Geometric Implicit Decals

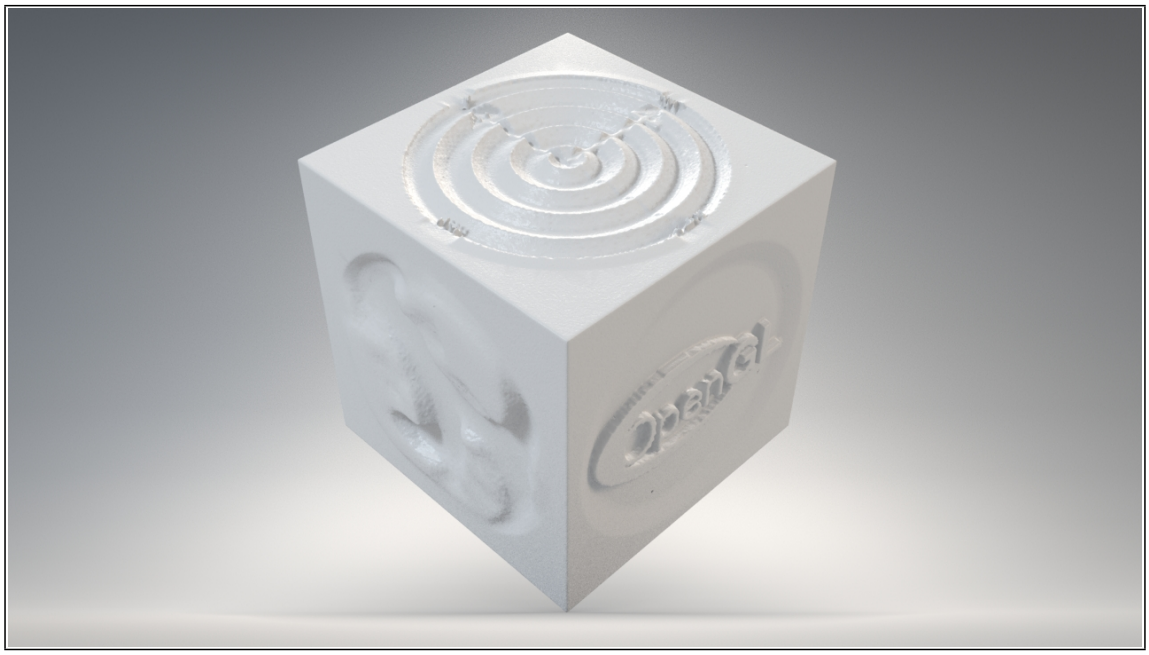
Adding small scale features to implicit surfaces using implicit decals

Wojtek Zankowski

Master in Sciences

Computer Animation and Visual Effects

August, 2014



Abstract

Adding detail to implicit surfaces is difficult due to a lack of surface parameterisation, which significantly limits their current applications. This thesis proposes a recent technique of adding small scale features to implicit surfaces through calculating a surface's local parameterisation using implicit decals - particle like objects scattered on the surface. The parameterisation is used to perform a geometric operation on the implicit surface, effectively refining it locally. The research is focused on applying details with use of texture maps, however different methods of local modification are also discussed.

Keywords: implicit decals, implicit surfaces, modeling with textures, implicit texturing

Contents

Chapter 1

Introduction

1.1 Implicit surfaces

In majority of nowadays computer 3D modelling, geometrical objects are expressed in a form of boundary representation, such as polygonal and triangular meshes or NURBS surfaces. However, these are not optimal and the most efficient when trying to represent soft and organic objects.

A good solution to express such entities is to use volumetric representation, such as isosurfaces running through a scalar field defined by some field function (also known as scalar field function, space function or defining function), a method often referred to as *implicit surfaces*.

In this technique, geometry is represented by a field function $f(x)$ and can be visualised by using so called *polygonisation*. It is a process which constructs a mesh that approximates the isosurface of the scalar field defined by $f(x)$. Therefore different field functions result in different polygonal meshes.

Furthermore, a number of field functions can be combined together using a geometric operation (union, intersection, etc.).

