const unsigned char *color)
```

Sets the internal color array data to the values pointed to by `color`. This should be a pointer to an array of floats or unsigned chars of length `GetNumVertices() * 4` or `GetNumVertices() * 4` respectively. Returns `true` upon success.

SetColorElement

```
bool SetColorElement(unsigned int n, float r, float g, float b, float a=1.0f)
bool SetColorElement(unsigned int n, unsigned char r, unsigned char g,
                     unsigned char b, unsigned char a=255)
```

Sets an element in the internal color array. The index `n` should be less than `GetNumVertices()`. The alpha value, `a`, defaults to be opaque. Returns `true` upon success.

SetCurvature

```
bool SetCurvature(const float *curvature)
```

Sets the internal curvature array data to the values pointed to by `curvature`. This should be a pointer to an array of floats of length `GetNumVertices()`. Returns `true` upon success.

SetCurvatureElement

```
bool SetCurvatureElement(unsigned int n, float value)
```

Sets an element in the internal curvature array. The index `n` should be less than `GetNumVertices()`. Returns `true` upon success.

SetDistance

```
bool SetDistance(const float *distance)
```

Sets the internal distance array data to the values pointed to by `distance`. This should be a pointer to an array of floats of length `GetNumVertices()`. Returns `true` upon success.

SetDistanceElement

```
bool SetDistanceElement(unsigned int n, float value)
```

Sets an element in the internal distance array. The index `n` should be less than `GetNumVertices`. Returns `true` upon success.

SetFaceNormals

```
bool SetFaceNormals(const float *faceNormals)
```

Sets the internal face normals array data to the values pointed to by `faceNormals`. This should be a pointer to an array of floats of length `GetNumTriangles() * 3`. Face normals are associated with triangles by their location in the array. Returns `true` upon success.

SetFaceNormalsElement

```
bool SetFaceNormalsElement(unsigned int n, float value)
```

Sets an element in the internal face normals array. The index `n` should be less than `GetNumTriangles() * 3`. Note that each individual vector is composed of three floats: a float for every component of the vector. The user will normally need to call this function three times incrementing `n` by one each time to obtain the full 3-dimensional vector. Returns `true` upon success.

SetNormals

```
bool SetNormals(const float *normals)
```

Sets the internal vertex normals array data to the values pointed to by `normals`. This should be a pointer to an array of floats of length `GetNumVertices() * 3`. Vertex normals are associated with vertices by their location in the array. Returns `true` upon success.

SetNormalsElement

```
bool SetNormalsElement(unsigned int n, float value)
```

Sets an element in the internal vertex normals array. The index `n` should be less than `GetNumTriangles() * 3`. Note that each individual vector is composed of three floats: a float for every component of the vector. The user will normally need to call this function three times incrementing `n` by one each time to obtain the full 3-dimensional vector. Returns `true` upon success.

SetNumTriangles

```
bool SetNumTriangles(unsigned int n)
```

Warning: Expert use only.

Set the number of triangles in this surface. Setting this to a value lower than `OESurface::GetNumTriangles` effectively erases triangles from the surface. The memory for the `OESurface` is not freed immediately, but cached for reuse. The `Clear` method should be used if deallocation is desired. Deallocation is also handled automatically by the class destructor.

If the user wants to add more triangles to the surface, `SetNumTriangles` should be called first to resize the internal triangles array. Note that the `*Element` methods should not be used after a call to `SetNumTriangles`. Any data reliant on the number of triangles should first be set using the array based methods. This includes `OESurface::SetTriangles` and `OESurface::SetFaceNormals`.

SetNumVertices

```
bool SetNumVertices(unsigned int n)
```

Warning: Expert use only.

Set the number of vertices in this surface. Setting this to a value lower than `GetNumVertices` effectively erases vertices from the surface. The memory for the `OESurface` is not freed immediately, but cached for reuse. The `OESurface::Clear` method should be used if deallocation is desired. Deallocation is also handled automatically by the class destructor.

If the user wants to add more vertices to the surface, `OESurface::SetNumVertices` should be called first to resize the internal vertices array. Note that the `*Element` methods should not be used after a call to `SetNumVertices`. Any data reliant on the number of vertices should first be set using the array based methods. These include the following:

- `OESurface::SetVertices`
- `OESurface::SetNormals`
- `OESurface::SetCurvature`
- `OESurface::SetDistance`
- `OESurface::SetAtoms`
- `OESurface::SetColor`
- `OESurface::SetPotential`
- `OESurface::SetVertexClique`

SetPotential

```
bool SetPotential(const float *potential)
```

Sets the internal potential array data to the values pointed to by `potential`. This should be a pointer to an array of floats of length `GetNumVertices() * sizeof(float)`. Returns true upon success.

SetPotentialElement

```
bool SetPotentialElement(unsigned int n, float value)
```

Sets an element in the internal potential array. The index `n` should be less than `GetNumVertices()`. Returns true upon success.

SetPotentialName

```
void SetPotentialName(const char *name)
```

Set the name of the potentials that are set on the surface.

SetTitle

