IMPLICIT SURFACES AND MODELING

MODELING AND ANIMATION, LAB 4

Mark Eric Dieckmann Erik Junholm Emma Broman Robin Skånberg

Wednesday 22nd March, 2023

Abstract

What makes implicit surfaces attractive in modeling and animation is their robustness. An implicit surface is guaranteed to be physically realisable and therefore known to work well in simulations. Implicit surfaces are also free of holes and self-intersections.

In this lab we implement a simple but yet effective modeling framework for implicit geometry. The framework contains operators for constructive solid geometry (CSG) such as union and intersection as well as their blending counterparts. We restrict our system (for the moment) to analytical surfaces such as quadrics and torii, and investigate analytical differential geometrical properties.

1 Implicit surfaces

As the name implies, the *implicit* surface representation can be seen as an indirect way of representing surfaces. In explicit surface representations, like meshes, the elements of the surface (triangles, edges and points for example), directly define the surface. But for the implicit or indirect surface representation, the surface is instead defined by an equation that needs to be solved in order to find (or approximate) the surface geometry. To show the difference between explicit and implicit representations we will look at the analytical circle with radius R. The explicit representation of this circle is

$$x = \pm\sqrt{R^2 - y^2}. \tag{1}$$

This equation will give you a value for x for every value of $y \in [-R, R]$ such that the points $[x, y]$ and $[-x, y]$ lie on the circle. The equation *explicitly* tells you where the circle is. Now let us take a look at the *implicit* representation of a circle. The implicit circle is described by the function:

$$f(x, y) = x^2 + y^2 \tag{2}$$

The function f assigns a scalar to every point in the x-y plane. In order to find our circle we need to look at a specific iso-value, C, in this scalar field. By choosing all points in the x-y plane where $f(x, y) = C$ we can cut out a subset of the entire x-y domain. This subset is called a *level-set* of the implicit function $f(x, y, R)$. Alternate names are *iso-surface*, *level-surface*, or the dimension-independent *interface* or *contour*. This level-set will be an object of co-dimension 1, i.e. the level set of a scalar function will have the dimension of the scalar function minus 1. A level-set of, for example, a 2D scalar function will be a 1D contour - a line. If, for the circle, we choose the constant $C = R^2$ we will find the level-set represented by the equation $f(x, y) = R^2$. The set of points $[x, y]$ that satisfies this equation is the implicit representation of a circle with the radius R. As an exercise you can convince yourself of this by deriving equation (1) from:

$$\begin{cases} f(x, y) = x^2 + y^2 \\ f(x, y) = C = R^2 \end{cases} \tag{3}$$

Usually we prefer to look at the zero level-set, i.e. $C = 0$, of the scalar function. By rearranging the terms and moving R^2 to the left in equation (2) we

arrive at:

$$\begin{cases} f(x,y) = x^2 + y^2 - R^2 \\ f(x,y) = C = 0 \end{cases} \tag{4}$$

The surfaces described by (3) and (4) are the same. You can convince yourself of this by deriving equation (1) from both (3) and (4). Note that the equation (2) actually contains information about every possible circle and by specifying an iso-value we obtain the specific circle with the radius $\sqrt{C}$.

In the simple case of a circle, it is easy to derive the explicit representation from the implicit and, as a result, the difference between the two may not be entirely clear. In more involved cases, however, the difference between these two representations will be more obvious. Consider for example the level-set:

$$\begin{cases} f(x,y,z) &= x^3 + y^2 + z + xy \\ & \quad + yz - sin^2(x + y + z) \\ f(x,y,z) &= C = 0 \end{cases} \tag{5}$$

Though the implicit representation is fairly compact and easy to read the explicit representation of this geometry is far from trivial.