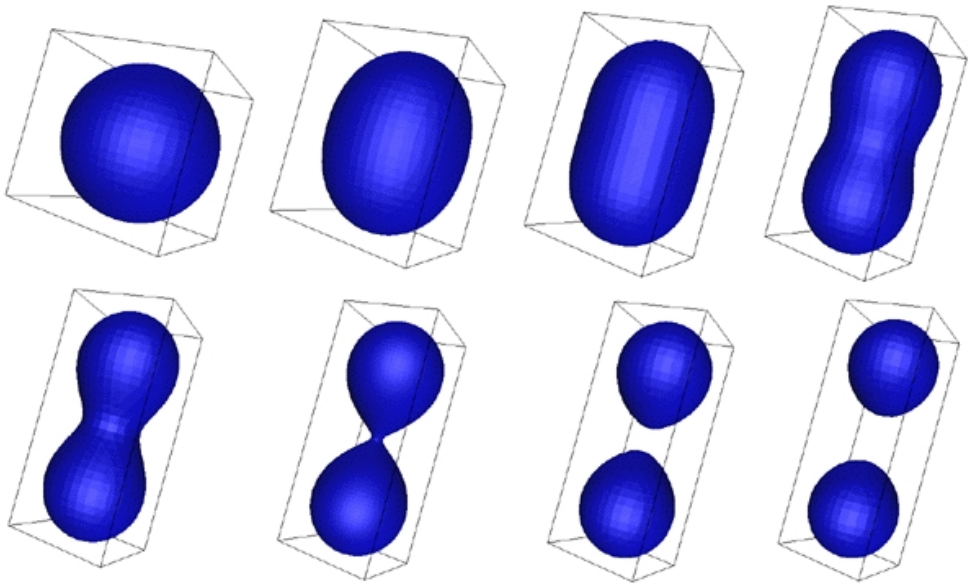


Figure 1.1: *Implicit surface created from two sphere functions. Bourke*

1.2 Why is it difficult to modify them locally

The field function $f(x)$ used in implicit surfaces is in a form which is difficult for user to visualise and interact with, ie. its interface is unnatural, the workflow‘s turnaround is slow. Only after the isosurface is polygonised and displayed, user may make quick, intuitive decisions about its look. Still, putting these decisions in action by modifying the field function is yet not quite intuitive and the workflow does not provide a quick feedback

Consider an example:

- An implicit surface P_s is defined by the field function

$$P_{f(x,y,z)} = y \tag{1.1}$$

  which constructs a plane

- After visualising the P_s through polygonisation, user decides to modify the surface around one of its corners.

- To modify an implicit surface one has to change its field function

$f(x)$. Therefore the simple $f(x, y, z) = y$ has to be exchanged with much more complex function that would identify the coordinates around the plane's corner and modify them according to user's expectations.

- After modifying the field function $f(x)$, the program needs to repolygonise the scalar field.
- If the modification turns out to be incorrect or unsatisfying, the whole process needs to be repeated.

This workflow contains some major drawbacks:

1. The process is very difficult and involving since it requires extraordinary spatial intelligence, as well as great understanding of geometry.
2. Without having appropriate programing knowledge, user is unable to add small scale modifications to the implicit surface.
3. It proves to be very slow, as each time a modelling decision is made the field function needs to be modified manually and then repolygonised.

1.3 How implicit decals could serve as a solution

A way of overcoming the problems outlined in the last paragraph could be use of *implicit decals*. These, as first described by Pedersen, later enhanced by Schmidt *et al.* and de Groot *et al.*, are particle-like objects defined on, or near the surface of an underlying geometry. Implicit decals, as described by de Groot are used to calculate a local parameterisation (or more specifically texture coordinates) using a spherical field function, centered on decal's position, therefore avoiding computationally-expensive global parameterisation proposed by Schmidt *et al.*. Instead, Euclidian distance between the surface point q and the decal is used to calculate the parameterisation. This method is quick enough to execute in real time,

allowing intuitive interaction with the decaling interface in the viewport, even at high quantities of decals.

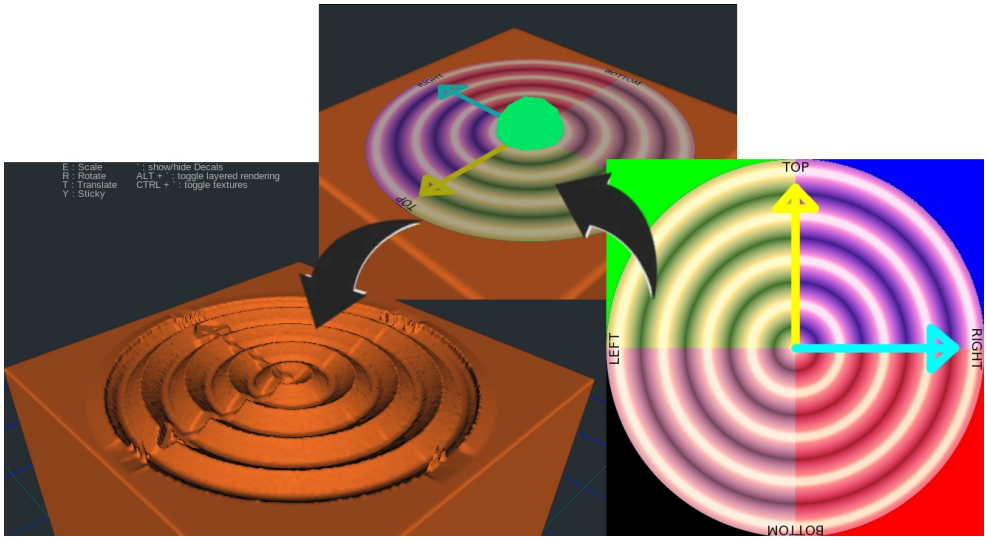


Figure 1.2: *Using decals to add small scale features*

The principle of calculating local parameterisation proposed by de Groot *et al.* can be employed to modify the result of an implicit surface's field function, without needing to change the function itself. There is a number of ways in which a decal could affect the field function's output, dependant on required implementation. This project present an interface which combines the idea of implicit decals with marching cubes algorithm used to polygonise implicit surfaces. Resulting tool allows to modify an implicit surface in a similar way an implicit decal - as described by de Groot - would modify a texture of a surface, as if it was embedding the textures onto the surface. The main advantage of the proposed system is the fact that decals are fast and intuitive to create and modify, which resolves the issues of slow turnaround and workflow's difficulty and unnaturality present in traditional implicit surfaces.