```
bool SetTitle(const char *title)
bool SetTitle(const std::string &title)
```

Set the title of the surface.

SetTriangles

```
bool SetTriangles(const unsigned int *triangles)
```

Sets the internal triangles array data to the values pointed to by `triangles`. This should be a pointer to an array of unsigned ints of length `GetNumTriangles() * 3`. Returns `true` upon success.

SetTrianglesElement

```
bool SetTrianglesElement(unsigned int n, unsigned int value)
```

Sets an element in the internal triangles array. The index `n` should be less than `GetNumTriangles() * 3`. Returns `true` upon success.

SetVertexClique

```
bool SetVertexClique(const unsigned int *vertexClique)
```

Sets the internal vertex clique array data to the values pointed to by `vertexClique`. This should be a pointer to an array of unsigned ints of length `GetNumVertices()`. Returns `true` upon success.

SetVertexCliqueElement

```
bool SetVertexCliqueElement(unsigned int n, unsigned int value)
```

Sets an element in the internal vertex clique array. The index `n` should be less than `GetNumVertices()`. Returns `true` upon success.

SetVertices

```
bool SetVertices(const float *vertices)
```

Sets the internal vertices array data to the values pointed to by `vertices`. This should be a pointer to an array of floats of length `GetNumVertices() * 3`. Returns `true` upon success.

SetVerticesElement

```
bool SetVerticesElement(unsigned int n, float value)
```

Sets an element in the internal vertices array. The index n should be less than `GetNumVertices() * 3`. Returns `true` upon success.

3.2 OESpicoli Constants

3.2.1 OESurfaceFileType

This namespace contains constants for passing to the `OEWriteSurface` function.

See Also:

Surface Storage

UNDEFINED

Used to indicate errors.

Grasp

The legacy *GRASP* file format. Will not store all data present on the `OESurface` object.

OESurface

Binary Surface format developed by OpenEye. It will store all `OESurface` data.

3.2.2 OEVoxelizeMethod

The `OEVoxelizeMethod` namespace is used to instruct the `OEMakeGridFromSurface` function how to voxelize the surface onto the grid. A voxel is the three dimensional equivalent of a pixel.

See Also:

The `OEMakeGridFromSurface` function for how to pass this constant into the function.

Distance

Each grid point is filled with the minimum distance squared from the surface. The distance is negative if the grid point is inside the surface.

Note: The distance is squared to avoid calling the costly square root function.

Gaussian

Same as distance, but the value placed in the grid is plotted as a Gaussian function. This means each gridpoint is set to $e^{-0.33*d^2}$, where d is the minimum distance to the surface. Note, where d can still be negative inside the surface. Therefore, the grid values have the following correlation to the volume enclosed by the surface:

Inside the Surface Values are greater than 1.0

At the Surface Values are 1.0

Outside the Surface Values are less than 1.0 and greater than 0.0

Blank

Same as distance, but grid points inside the surface are set to 1.

Blur

Discern the shape of the volume enclosed by the surface. This is done by mapping a gaussian for every grid point inside the volume. This smears out the surface boundary a bit and allowing a smooth drop off from the surface.

Default

The default voxelization method is `OEVoxelizeMethod::Distance`.

3.3 OESpicoli Functions

3.3.1 OEAddSurfaces

```
bool OEAddSurfaces(OESurface &surf1, const OESurface &surf2)
```

Adds surf2 to surf1 using `OESurface::operator+=`. Always returns true.

3.3.2 OECalculateFaceNormals

```
bool OECalculateFaceNormals(OESurface &surf)
```

Will calculate a normal vector for every triangle of the surface and store them on the surf. To retrieve the vectors after calculation, use the `OESurface::GetFaceNormals` or `OESurface::GetFaceNormalsElement` methods.

See Also:

Normal Vectors

3.3.3 OECalculateNormals

```
bool OECalculateNormals(OESurface &surf)
```

Will calculate a normal vector for every vertex of the surface and store them on the surf. To retrieve the vectors after calculation, use the `OESurface::GetNormals` or `OESurface::GetNormalsElement` methods.

See Also:

Normal Vectors

3.3.4 OECalculateSurfaceCurvature

```
bool OECalculateSurfaceCurvature(OESurface &surf, const OEChem::OEMolBase &mol,
                                  float probeRadius=1.4f)
```

Will calculate the surface curvature at every vertex according to the definition given in the *Specific Data* section. The data can then be retrieved using the `OESurface::GetCurvature` or `OESurface::GetCurvatureElement` methods.

The molecule passed in as `mol` is used to construct a grid marking which regions of space are inaccessible to solvent molecules. Typically this is just the molecule used to construct the surface.

3.3.5 OECalculateTriangleAreas

```
bool OECalculateTriangleAreas(const OESurface &surf, float *areas)
```

Will calculate the area of every triangle of `surf` and store the result in the array `areas` of that should be large enough to hold `GetNumTriangles()` floats. The array is indexed by the triangle indices.

Listing 1 demonstrates how to calculate the surface area contribution from each atom. It is assumed `surf` is an `OESurface` created from the molecule `mol`.

Listing 1: Calculates the surface area contribution from each atom