1.1 Definition

Given a real valued function:

$$f(\mathbf{x}) \rightarrow \eta, \mathbf{x} \in \Re^n, \eta \in \Re \tag{6}$$

we can define the implicit surface, or level-set:

$$\mathcal{S}(C) \equiv \Big\{ \{\mathbf{x}\} : f(\mathbf{x}) = C \Big\} \tag{7}$$

as the collection of points for which the function $f(\mathbf{x})$ evaluates to C. To get a different level-set, we can change the iso-value, C.

1.2 Properties of implicit surfaces

Given an equation like (5), how do we find the implicit surface it describes? First of all it is important to note that we can classify every point in space as either inside, outside or on the implicit surface. Given a scalar function $f(\mathbf{x})$ and an iso-value C the point $\mathbf{x}$ is classified as follows:

$$\begin{array}{ll} \text{Inside:} & f(\mathbf{x}) < C \\ \text{Outside:} & f(\mathbf{x}) > C \\ \text{On surface:} & f(\mathbf{x}) = C \end{array} \tag{8}$$

If we look at equation (5) we can see that the point (1,1,1) is outside since $f(1,1,1) \approx 5$. The point (1,1,-2) is inside since $f(1,1,-2) = -1$ and the point (0,0,0) is on the surface since $f(0,0,0) = C = 0$.

The implicit surface can be seen as the co-dimension 1 *interface* that separates the scalar field into an inside and an outside region and thus the implicit surface is often referred to simply as *the interface*. By testing all points in space and classifying them according to (8) we can select the set of points that make up the interface.

The scalar value obtained from f can be useful for other purposes. For the implicit circle described in (3) the scalar value denotes the squared distance from that point to the closest point on the circle.

1.3 Differential properties

One of the strengths of implicit representations is the efficiency when deriving differential attributes of a surface. Two good examples of this are surface normals and mean curvature. Let's start by considering the surface normal. We can derive the expression for the normal of a 3D scalar field by looking at the gradient ∇ of the scalar field $f(x,y,z)$:

$$\nabla = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right] \equiv \vec{e}_x \frac{\partial}{\partial x} + \vec{e}_y \frac{\partial}{\partial y} + \vec{e}_z \frac{\partial}{\partial z} \tag{9}$$

Assuming that we can find a coordinate frame where one of the basis vectors points in the normal direction we can rewrite (9) as:

$$\begin{aligned} \nabla = \quad & \vec{e}_1 \tfrac{\partial}{\partial \vec{e}_1} + \vec{e}_2 \tfrac{\partial}{\partial \vec{e}_2} + \vec{n} \tfrac{\partial}{\partial \vec{n}} = \\ & \vec{e}_1 (\vec{e}_1 \cdot \nabla) + \vec{e}_2 (\vec{e}_2 \cdot \nabla) + \vec{n} (\vec{n} \cdot \nabla) \end{aligned} \tag{10}$$

This gives:

$$\nabla f = \vec{e}_1 (\vec{e}_1 \cdot \nabla f) + \vec{e}_2 (\vec{e}_2 \cdot \nabla f) + \vec{n} (\vec{n} \cdot \nabla f) \tag{11}$$

Assuming that our level-set is a manifold we know that an infinitesimally small patch of the surface will be flat. Since $\vec{n}$ is defined to be in the direction of the surface normal and the manifold is locally flat we know that $\vec{e}_1$ and $\vec{e}_2$ will both lie in the tangent plane of the surface and on the surface itself. Thus the scalar field f will be locally constant everywhere in the tangent plane and we get:

$$\begin{aligned} \frac{\partial f}{\partial \vec{e}_1} = \frac{\partial f}{\partial \vec{e}_2} &= 0 \\ &\Rightarrow \quad \nabla f = \vec{n}(\vec{n} \cdot \nabla f) \\ &\Rightarrow \quad \vec{n} = \pm \frac{\nabla f}{|\nabla f|} \end{aligned} \tag{12}$$

which reduces to

$$\vec{n} = \frac{\nabla f}{|\nabla f|} \tag{13}$$

when using the sign convention in equation (8). We now see that as long as the iso-surfaces of a scalar field f are manifolds the normalized gradient of a scalar field is the surface normal to the manifold that intersects that point.