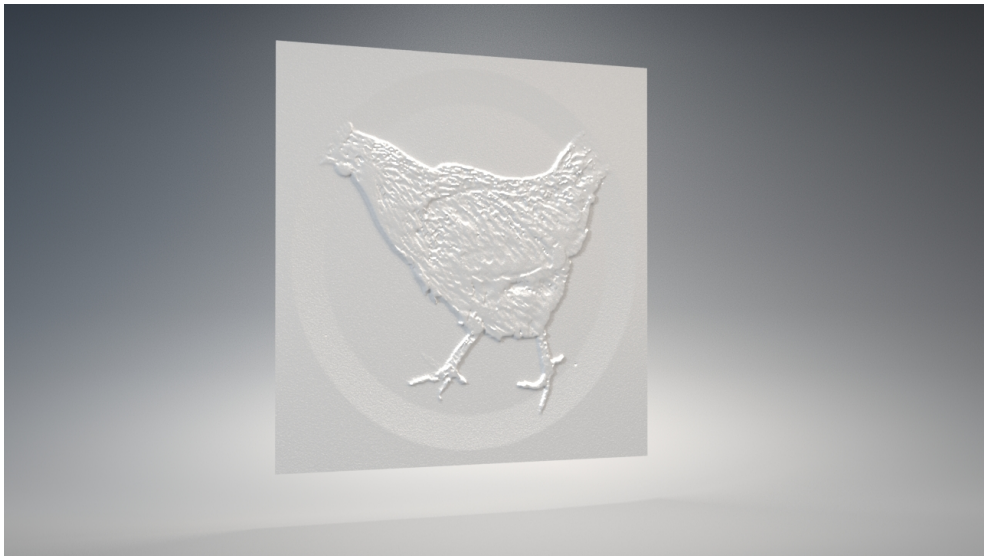


Figure 1.3: *Implicit decal technique used with polygonisation to imprint the texture onto a planar implicit surface*

This thesis is constructed in the following way. Having outlined other relevant approaches of using implicit decals in (2), it describes in detail the idea of a decal (3.1), of texturing with decals (3.2) and the marching cubes algorithm (3.3). Afterwards, the concept of combining that algorithm with implicit decals to modify an implicit surface along with a detailed implementation of geometric implicit decals is presented in chapter (4). Later it describes the underlying interface of the program (5), along with the project‘s history (5.3). Finally, chapter (6) contains results, tests and evaluation, followed by a conclusion drawn in chapter (7). Note that symbols and class names appearing in this thesis are outlined and briefly described in the Appendix A. Also, some examples refer to parts of a video that should be submitted along with this document.

Chapter 2

Related Work

2.1 Implicit decals

Concept of texturing a surface through local parameterisation is a subject that has been researched in various forms over the past 20 years. Pedersen was one of the first to develop a technique that gave an ability to define *patchinos* (patches) on an implicit surface, which then could be used to derive a local parameterisation needed to apply a texture onto the surface. These effectively were one of first definitions of implicit decals, described as independent objects bound to be positioned on the underlying surface.

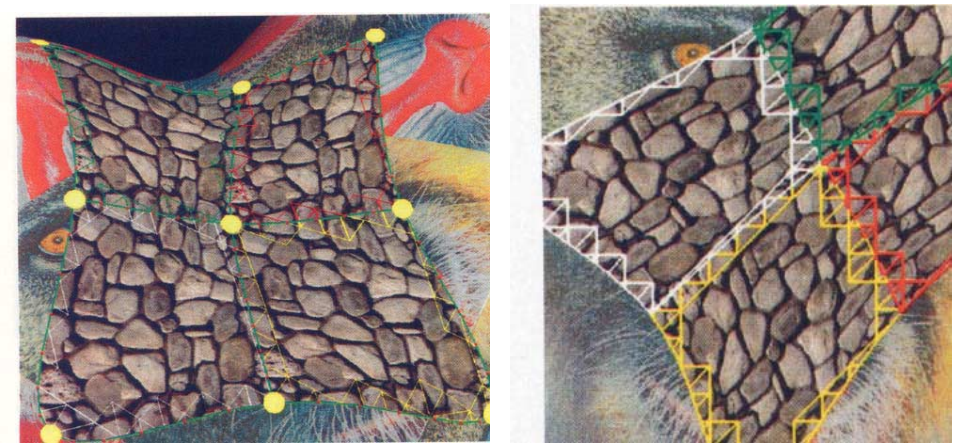


Figure 2.1: *Pedersen's patchinos. Pedersen*

A more advanced technique was developed in 2006 by Schmidt *et al.* as an improvement of Pedersen's decaling interface. In this approach, a local parameterisation in form of an exponential map is calculated from a decal's centre and a geodesic radius, requiring the underlying surface to be a continuous mesh. Nevertheless being computationally expensive and difficult to implement, the method provided a good way for texturing animated surfaces, as the parameterisation was preserved under changes of the underlying geometry. Furthermore, Schmidt's proposed a simple compositing tool allowing to combine and reuse decal textures in form of patches.