```
float *areas = new float[surf.GetNumTriangles()];
OECalculateTriangleAreas(surf, areas);

float *atomareas = new float[mol.GetMaxAtomIdx()];
memset(atomareas, 0, mol.GetMaxAtomIdx()*sizeof(float));

unsigned int v1, v2, v3, a1, a2, a3;
for (unsigned int i=0; i < surf.GetNumTriangles(); i++)
{
    v1 = surf.GetTrianglesElement(i * 3);
    v2 = surf.GetTrianglesElement(i * 3 + 1);
    v3 = surf.GetTrianglesElement(i * 3 + 2);

    a1 = surf.GetAtomsElement(v1);
    a2 = surf.GetAtomsElement(v2);
    a3 = surf.GetAtomsElement(v3);

    atomareas[a1] += areas[i]/3.0f;
    atomareas[a2] += areas[i]/3.0f;
    atomareas[a3] += areas[i]/3.0f;
}

for (OEIter<OEAtomBase> atom = mol.GetAtoms(); atom; ++atom)
    std::cout<<"atom " <<atom->GetIdx()<<" area = " <<atomareas[atom->GetIdx()]<<std::endl;

delete areas;
delete atomareas;
```

3.3.6 OEGetCenterAndExtents

```
void OEGetCenterAndExtents(const OESurface &surf, float *center, float *extents)
```

Iterates through all the vertices in `surf` to find the center and extents of the surface in Cartesian coordinates. The center will be placed in the memory pointed to by `center` that should be large enough to hold 3 floats. The extents will be placed in the memory pointed to by `extents` that should be large enough to hold 3 floats. The extents are the difference between the maximum of the surface along any dimension and the minimum along that same dimension.

See Also:

- `OEMakeGridFromCenterAndExtents`

3.3.7 OEGetSurfaceFileType

```
unsigned int OEGetSurfaceFileType(const char *ext)
```

Returns a constant from the `OESurfaceFileType` namespace for the file extension `ext`.

3.3.8 OEGetSurfaceFormatExtension

```
const char *OEGetSurfaceFormatExtension(unsigned int tag)
```

Returns a comma-separated list of possible file extensions corresponding to the specified parameter. The parameter, `tag`, should be drawn from the `OESurfaceFileType` namespace.

3.3.9 OEGetSurfaceFormatString

```
const char *OEGetSurfaceFormatString(unsigned int tag)
```

Returns the name of the file format associated with the symbolic constant `tag` from the `OESurfaceFileType` namespace.

3.3.10 OEInitSurfaceHandlers

```
void OEInitSurfaceHandlers(OEChem::oemolstreambase &fs)
bool OEInitSurfaceHandlers(OESystem::OEBinaryIOHandlerBase &b)
```

Needs to be called in order for the `OEBinary` writer to be able to write surfaces as generic data tagged to a molecule or any `OEBase` object.

Warning: Deprecated as of Spicoli 1.0.2. No longer needs to be called.

See Also:

Attached to Molecules

3.3.11 OEInvertSurface

```
bool OEInvertSurface(OESurface &surf)
```

Reverses all the triangle orderings and normal vectors in the surface. This essentially turns the surface inside-out because all the normals facing out are turned in and vice versa.

Hint: Water pockets inside of protein structures have their normals facing into the protein. This can be used to make the disconnected surface a surface enclosing the occluded water.

3.3.12 OEIsReadableSurface

```
bool OEIsReadableSurface(unsigned int type)
bool OEIsReadableSurface(const std::string &filename)
```

Returns `true` if the supplied file format is readable by Spicoli. The `filename` can be a file name or a file extension. The `unsigned int` parameter value should be drawn from the `OESurfaceFileType` namespace.

3.3.13 OEIsWritableSurface

```
bool OEIsWritableSurface(unsigned int type)
bool OEIsWritableSurface(const std::string &filename)
```

Returns `true` if the supplied file format is writable by Spicoli. The `filename` can be a file name or a file extension. The `unsigned int` parameter value should be drawn from the `OESurfaceFileType` namespace.

3.3.14 OEMakeAccessibleSurface

```
bool OEMakeAccessibleSurface(OESurface &surf, const OEChem::OEMolBase &mol,
                             float resolution=0.5f, float probeRadius=1.4f)
bool OEMakeAccessibleSurface(OESurface &surf, const float *coords,
                             const float *radii, int natoms,
                             float resolution=0.5f, float probeRadius=1.4f)
```

Fill the surface `surf` with the accessible surface for the molecule `mol`. See the *Molecules: Accessible vs Molecular* section for a full description of the accessible surface. The atomic radius of each atom in the molecule should already be set before calling this function. The suggested radii can be set with `OEAssignBondiVdWRadii`.

The resolution is the grid spacing to use during surface construction. This is typically scaled with the size of the molecule. For most molecules, 0.5 is good.

The default `probeRadius` is 1.4 Å, the radius of water approximated as a sphere.

The overloaded version of the function takes coordinate and radii arrays, `coords` and `radii` respectively, of length `natoms` to create the accessible surface.

This function will set the vertex normals and atoms arrays on the surface.

See Also:

Molecules: Accessible vs Molecular

3.3.15 OEMakeBitGridFromSurface