Since we now know that the surface normal and the gradient are parallel we can use this to derive an expression for the mean curvature. We arrive at an expression for the mean curvature of a 2D surface embedded in 3D:

$$\kappa = \frac{1}{2} \nabla \cdot \vec{n} = \frac{1}{2} \left(\frac{\partial n_1}{\partial x} + \frac{\partial n_2}{\partial y} + \frac{\partial n_3}{\partial z} \right) \tag{14}$$

where $\vec{n} = (n_1, n_2, n_3)$. The curvature is given as the *divergence* of the normal at a point. Geometrically, if the normals in a neighborhood around a point is the same, the curvature is zero (the surface is flat). On the other hand, if the normals are very different, pointing in very different directions, the curvature is large (the surface is "spiky").

1.4 Implementing implicit functions

Implementing basic implicit geometry is fairly easy. Since the geometry is based on evaluating functions we define an interface that our different classes must adhere to. In the very simplest case a `GetValue` function suffices.

```
class Implicit {
public:
  virtual float
  GetValue(float x, float y, float z);
}
```

Now we can just inherit from `Implicit` and define our own implicit geometry functions.

Since the x, y and z in `getValue` are world space coordinates we need to transform them into object space before we can use them. If the matrix $\mathbf{M}$ describes the object-to-world transform we can find the object space coordinates $\mathbf{p}'$ by the relation $\mathbf{p}' = \mathbf{M}^{-1}\mathbf{p}$ where $\mathbf{p}$ is the vector $(x, y, z, 1)$

Once we have the object space coordinates we simply evaluate the implicit function $f(x, y, z)$ at these coordinates. The pseudo code for the `GetValue` function for an implicit sphere with radius R will look like this:

```
float GetValue(float x, float y, float z)
{
  Vector4 pWorld(x,y,z,1);
  Matrix4x4 M = getObjectTransform();

  Vector4 pObject = M.inverse() * pWorld;

  float val = pObject.x()*pObject.x()
            + pObject.y()*pObject.y()
            + pObject.z()*pObject.z()
            - R*R;

  return val;
}
```

With the above function we can evaluate the implicit sphere function for every point in space taking into account the object transformation. But what about normals and curvature? Both require calculating differentials on the implicit surface. We could use the definition of the differential operator:

$$\frac{\partial f}{\partial x} \equiv \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \tag{15}$$

However, due to the limited precision provided by the computer's floating point numbers we cannot evaluate equation (15) to arbitrary precision, because we cannot allow h to be arbitrarily small.

However if we choose a sufficiently small value ϵ for h we can approximate equation (15). We arrive at the discrete differential D along x:

$$D_x \equiv \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \approx \frac{f(x_0+\epsilon)-f(x_0)}{\epsilon} \quad (16)$$

We can now calculate gradients using this discretized differential operator which in turn allows us to evaluate both normals and curvature at any point in the scalar field described by $f(x,y,z)$.

Equation (16) is asymmetric and more weight is placed on the values in the positive x direction, which may be good or bad. We can get rid of this asymmetry by using the central difference approximation instead:

$$D_x \equiv \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \approx \quad (17)$$
$$\approx \frac{f(x_0+\epsilon)-f(x_0-\epsilon)}{2\epsilon}$$

Regarding the curvature, there are many ways to discretize equation (14) (see [**?**]) with the easiest one being to directly compute the partial derivatives using finite differences. However, there are simpler approximations, such as:

$$\kappa \approx \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \quad (18)$$

where the second partial derivatives can be computed as

$$\frac{\partial^2 f}{\partial x^2} = D_{xx} \approx \frac{f(x_0+\epsilon)-2f(x_0)+f(x_0-\epsilon)}{\epsilon^2} \quad (19)$$

What assumption is being posed on f for this approximation to be reasonable? Hint: Relate this expression to equations (14) and (12).