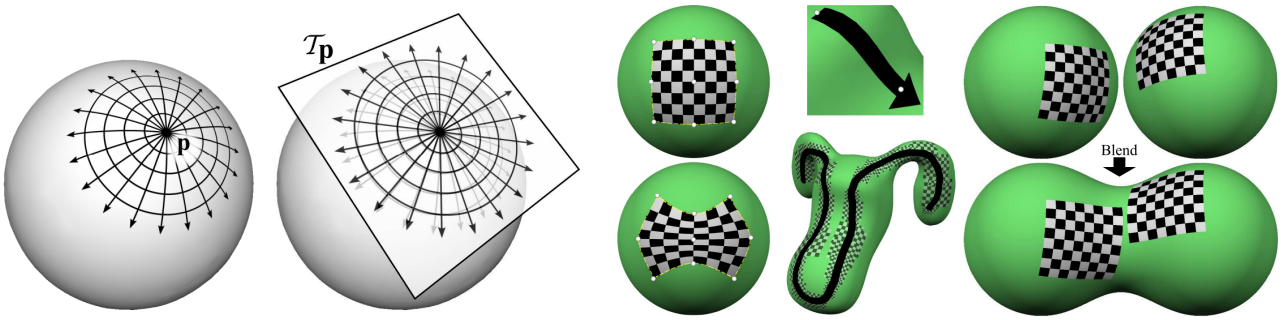


Figure 2.2: *Schmidt's exponential map based decals. Schmidt* et al.

Another approach, similar to Schmidt's work, that geometric implicit decals are basing on was researched in 2010 by de Groot *et al.* and published in a paper named Implicit Decals in 2013. In his work, de Groot describes implicit decals as a similar tool to exponential maps of Schmidt, yet relying the parameterisation on Euclidian distance and spherical field functions instead of geodesic calculations. Resulting interface is a much faster tool, however constrained by the size of the textures as it assumes to be used in large quantities on a relatively smooth and continuous surface. Nevertheless, it proves to be an effective and quick implicit texturing method, which is employed to modify polygonisation of an implicit surface, a technique proposed in this thesis. Furthermore, de Groots decals can be distorted as their parameterisation functions allow for field deformation and implicit composition operators. His work also implements a particle scattering system, allowing to evenly generate decals on the underlying surface, simplifying the process of decal

placement.

de Groot sums the advantages of implicit decals as:

- *The very fast computation of local parameterizations based on the Euclidean distance over a model. This local parametrization can be computed at arbitrary resolution and is independent of the underlying geometric representation.*
- *The technique is simple enough to implement in a pixel shader, without modifying the graphics pipeline nor limiting the use of other shaders, and it allows thousands of decals to be placed and edited interactively.*
- *Our decals can compete for space and deform when they interact with nearby decals. Surface connectivity is not required thus a decal can be placed across multiple objects or across gaps in an object without changing the object representation.*

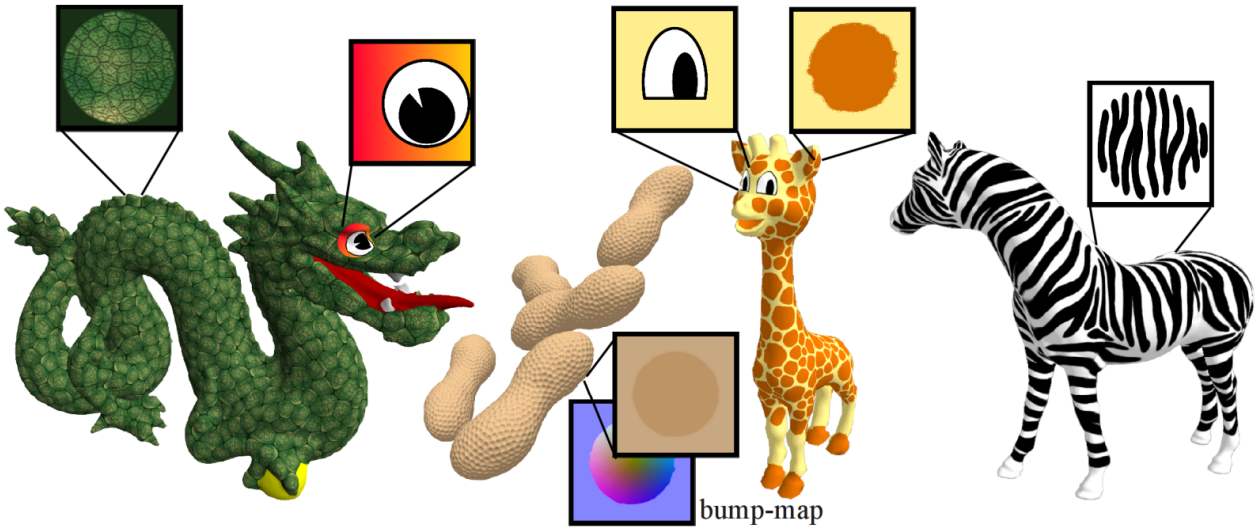


Figure 2.3: *de Groot's implicit decals used in high quantities to add small small textures onto the geometry. de Groot* et al.

2.2 Implicit surfaces

In 1994 Bourke published *Polygonising a scalar field*, a document describing an algorithm for polygonising an implicit surface named *marching cubes*. Author accurately sums the method as follows:

The fundamental problem is to form a facet approximation to an isosurface through a scalar field sampled on a rectangular 3D grid. Given one grid cell defined by its vertices and scalar values at each vertex, it is necessary to create planar facets that best represent the isosurface through that grid cell.