```
bool OEMakeBitGridFromSurface(OESystem::OEScalarGrid &grid,
                             const OESurface &surf)
bool OEMakeBitGridFromSurface(OESystem::OEScalarGrid &grid,
                             const OESurface &surf, float spacing, float buffer)
```

Fill `grid` with a either 0.0 or 1.0 marking whether that grid points lies inside or outside the volume enclosed by the surface. 0.0 indicates the point is outside the surface. 1.0 indicates the point is inside the surface.

3.3.16 OEMakeBoxSurface

```
bool OEMakeBoxSurface(OESurface &surf, const float *center,
                    const float *extents)
```

Creates a box in `surf` located at `center` and extending `extents` in each dimension. Each of the six faces of the box will be composed of two triangles.

3.3.17 OEMakeCavitySurfaces

```
bool OEMakeCavitySurfaces(const OEChem::OEMolBase &mol, OESurface &surf,
                        float resolution=0.5f)
```

Create a surface, `surf`, from the macromolecule, `mol`, representing where solvent can be occluded inside the molecule. Therefore, this will not give cavities contiguous with the outside of the molecule. The `resolution` is the grid spacing to use during surface construction.

3.3.18 OEMakeCliqueSurface

```
bool OEMakeCliqueSurface(OESurface &newsurf, const OESurface &surf,
                        unsigned int clique)
```

Creates a new surface in `newsurf` from the triangles in `surf` marked with the vertex clique value `clique`. Zero is not a valid clique. See the *Cliques* section on how vertex cliques are assigned to triangles. See the *Surface Subsets* section for an explanation of how Spicoli handles subset boundaries. Returns `true` if successful.

3.3.19 OEMakeComplexCavities

```
bool OEMakeComplexCavities(const OEChem::OEMolBase &mol1,
                          const OEChem::OEMolBase &mol2, OESurface &surf,
                          float resolution=0.5f)
```

3.3.20 OEMakeConnectedSurfaceCliques

```
unsigned int OEMakeConnectedSurfaceCliques(OESurface &surf)
```

Assigns a different clique value, starting at 1, to each connected component of the surface. The number of connected components is returned. [Listing 2](#) demonstrates cropping a surface to its largest component.

Listing 2: Extracting the largest portion of the surface

```
unsigned int nclqs = OEMakeConnectedSurfaceCliques(surf);

unsigned int maxclq = 0;
float maxarea=0.0f;
for (unsigned int i=1; i <= nclqs; i++)
{
    float area = OESurfaceCliqueArea(surf, i);
    if (maxarea < area)
    {
        maxarea = area;
        maxclq = i;
    }
}
OESurfaceCropToClique(surf, maxclq);
```

3.3.21 OEMakeEllipsoidSurface

```
bool OEMakeEllipsoidSurface(OESurface &surf, float *center, float a, float b,
                             float c, float *dir1, float *dir2, float *dir3,
                             int level=4)
```

3.3.22 OEMakeGridFromSurface

```
bool OEMakeGridFromSurface(OESystem::OEScalarGrid &grid, const OESurface &surf,
                           unsigned int method=OEVoxelizeMethod::Default)
```

Fill grid with a voxelized representation of the surface by the method passed as the method parameter. The constants for method parameter are defined by the `OEVoxelizeMethod` namespace.

See Also:

The `OEVoxelizeMethod` namespace.

3.3.23 OEMakeMolecularSurface

```
bool OEMakeMolecularSurface(OESurface &surf, const OEChem::OEMolBase &mol,
                             float resolution=0.5f, float probeRadius=1.4f)
bool OEMakeMolecularSurface(OESurface &surf, const float *coords,
                             const float *radii, int natoms, float resolution=0.5f,
                             float probeRadius=1.4f)
```

Fill the surface surf with the molecular surface for the molecule mol. See *Molecules: Accessible vs Molecular* section for a full description of the molecular surface. The atomic radius of each atom in the molecule should already be set before calling this function. The suggested radii can be set with `OEAssignBondiVdWRadii`.

The resolution is the grid spacing to use during surface construction. This is typically scaled with the size of the molecule. For most molecules 0.5 is good.

The default probeRadius is 1.4 Å, the radius of water approximated as a sphere.

The overloaded version of the function takes a coordinate and radii array, `coords` and `radii` respectively, of length `natoms` to create the accessible surface.

This function will set the array of vertex normals and array of atom indices on the surface.

See Also:

Molecules: Accessible vs Molecular

3.3.24 OEMakeSphericalSurface

```
bool OEMakeSphericalSurface(OESurface &surf, float *center, float radius,
                           unsigned int level=4)
```

Creates a spherical surface in `surf` with the center located at `center` (an array of three floats) with the radius `radius`. The `level` argument determines the number of triangles used to approximate the sphere. The number of triangles is equal to $level (level + 1) * 8$. For example the default level of 4 generates 160 triangles which is generally considered to be a good sphere to the human eye.

3.3.25 OEMakeSurfaceFromGrid