2 Quadrics

A quadric surface is simply defined by a quadratic implicit function. In general, a quadratic function in three variables can be written:

$$\begin{aligned} f(x,y,z) &= Ax^2 + 2Bxy + 2Cxz \quad (20)\\ &+ 2Dx + Ey^2 + 2Fyz \\ &+ 2Gy + Hz^2 + 2Iz \\ &+ J \end{aligned}$$

or more succinctly on matrix form:

$$\mathbf{p}^T\mathbf{Q}\mathbf{p} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (21)$$

2.1 Normals

The normal for an implicit surface is defined in equation 12 above. Since a quadric surface is known analytically, we can apply the differentiation to the quadric with the coefficient matrix $\mathbf{Q}$ directly. This gives a nice analytic expression involving only the same coefficients, $Q_{i,j}$:

$$\begin{aligned} \nabla f(x,y,z) &= 2 \begin{bmatrix} Ax+By+Cz+D \\ Bx+Ey+Fz+G \\ Cx+Fy+Hz+I \end{bmatrix} \quad (22)\\ &= 2 \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ &= 2\mathbf{Q}_{sub}\mathbf{p} \end{aligned}$$

2.2 Affine transformations

Another nice property of quadrics is that they allow you to evaluate the function, $f(x,y,z)$, for a transformed point, $\mathbf{p}'$, by transforming the coefficients of $\mathbf{Q}$. Thus we can transform the quadric matrix to world space *once* and evaluate the function for global coordinates instead of having to transform every evaluation point, which is much more costly, especially if done frequently. Given a quadric the matrix evaluation is:

$$f(x,y,z) = f(\mathbf{p}) = \mathbf{p}^T\mathbf{Q}\mathbf{p}$$

The affine transformation $\mathbf{M}$ between the original point $\mathbf{p}$ and the transformed point $\mathbf{p}'$ is:

$$\mathbf{p}' = \mathbf{M}\mathbf{p} \quad \Leftrightarrow \quad \mathbf{p} = \mathbf{M}^{-1}\mathbf{p}'$$

Now we can write the quadric matrix expression as a function of transformed points:

$$\begin{aligned} f(\mathbf{p}) &= f(\mathbf{M}^{-1}\mathbf{p}') \\ &= \left(\mathbf{M}^{-1}\mathbf{p}'\right)^T \mathbf{Q}(\mathbf{M}^{-1}\mathbf{p}') \\ &= (\mathbf{p}')^T \left(\mathbf{M}^{-1}\right)^T \mathbf{Q}(\mathbf{M}^{-1})\mathbf{p}' \end{aligned}$$

And we conclude:

$$\begin{aligned} f(\mathbf{p}) &= (\mathbf{p}')^T \mathbf{Q}'\mathbf{p}', \\ &\text{where} \quad \mathbf{Q}' = \left(\mathbf{M}^{-1}\right)^T \mathbf{Q}\mathbf{M}^{-1} \end{aligned}$$

2.3 Spheres, planes, cylinders and what-not all-in-one

Equation 16 describes a family of surfaces; actually a total of 17 standard-form types. They include planes, cylinders, spheres and ellipsoids, cones, paraboloids and hyperboloids, see figure 1. Below we have listed the most common quadrics along with their analytical expressions. Only the unique cases are listed, rotational and axial symmetry provide the complementary quadrics.