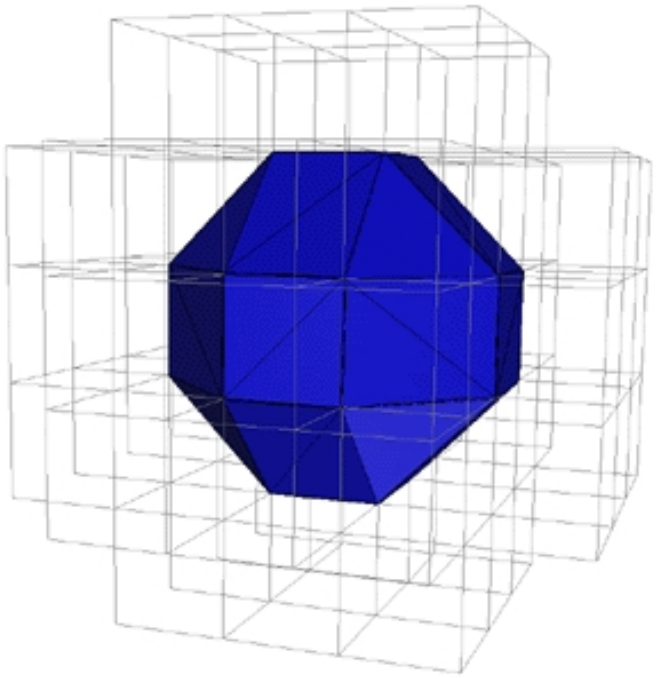


Figure 2.4: *Visualisation of the marching cubes algorithm, illustrating the 3D cell grid. Bourke*

This was executed through usage of a lookup table which stored all possible cases in which an isosurface could pass through a cell. In more detail, marching cubes are explained in section 3.3.

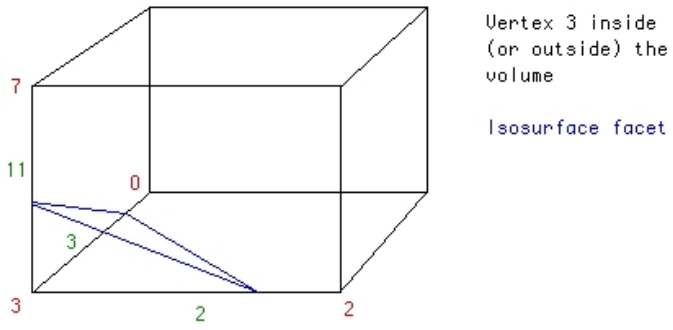


Figure 2.5: *Illustration of a the principle behind marching cubes. In this example cell corner number 3 was found on the opposite side of the isosurface then the rest of the corners, hence a triangle was created. Bourke*

This algorithm have been improved and continued in various forms over the last two decades, however in its original form is still one of the quickest methods of polygonising an indiscrete scalar field. Yet, it is not effective in creating sharp features and is not adaptive, which results in a relatively large topological density of resultant meshes. A number of solutions is proposed by different algorithms such as dual contouring, marching squares, or marching triangles, however all prove to be slower and not as robust as marching cubes. Since geometric implicit decal aim to approach realtime user interface, marching cubes was selected as the polygoniser. Nevertheless, the concept described in this work can be used with other similar polygonising algorithms.

2.3 Other work

Topic of adding small scale features to implicit surfaces has been researched by Zanni *et al.*, who in his publication from 2012 proposes a modeling technique able to enhance implicit surfaces with procedural geometric detail. Considered as one of the first research publications in the subject, Zanni‘s method uses Gabor noise, automatically aligned with respect to the orientation of the underlying geometry. It allows to add distributed anisotropic detail over implicit surfaces, providing an intuitive control

over their transformation and enables blending of the small scale features without blurring them.