```
bool OEMakeSurfaceFromGrid(OESurface &surf, const OESystem::OESkewGrid &grid,
                           float contour)
bool OEMakeSurfaceFromGrid(OESurface &surf,
                           const OESystem::OEFixedGrid<float> &grid,
                           float contour)
bool OEMakeSurfaceFromGrid(OESurface &surf,
                           const OESystem::OEFixedGrid<float> &grid,
                           float contour, float resolution)
```

An implementation of the *marching cubes* algorithm to create a surface, `surf`, through the values in the grid, `grid`. Grid points with values less than `contour` are inside the surface and the converse is true for grid points outside the surface.

See Also:

The *Grids* section for more information on converting grids to surfaces.

3.3.26 OEMakeVoidVolume

```
bool OEMakeVoidVolume(const OEChem::OEMolBase &mol1,
                     const OEChem::OEMolBase &mol2,
                     OESystem::OEScalarGrid &grid, float resolution)
```

3.3.27 OEReadSurface

```
bool OEReadSurface(const std::string &fname, OESurface &surf)
bool OEReadSurface(OEPlatform::oeistream &ifstream, OESurface &surf,
                  unsigned int type)
```

Read the contents of the file named `fname` into the surface `surf`. Returns `false` if `fname` is not a valid surface file name.

An overload is provided to take any arbitrary `oeistream`.

3.3.28 OESetSurfaceColor

```
bool OESetSurfaceColor(OESurface &surf, float r, float g, float b, float a=1.0f)
bool OESetSurfaceColor(OESurface &surf, unsigned char r, unsigned char g,
                        unsigned char b, unsigned char a=255)
```

Sets the color of the entire surface to the specified color.

3.3.29 OESetSurfacePotentials

```
bool OESetSurfacePotentials(OESurface &surf,
                            const OESystem::OEScalarGrid &grid)
```

Linearly interpolates the values in the `OEScalarGrid` `grid` onto the surface `surf`. The scalar value associated with each vertex can then be accessed using the `OESurface::GetPotential` and `OESurface::GetPotentialElement` methods.

3.3.30 OESpicoliGetArch

```
const char *OESpicoliGetArch()
```

3.3.31 OESpicoliGetLicensee

```
bool OESpicoliGetLicensee(std::string &licensee)
```

3.3.32 OESpicoliGetPlatform

```
const char *OESpicoliGetPlatform()
```

Return the internal build string used by OpenEye Scientific Software to identify a platform. The format of these strings may change over time, and future distributions may contain different values even when using the same operating system, compiler and processor. For example, on a `x86_64` Red Hat Enterprise Linux box this would return `redhat-RHEL5-g++4.1-x64`.

3.3.33 OESpicoliGetRelease

```
const char *OESpicoliGetRelease()
```

Return the release name of the `OESpicoli` library being used. This returns a value similar to `1.0` for production versions of the library, and `1.0 debug` for the checking version of the library.

3.3.34 OESpicoliGetSite

```
bool OESpicoliGetSite(std::string &site)
```

3.3.35 OESpicoliGetVersion

```
unsigned int OESpicoliGetVersion()
```

Return the version number of the library being used. This is an unsigned integer value indicating the date on which the library was built, for example 20020903, for the 3rd of September 2002. This value should be used when reporting problems, and unlike the release string, may be used in comparisons if needed.

3.3.36 OESpicoliIsLicensed

```
bool OESpicoliIsLicensed(const char *feature=0, unsigned int *expdate=0)
```

Determine whether a valid license file is present. This function may be called without a legitimate run-time license to determine whether it is safe to call any of Spicoli's functionality.

The features argument can be used to check for a valid license to Spicoli along with that feature. For example, to verify that Spicoli can be used from Python:

```
if (!OESpicoliIsLicensed("python"))
    OThrow.Warning("OESpicoli is not licensed for the python feature");
```

The second argument can be used to get the expiration date of the license. This is an array of size three with the date returned as {day, month, year}. Even if the function returns false due to an expired license, the expdate will show that expiration date. The array full of zeroes implies that no license or date was found.

```
unsigned int expdate[3];
if (OESpicoliIsLicensed(0, expdate))
{
    OThrow.Info("License expires: day: %d month: %d year: %d",
               expdate[0], expdate[1], expdate[2]);
}
```

3.3.37 OESurfaceArea

```
float OESurfaceArea(const OESurface &surf)
```

Returns the total surface area of surf.

3.3.38 OESurfaceCliqueArea

```
float OESurfaceCliqueArea(const OESurface &surf, unsigned int clique)
```

Return the surface area for a specified clique. Zero is not a valid clique. See Section 2.3.1 on how vertex cliques are assigned to triangles.

3.3.39 OESurfaceCliqueCentroid

```
bool OESurfaceCliqueCentroid(const OESurface &surf, unsigned int clique,
                             float *centroid)
```

Return the centroid coordinates (x,y,z) in centroid for clique. Zero is not a valid clique.

See Also:

The *Cliques* section on how vertex cliques are assigned to triangles.

3.3.40 OESurfaceCliqueVolume