- Planes

$$f(x,y,z) = ax + by + cz = 0 \tag{23}$$

- Cylinders

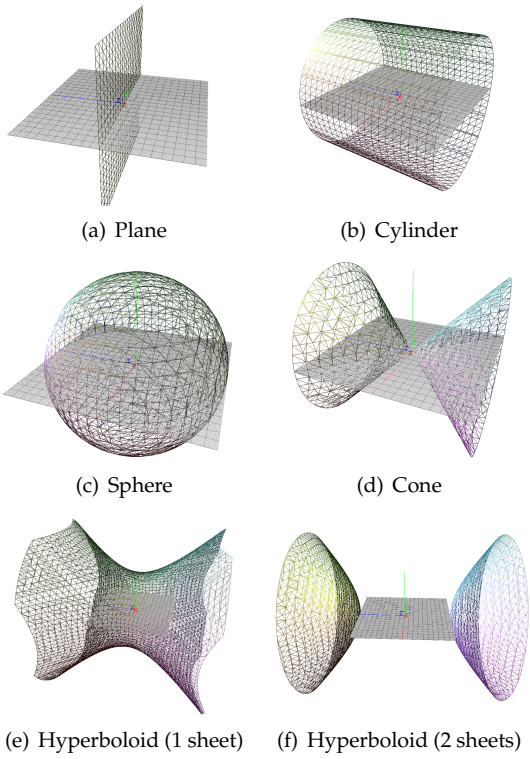
$$f(x,y,z) = x^2 + y^2 - 1 = 0 \tag{24}$$

- Spheres and ellipsoids

$$\begin{aligned} f(x,y,z) &= x^2 + y^2 + z^2 - 1 = 0 \\ f(x,y,z) &= \tfrac{x^2}{a^2} + \tfrac{y^2}{b^2} + \tfrac{z^2}{c^2} - 1 = 0 \end{aligned} \tag{25}$$

- Cones

$$f(x,y,z) = x^2 + y^2 - z^2 = 0 \tag{26}$$



(a) Plane (b) Cylinder (c) Sphere (d) Cone (e) Hyperboloid (1 sheet) (f) Hyperboloid (2 sheets)

Figure 1: Quadric surfaces. Triangulated with marching cubes in a bounding box.

- Paraboloids

$$f(x,y,z) = x^2 \pm y^2 - z = 0 \tag{27}$$

- Hyperboloids

$$f(x,y,z) = x^2 + y^2 - z^2 \pm 1 = 0 \tag{28}$$

3 Constructive solid geometry

One benefit of implicit surfaces is that you can efficiently use a very intuitive modeling concept, called *constructive solid geometry* (CSG). This technique is based on the idea that you create complex objects and shapes by combining simpler objects (primitives), such as spheres and cubes. It is very much

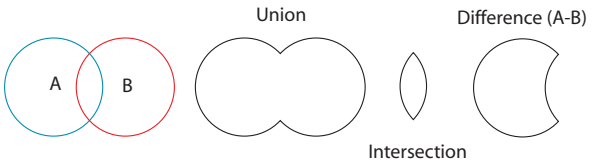


Figure 2: Boolean operations

like building with LEGO[1], where the primitives are blocks of different shapes and sizes. CSG objects are constructed from these primitive shapes and a set of boolean operations, such as union, intersection and difference.

3.1 Boolean operations

Performing boolean operations on two implicit surfaces is very simple. Given the primitives A and B in Figure 2, the union, intersection and difference operations are computed as follows:

$$\textbf{Union}(A,B) = A \cup B = \min(A,B) \tag{29a}$$
$$\textbf{Intersection}(A,B) = A \cap B = \max(A,B) \tag{29b}$$
$$\textbf{Difference}(A,B) = A - B = \max(A,-B) \tag{29c}$$

3.2 Blending operations

A problem with these simple operations is that the interface will only be C^0 continuous along the intersection. This means that the normal will not be defined at these points, so we will have very sharp and ill-defined creases in our otherwise smooth models. A way around this problem is to approximate the boolean operations so we achieve smooth transitions between the intersection.