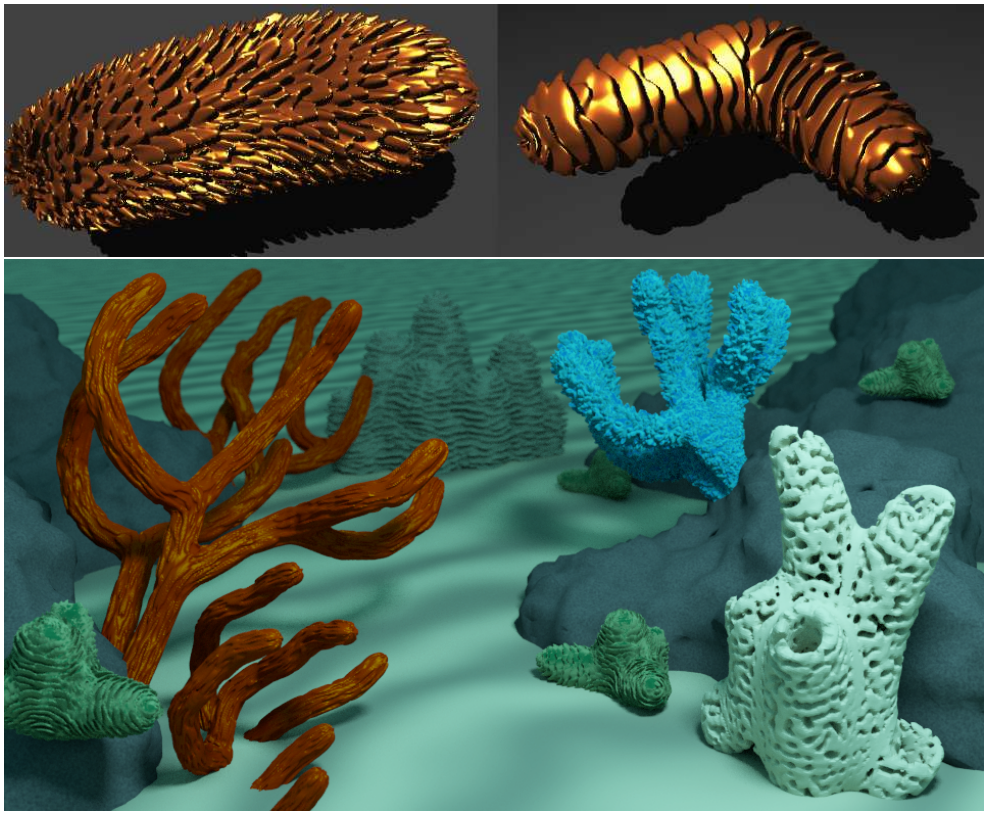


Figure 2.6: *Zanni's implicit surfaces with details, one of the currently available solutions of adding small scale features. Zanni* et al.

Interestingly, in his work he states:

Although many methods are available to add detail to mesh-based surfaces (...), methods to add detail (or texture) to implicit surfaces are scarce due to the difficulty to add coherent surface detail when no parameterization is available.

Ability of implicit decals to quickly derive local parameterisation can be used to overcome this problem, hence implementing them into an algorithm of isosurface polygonisation presents a technique of adding small scale details to implicit surfaces.

Chapter 3

Background

3.1 Decal

Word decal is short for Decalcomania or Decalcomanie. It is a plastic, cloth, paper or ceramic substrate that has printed on it a pattern or image that can be moved to another surface upon contact, usually with the aid of heat or water.

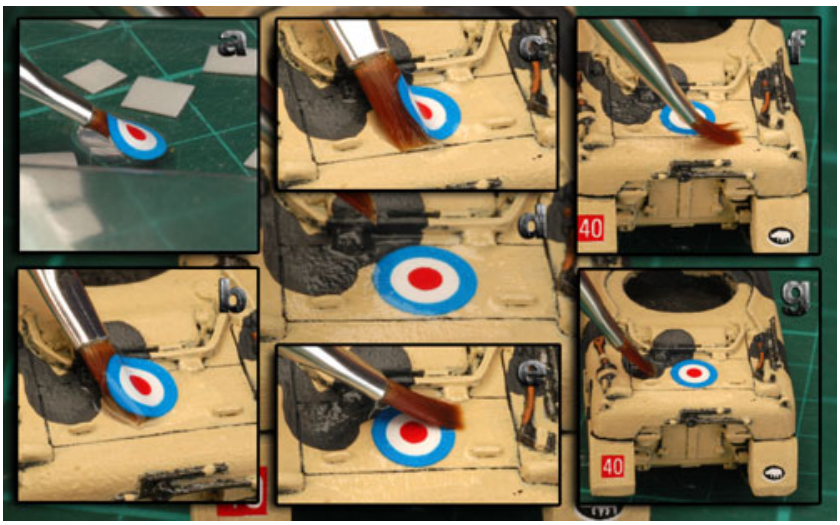


Figure 3.1: *Real decal application. Turner*

In real life decals are solely made for texturing, however this not necessarily must be true in computer graphics, as it will be discussed later in section 4.1.2. The concept of a decal translates to computer graphics as follows.

Each decal:

- Is relevant only in relation to an object it is parented to, hence its position, orientation and scale in relation to the parent is crucial.
- Has position attribute D_p that can be defined in a number of ways. One of the simplest ways of expressing decal's location would be Cartesian coordinates, relative to its parent or absolute (ie. world position). Still, it may as well be a set of any other values, such as polar coordinates or some parent-related embedded attributes.
- Has a way of modifying the parent in some way. This is solely implementation dependant. For instance, a decal could modify the geometry, texture, shader properties, or density of its parent.

Furthermore, in most cases each decal:

- Has a local frame of reference made of three axes - DX, DY and DZ, which is usually orthonormal.
- Has a radius D_r or a scale.
- Is positioned on the surface of its parent, as if it was sliding on it.

Considering this project, parent object P is an implicit surface, but in different applications it could describe any other geometrical object.

A rough real-life analogy of a decal would be a round fridge magnet. It is possible to move it around the fridges surface, rotate it or put it away. Even though not permanently, it changes the way the fridge looks in small proximity of magnets center. In this simple example the magnet is the decal D, its orientation is its frame of reference, while the fridge is decals parent P.

3.2 Texturing decal

3.2.1 Outline

To follow up with the real life purpose of decals, one of their applications in computer graphics is defining the texture of the parent surface P_s. There is a number of approaches to this topic, such as ones presented by Pedersen and Schmidt *et al.*, however one used in this project is based on the technique proposed by de Groot *et al.* since it allows for a real-time interaction, even with higher quantities of decals. Nevertheless, the algorithms described by Pedersen and Schmidt present valuable solutions to some of the shortcomings of geometric implicit decals, what will be further discussed in the Results section 6.4.