```
float OESurfaceCliqueVolume(const OESurface &surf, unsigned int clique)
```

Return the volume for a specified clique. Zero is not a valid clique.

See Also:

The *Cliques* section on how vertex cliques are assigned to triangles.

Note: The volume may be negative if the clique is an open surface. However, the sum of all surface partitions will equal the total volume of the enclosed surface.

3.3.41 OESurfaceCropToClique

```
bool OESurfaceCropToClique(OESurface &surf, unsigned int clique)
```

Trim surf of the triangles not marked with the vertex clique value clique. Zero is not a valid clique. See Section 2.3.1 on how vertex cliques are assigned to triangles. Returns true if successful.

See Also:

- The *Cliques* section on how vertex cliques are assigned to triangles.
- The *Surface Subsets* section for an explanation of how spicoli handles subset boundaries.

3.3.42 OESurfaceIsOpen

```
bool OESurfaceIsOpen(const OESurface &surf)
```

Returns whether the surface is open. A surface is open if any triangle does not border exactly three other triangles.

3.3.43 OESurfaceToMoleculeDistance

```
bool OESurfaceToMoleculeDistance(OESurface &surf, const OEChem::OEMolBase &mol)
```

Will calculate the minimum distance between every vertex in the surface (surf) and every atom in the molecule (mol). The minimum distance for each vertex is then stored on the surface to be accessed via the `OESurface::GetDistance` and `OESurface::GetDistanceElement` methods. Distances are calculated as the euclidean distance between the vertex and atom center minus the atomic radii. Note that this can result in distances that are negative to denote that they are inside the molecule. If atomic radii are not specified on the molecule `BondiVdWRadii` will be used.

This function will also reassign the atom indices associated with every vertex of the surface. To access these changes use the `OESurface::GetAtoms` and `OESurface::GetAtomsElement` methods.

3.3.44 OESurfaceVolume

```
float OESurfaceVolume(const OESurface &surf)
```

Returns the volume enclosed by the surface.

3.3.45 OEWriteSurface