3.2.1 Density functions

One way to achieve smooth transitions is to use *density functions*. Roughly speaking, these functions describe the "density" D_A of an implicit geometry A in the following manner:

$$D_A(\mathbf{x}) = \begin{cases} > 1 & \text{if } \mathbf{x} \text{ is inside the surface} \\ = 1 & \text{if } \mathbf{x} \text{ is on the surface} \\ \in [0,1) & \text{if } \mathbf{x} \text{ is outside the surface} \end{cases} \tag{30}$$

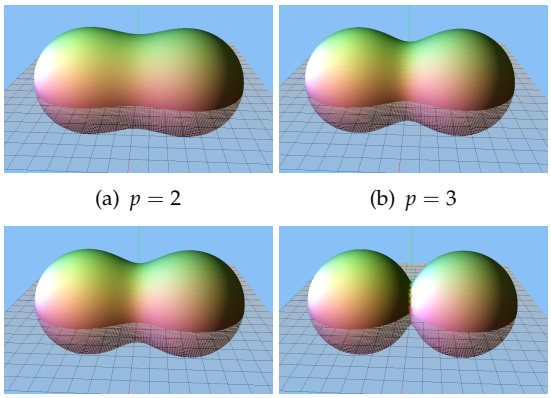
In order to transform an implicit surface A into a density function, the following transformation can be applied (assuming a negative-inside sign convention):

$$D_A(\mathbf{x}) = e^{-A(\mathbf{x})} \tag{31}$$

We can perform a *super-elliptic blending* by:

$$D_{A\cup B} = \left(D_A^p + D_B^p\right)^{1/p} \tag{32a}$$
$$D_{A\cap B} = \left(D_A^{-p} + D_B^{-p}\right)^{-1/p} \tag{32b}$$

Again, difference is obtained by negating either A or B and performing an intersection. Interestingly, as $p \to \infty$, we obtain the same result as in Equation 29. Results as p varies are shown in Figure 3. How do you transform the density function back to an implicit function (i.e. inverse of equation (31)) ?



(a) $p = 2$ (b) $p = 3$

(c) $p = 7$ (d) $p = 20$

Figure 3: Super-elliptic blending as p varies.

4 A simple CSG framework

Putting all of this together, a simple CSG framework can be created. We have defined our primi-

[1]The name LEGO comes from the Danish "leg godt", which translates to "play well".

tives as spheres, planes, cylinders or quadrics. In the previous section we explained how to combine two primitives by boolean operations, and how to achieve smooth intersections through super-elliptic blending. Thus, by combining several simple primitives in a hierarchical manner, we can model fairly complex structures without losing the benefits of the implicit representation. In practice, this can be achieved by building a tree structure where the nodes represent boolean operations acting on two leaf nodes. We call the nodes *CSG operators* with the structure presented in Listing 1. Note that the `CSG_Operator` inherits from `Implicit`. Thus, it can be treated as an implicit surface and inherits the required `GetValue`-function as discussed in Section 1.4. Also note the leaf pointers `left` and `right` which can be pointed to any implicit primitive as well as other `CSG_Operators` giving us a recursive tree structure. An example operator `Union` is also given in Listing 1, where the `GetValue`-function is to be implemented to compute the basic C^0 continuous union operation (Equation 29).

Listing 1: CSG operation base class.

```
class CSG_Operator : public Implicit {
protected:
  Implicit *left, *right;
};

class Union : public CSG_Operator {
public:
  float
  GetValue(float x, float y, float z) {
    // Compute the value based on
    // left and right's values in
    // position (x, y, z)
    return 0.f;
  }
};
```

5 Assignments

5.1 Grading

The assignments marked with (3) are mandatory to pass the lab, grade 3. A grade 4 is awarded for the completion of assignment (4) and a grade 5 is awarded for completing assignments (4,5a) or (4,5b).

5.2 Assignments

In this lab you will experiment with implicit surfaces and CSG operators. The classes involved are `Implicit`, `CSG`, `Sphere` and `Quadric`. Study these files and some example code in e.g. `FrameMain::AddObjectImplicitSphere()` to get a feel for how they are interconnected.