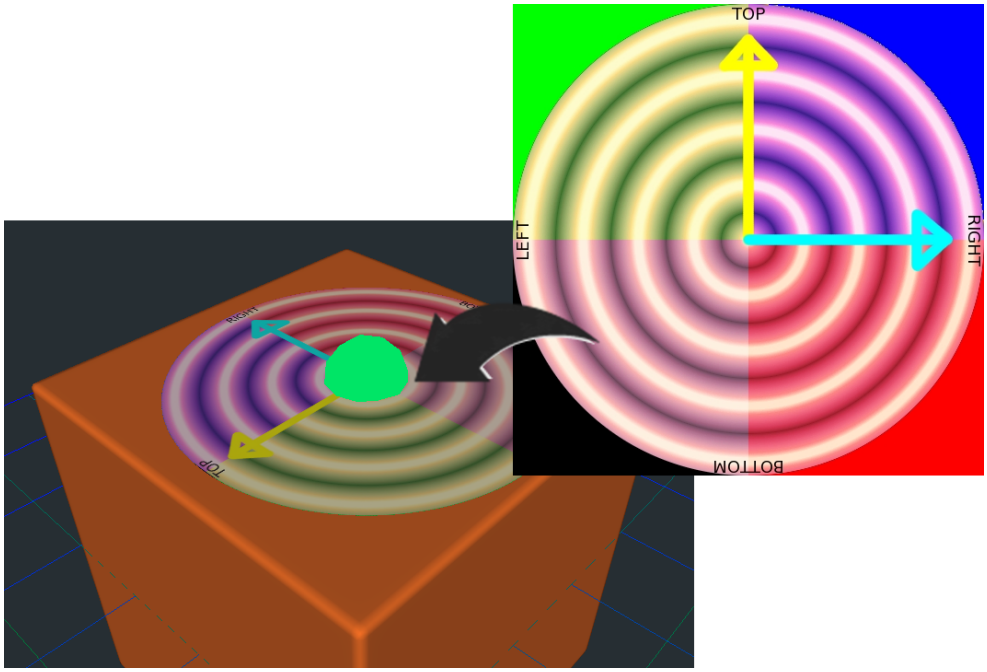


Figure 3.2: *Texturing with decals. The texture map is sampled using parameterisation computed using the decal (the teal sphere).*

Employing a concept of a decal for texturing implies that:

- Each decal D is linked to a certain texture map D_t.
- When positioning D on the parent surface P_s, its DY axis is automatically modified according to Ps‘ normals. DX and DZ

are then aligned with DY appropriately.

- Decals‘ properties are used to calculate the texture coordinates of its underlying surface P_s and later sample the texture D_t in order to define the colour of P_s.

For each processed point q of a surface P_s a number of operations needs to be performed:

1. Select a decal D
2. Calculate the local parameterisation, what in case of texturing is equal to the UV coordinates
3. Sample the texture D_t

3.2.2 Selecting a decal D

Firstly, program must decide whether q is in range of any of available decals. It is achieved through comparing q with positions of all decals (however this can be enhanced through spatial partitioning) and selecting the decal D that satisfies some form of a rule.

In a simple form it means that the closest decal available is used, while it may also implement more complicated formulas, implementing deformable functions, geodesics or exponential mapping.

3.2.3 Calculating UV

The idea behind using decals for texturing comes down to calculating the correct $[u, v]$ (sometimes named $[s, t]$) coordinates of the currently processed/shaded point q.

These can be acquired in a number of ways. de Groot *et al.* in his paper suggest the following method. For any point q and its proxime decal D:

1. Calculate the polar radius of q:

$$radius = \frac{g^{-1}\big(f(q)\big)}{g^{-1}\big(\frac{1}{2}\big)} \tag{3.1a}$$

where

$$f(q) = g\left(\frac{||q - p_i||}{D_r}\right) \tag{3.1b}$$

$$g(d) = \begin{cases} (1-d^2)^3 & \text{if } d \leq 1 \\ 0 & \text{if } d > 1 \end{cases} \tag{3.1c}$$

Unfortunately, de Groot *et al.* does not specify the $g^{-1}(q)$. A following function was found to be working in its place:

$$g^{-1}(d) = \begin{cases} \sqrt{1 - \sqrt[3]{d}} & \text{if } d \leq 1 \\ 0 & \text{if } d > 1 \end{cases} \tag{3.1d}$$

Above procedure is defined to accommodate more advanced types of $f(q)$, such as deformable functions. For a simpler case it can be replaced with a more uncomplicated equation:

$$radius = \frac{||q - D_p||}{D_r} \tag{3.2}$$

2. Calculate the polar angle theta of q :

$$\theta = \arctan\left(\frac{DZ \cdot (q - p_i)}{DX \cdot (q - p_i)}\right) \tag{3.3}$$

3. Convert polar coordinates to square coordinates:

$$u = radius * cos(\theta) \tag{3.4a}$$

$$v = radius * sin(\theta) \tag{3.4b}$$

3.2.4 Sampling the D_t

After the $[u, v]$ coordinates of q are found, program uses them to sample the texture D_t of the selected decal D in order to calculate the colour of q.

If no decal is satisfying the selection rule, eg. none is in close proximity of q, another value can be calculated to define the colour of q, such as a diffuse colour calculated with the Lambert shading model.

3.3 Polygoniser

3.3.1 Overview

Geometric implicit decals facilitate the method described in the previous seciton to extend a polygonisation algorithm (also called a *polygoniser*), effectively creating an interface which allows to modify an implicit surface using implcit decals.