```
bool OEWriteSurface(const std::string &fname, const OESurface &surf)
bool OEWriteSurface(OEPlatform::oeostream &ofs, const OESurface &surf,
                    unsigned int type)
```

Write surf to the surface file `fname`. Returns `false` if `fname` is not a valid surface file name. This will overwrite the previous file if it existed.

The overload that takes an `oeostream` can be used to write multiple surfaces to the same destination. Note that the format must be specified by a constant from the `OESurfaceFileType` namespace as the `type` argument. The `OESurfaceFileType::Grasp` format does not support this kind of use. Returns `true` if written successfully.

GLOSSARY AND BIBLIOGRAPHY

4.1 Glossary

marching cubes (online: http://en.wikipedia.org/wiki/Marching_cubes)

GRASP GRASP is a molecular visualization and analysis program. It is particularly useful for the display and manipulation of the surfaces of molecules and their electrostatic properties. (online: http://wiki.c2b2.columbia.edu/honiglab_public/index.php/Software:GRASP)

4.2 Bibliography

RELEASE NOTES

5.1 SpicoliTK 1.1.1

5.1.1 Bug fixes

- Minor documentation fixes.
- `OEMakeComplexCavities` example program now properly suppresses hydrogen and passes the protein and ligand to the function in the proper order.

5.2 SpicoliTK 1.1.0

5.2.1 New Features

- Added the `OEMakeBitGridFromSurface` function. This is a lot faster than using `OEMakeGridFromSurface` to generate a lookup grid for whether a point is inside or outside a surface.

5.2.2 Bug fixes

- The previous release (1.0.2) introduced the feature that didn't require the user to call `OEInitSurfaceHandlers`. This didn't work sometimes when statically linking an executable. This was because the library initialization code that needed to get run was getting discarded by the linker because none of the functions in that translation unit were being called. The library initialization code has been migrated to the same translation unit as the `OESurface` since these functions always have to be called to attach a molecule as generic data.

Note: This only affected C++ users.

5.3 SpicoliTK 1.0.2

5.3.1 New features

- The user is no longer required to call `OEInitSurfaceHandlers` in order to attach surfaces to molecules and then write them out to OEB. This occurs at library link time now.

5.3.2 Minor bug fixes

- `OESurface::operator=` now copies the OEBase data.
- `OEMakeCliqueSurface` and `OESurfaceCropToClique` now preserve OEBase data.

5.4 SpicoliTK 1.0.1

5.4.1 Bug fixes

- `OEinvertSurface` properly inverts all surface normals.
- `OEMakeCavitySurfaces` was totally busted before. Added a default resolution of 0.5.
- Added a default resolution of 0.5 to `OEMakeComplexCavitySurfaces`.
- `OEMakeBoxSurface` now takes a `const float *` as it should.

5.5 Indices and tables

- *Index*
- *Search Page*

BIBLIOGRAPHY

- [Nicholls-2004] Spicoli: A Surface Toolkit, dude. Anthony Nicholls, Presentation in CUP V, **2004** (online: www.eyesopen.com/about/events/archives/cup_v/nicholls/Spicoli.ppt)
- [Sharp-Nicholls-1991] KA Sharp, A Nicholls, RF Fine, and B Honig **Reconciling the magnitude of the microscopic and macroscopic hydrophobic effects** *Science*, Vol. 252, pp. 106–109, **1991**. (online: <http://www.sciencemag.org/cgi/content/abstract/252/5002/106>)
- [Connolly-1983] M. L. Connolly **Analytical molecular surface calculation** *Journal of Applied Crystallography*, Vol. 16, pp. 548–558, **1983** (online: <http://journals.iucr.org/j/issues/1983/05/00/issconts.html>)
- [Lee-Richards-1971] B. Lee and F. M. Richards **The interpretation of protein structures: Estimation of static accessibility** *Journal of Molecular Biology*, Vol. 55, pp. 379–380, **1971** (online: [http://dx.doi.org/10.1016/0022-2836\(71\)90324-X](http://dx.doi.org/10.1016/0022-2836(71)90324-X))

INDEX

C

Clear
 OESpicoli::OESurface, 14
ClearVertexClique
 OESpicoli::OESurface, 14
Constructors
 OESpicoli::OESurface, 13
CreateCopy
 OESpicoli::OESurface, 14

G

GetAtoms
 OESpicoli::OESurface, 14
GetAtomsElement
 OESpicoli::OESurface, 14
GetColor
 OESpicoli::OESurface, 14
GetColorElement
 OESpicoli::OESurface, 15
GetCurvature
 OESpicoli::OESurface, 15
GetCurvatureElement
 OESpicoli::OESurface, 15
GetDataType
 OESpicoli::OESurface, 15
GetDistance
 OESpicoli::OESurface, 15
GetDistanceElement
 OESpicoli::OESurface, 15
GetFaceNormals
 OESpicoli::OESurface, 15
GetFaceNormalsElement
 OESpicoli::OESurface, 16
GetNormals
 OESpicoli::OESurface, 16
GetNormalsElement
 OESpicoli::OESurface, 16
GetNumTriangles
 OESpicoli::OESurface, 16
GetNumVertices
 OESpicoli::OESurface, 16

GetPotential
 OESpicoli::OESurface, 16
GetPotentialElement
 OESpicoli::OESurface, 17
GetPotentialName
 OESpicoli::OESurface, 17
GetResolution
 OESpicoli::OESurface, 17
GetTitle
 OESpicoli::OESurface, 17
GetTriangles
 OESpicoli::OESurface, 17
GetTrianglesElement
 OESpicoli::OESurface, 17
GetVertexClique
 OESpicoli::OESurface, 17
GetVertexCliqueElement
 OESpicoli::OESurface, 18