Implement CSG operators (3)

The first assignment is to implement the basic boolean operators union, intersection and difference. A skeleton for this is prepared in `CSG.h`. Implement the `GetValue`-functions in classes `Union`, `Intersection` and `Difference` and verify that they work by combining a set of implicit geometries. Use the CSG panel in the GUI to execute the operations. Select multiple objects by clicking while pressing shift. (Note: the bounding boxes created in the constructor are used when triangulating the surface, see `Implicit::Triangulate()`). How does the resulting mesh depend on the sampling distance? Experiment with this parameter by modifying the "Mesh sampling" parameter in the GUI.

Recall that the implicit surface is represented by a scalar function $f(x,y,z)$. The surface $\mathcal{S}$ being visualized is the zero level set of this function, $\mathcal{S} = \{(x,y,z) : f(x,y,z) = 0\}$. In order to visualize the function f directly, you can use a "Scalar cut plane" by selecting the implicit object and adding the cut plane through the "Add object" menu. It can be convenient to lower the opacity of the implicit surface to see the interior.

A recent addition to the lab code is an object called "Implicit mesh" which loads a mesh and computes the closest distance to the mesh within the bounding box. In this way, the mesh representation is transformed into an implicit geometry where the surface is the zero level set. By using this, you could perform CSG operations between mesh objects and implicits. However, the code is a recent addition and is not completely stable - load the cow.obj to see

the problems. If you have time and interest, feel free to search for the problem.

Implement the quadric surface (3)

`Quadric.h` declares a quadric surface that inherits from `Implicit`. The quadric is fully determined by its 4x4 matrix of coefficients. Implement `GetValue()` and `GetGradient()` using equations (21) and (22). The quadric inherits a bounding box from `Implicit` which initializes to zero. Unlike other implicit surfaces (e.g. `Sphere`) the bounding box cannot be determined from the quadric coefficients, so you need to set it manually using `Implicit::SetBoundingBox()`. Create the quadric matrices in `FrameMain::AddObjectQuadric*()` by using the equations in section 2.3 (see equations (20) and (21) to identify the matrix components). To visualize and verify your gradients you can use the "Gradients" visualization mode and check the "Visualize mesh normals" box to compare with the mesh normals.

Implement the discrete gradient operator for implicits (4)

In the file `Implicit.h` you have three "Get" functions: `GetValue()` should always be overloaded for the specific surface, but `GetGradient()` and `GetCurvature()` can be generally computed as discrete differentials for all types of implicits. Implement `GetGradient()` using equation (17). Note that you have to apply the equation three times to produce the vector, once in each coordinate direction. When implemented, you can use the "Gradients" visualization mode to draw the gradient vectors at the surface, or a "Vector cut plane" to draw gradient vectors in the implicit function. The ϵ parameter in equation (17) can be changed in the GUI through the "Differential scale" slider. Experiment with this setting to see how it affects the computation of gradients. Provide a discussion regarding this in the report.

Implement the discrete curvature operator for implicits (5a)

Implement `GetCurvature()` in `Implicit.cpp` using the approximation in equation (18). Then, the curvature can be visualized with the "Curvature" visualization mode. It can be convenient to use the Jet or HSV colormaps[2] and to set manual min/max ranges for the colormap instead of automatic. Then you can interpret the colors to see whether the curvature gives a good correspondence to convexity or concavity (positive and negative curvatures respectively). Like in the previous assignment, experiment with the ϵ parameter to see how it affects the computation, and write your conclusions in the report.

Implement super-elliptic blending (5b)

For this task you should implement the boolean operations using super-elliptic blending. Finalize classes `BlendedUnion`, `BlendedIntersection` and `BlendedDifference` to `CSG.h` and implement the ideas presented in Section 3.2.1. Study how the results vary as the blending parameter p changes.

6 Acknowledgements

The lab scripts and the labs were originally developed by Gunnar Läthén, Ola Nilsson, Andreas Söderström and Stefan Lindholm.

[2] `http://www.mathworks.com/help/techdoc/ref/colormap.html`