GetVertices
 OESpicoli::OESurface, 18
GetVerticesElement
 OESpicoli::OESurface, 18
GRASP, 37

I

IsAtomsSet
 OESpicoli::OESurface, 18
IsColorSet
 OESpicoli::OESurface, 18
IsCurvatureSet
 OESpicoli::OESurface, 18
IsDataType
 OESpicoli::OESurface, 18
IsDistanceSet
 OESpicoli::OESurface, 19
IsFaceNormalsSet
 OESpicoli::OESurface, 19
IsNormalsSet
 OESpicoli::OESurface, 19
IsPotentialSet
 OESpicoli::OESurface, 19
IsVertexCliqueSet

OESpicoli::OESurface, 19

M

marching cubes, 37

O

OESpicoli::OEAddSurfaces, 25

OESpicoli::OECalculateFaceNormals, 25

OESpicoli::OECalculateNormals, 25

OESpicoli::OECalculateSurfaceCurvature, 26

OESpicoli::OECalculateTriangleAreas, 26

OESpicoli::OEGetCenterAndExtents, 27

OESpicoli::OEGetSurfaceFileType, 27

OESpicoli::OEGetSurfaceFormatExtension, 27

OESpicoli::OEGetSurfaceFormatString, 27

OESpicoli::OEInitSurfaceHandlers, 27

OESpicoli::OEInvertSurface, 28

OESpicoli::OEIsReadableSurface, 28

OESpicoli::OEIsWritableSurface, 28

OESpicoli::OEMakeAccessibleSurface, 28

OESpicoli::OEMakeBitGridFromSurface, 29

OESpicoli::OEMakeBoxSurface, 29

OESpicoli::OEMakeCavitySurfaces, 29

OESpicoli::OEMakeCliqueSurface, 29

OESpicoli::OEMakeComplexCavities, 29

OESpicoli::OEMakeConnectedSurfaceCliques, 29

OESpicoli::OEMakeEllipsoidSurface, 30

OESpicoli::OEMakeGridFromSurface, 30

OESpicoli::OEMakeMolecularSurface, 30

OESpicoli::OEMakeSphericalSurface, 31

OESpicoli::OEMakeSurfaceFromGrid, 31

OESpicoli::OEMakeVoidVolume, 31

OESpicoli::OEReadSurface, 31

OESpicoli::OESetSurfaceColor, 32

OESpicoli::OESetSurfacePotentials, 32

OESpicoli::OESpicoliGetArch, 32

OESpicoli::OESpicoliGetLicensee, 32

OESpicoli::OESpicoliGetPlatform, 32

OESpicoli::OESpicoliGetRelease, 32

OESpicoli::OESpicoliGetSite, 33

OESpicoli::OESpicoliGetVersion, 33

OESpicoli::OESpicoliIsLicensed, 33

OESpicoli::OESurface, 13

Clear, 14

ClearVertexClique, 14

Constructors, 13

CreateCopy, 14

GetAtoms, 14

GetAtomsElement, 14

GetColor, 14

GetColorElement, 15

GetCurvature, 15

GetCurvatureElement, 15

GetDataType, 15

GetDistance, 15

GetDistanceElement, 15

GetFaceNormals, 15

GetFaceNormalsElement, 16

GetNormals, 16

GetNormalsElement, 16

GetNumTriangles, 16

GetNumVertices, 16

GetPotential, 16

GetPotentialElement, 17

GetPotentialName, 17

GetResolution, 17

GetTitle, 17

GetTriangles, 17

GetTrianglesElement, 17

GetVertexClique, 17

GetVertexCliqueElement, 18

GetVertices, 18

GetVerticesElement, 18

IsAtomsSet, 18

IsColorSet, 18

IsCurvatureSet, 18

IsDataType, 18

IsDistanceSet, 19

IsFaceNormalsSet, 19

IsNormalsSet, 19

IsPotentialSet, 19

IsVertexCliqueSet, 19

operator bool, 14

operator+=, 13

operator=, 13

SetAtoms, 19

SetAtomsElement, 19

SetColor, 20

SetColorElement, 20

SetCurvature, 20

SetCurvatureElement, 20

SetDistance, 20

SetDistanceElement, 20

SetFaceNormals, 21

SetFaceNormalsElement, 21

SetNormals, 21

SetNormalsElement, 21

SetNumTriangles, 21

SetNumVertices, 22

SetPotential, 22

SetPotentialElement, 22

SetPotentialName, 22

SetTitle, 23

SetTriangles, 23

SetTrianglesElement, 23

SetVertexClique, 23

SetVertexCliqueElement, 23

SetVertices, 23

SetVerticesElement, 23
 OESpicoli::OESurfaceArea, 33
 OESpicoli::OESurfaceCliqueArea, 33
 OESpicoli::OESurfaceCliqueCentroid, 34
 OESpicoli::OESurfaceCliqueVolume, 34
 OESpicoli::OESurfaceCropToClique, 34
 OESpicoli::OESurfaceFileType, 24
 OESpicoli::OESurfaceFileType::Grasp, 24
 OESpicoli::OESurfaceFileType::OESurface, 24
 OESpicoli::OESurfaceFileType::UNDEFINED, 24
 OESpicoli::OESurfaceIsOpen, 34
 OESpicoli::OESurfaceToMoleculeDistance, 34
 OESpicoli::OESurfaceVolume, 35
 OESpicoli::OEVoxelizeMethod, 24
 OESpicoli::OEVoxelizeMethod::Blank, 25
 OESpicoli::OEVoxelizeMethod::Blur, 25
 OESpicoli::OEVoxelizeMethod::Default, 25
 OESpicoli::OEVoxelizeMethod::Distance, 24
 OESpicoli::OEVoxelizeMethod::Gaussian, 24
 OESpicoli::OEWriteSurface, 35
 operator bool
 OESpicoli::OESurface, 14
 operator+=
 OESpicoli::OESurface, 13
 operator=
 OESpicoli::OESurface, 13

S

SetAtoms
 OESpicoli::OESurface, 19
 SetAtomsElement
 OESpicoli::OESurface, 19
 SetColor
 OESpicoli::OESurface, 20
 SetColorElement
 OESpicoli::OESurface, 20
 SetCurvature
 OESpicoli::OESurface, 20
 SetCurvatureElement
 OESpicoli::OESurface, 20
 SetDistance
 OESpicoli::OESurface, 20
 SetDistanceElement
 OESpicoli::OESurface, 20
 SetFaceNormals
 OESpicoli::OESurface, 21
 SetFaceNormalsElement
 OESpicoli::OESurface, 21
 SetNormals
 OESpicoli::OESurface, 21
 SetNormalsElement
 OESpicoli::OESurface, 21
 SetNumTriangles
 OESpicoli::OESurface, 21

SetNumVertices
 OESpicoli::OESurface, 22
 SetPotential
 OESpicoli::OESurface, 22
 SetPotentialElement
 OESpicoli::OESurface, 22
 SetPotentialName
 OESpicoli::OESurface, 22
 SetTitle
 OESpicoli::OESurface, 23
 SetTriangles
 OESpicoli::OESurface, 23
 SetTrianglesElement
 OESpicoli::OESurface, 23
 SetVertexClique
 OESpicoli::OESurface, 23
 SetVertexCliqueElement
 OESpicoli::OESurface, 23
 SetVertices
 OESpicoli::OESurface, 23
 SetVerticesElement
 OESpicoli::OESurface, 23