There is a number of polygonisation algorithms used, such as :

- Marching cubes
- Marching tetrahedra
- Dual contouring
- Adaptive skeleton climbing
- Transvoxel algorithm

The concept of geometric implicit decals can be applied to different polygonisers, while one selected for this project is marching cubes, due to its simplicity and speed. Its more detailed description can be found at Bourke or at Shirley *et al.* however in short the marching cubes algorithm, also known as 3D contouring or surface reconstruction, will be outlined in the next section.

3.3.2 Marching cubes algorithm

Polygonisation is aiming to create a polygonal mesh that approximates a certain isosurface of the scalar field defined by $P_{f(q)}$, sampled over a voxel grid. In general case, the isosurface is set at an area in which the value of $P_{f(q)}$ is changing from a positive to negative value, ie. is equal zero.

To create a boundary representation of an implicit surface P_s, that has a field function $P_{f(q)}$, where q is normally a 3D coordinate in form of $[x, y, z]$:

Defining the sampling boundary

Define a regular voxel grid over a 3D space in which the polygonisation is to be executed.

256 cases

Since a cube is used as the polygonising object (cell), there is 256 ways in which the isosurface may pass through it.

For example, the $P_{f(q)}$ may return a value above zero only at a location of one of the eight corners of a cell, meaning that only that corner is inside of the isosurface. In another case, four top corners may return a value above zero, meaning the isosurface is cutting the cell along its y-axis. Isosurface may not pass through the cell at all when it is fully inside or outside the manifold.

Each of such cases is stored in a lookup table.

For each cell

1. Calculate a value of the $P_{f(q)}$ at each corner of a cell q, using its coordinates as attributes of $P_{f(q)}$.
2. Having all 8 values calculated, the algorithm finds if and how the isosurface passes through the voxel. It then uses that information on the lookup table to decide what polygons to construct.

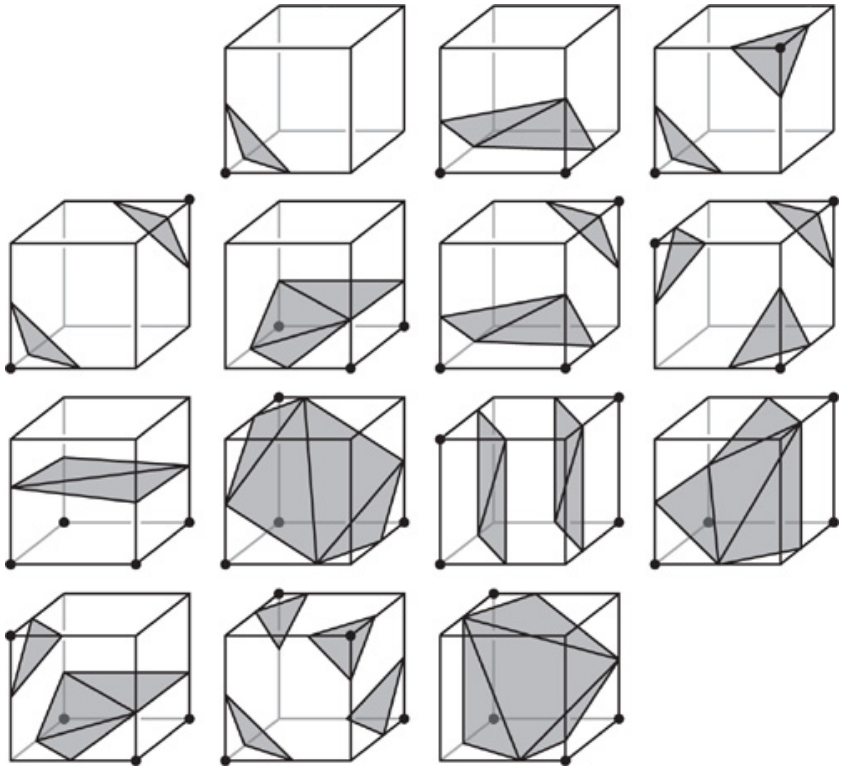


Figure 3.3: *Some of the cases in which an isosurface may pass through a cell. Geiss*

Performing that operation on all voxels in the space produces a boundary representation of P_s.

It is worth noting that accuracy of the result is dependent on the algorithm used as well as on the level of interpolation used. This will be further discussed in a latter section of this document.

Chapter 4

Geometric implicit decal

4.1 Theory

As outlined in the introduction 1, modifying the implicit surface P_s is a slow and unintuitive process, requiring to alter the $P_{f(q)}$ in some way. Geometric implicit decal is an extension of an implicit decal described by de Groot *et al.*, proposing as solution to that shortcoming of implicit surfaces.

In addition to features previously listed in 3.1 and 3.2, each geometric implicit decal has:

- An implicit function $D_{f(q)}$ used to calculate the local parameterisation of decal's underlying surface. This, as described by de Groot, is equal to - but not limited to (see 4.1.2) - calculating texture coordinates.
- A confidence function $D_{c(q)}$ that alters the result of $D_{f(q)}$
- A calling function $D_{call(q)}$, returning $D_{f(q)} + D_{c(q)}$

When calculating a value of $P_{f(q)}$ at a corner q of a cell, the polygoniser would also find a value of $D_{f(q)}$ of one or more decals that are satisfying a certain rule (eg. are proxime to q). These resultant values would then be combined together to define whether the q is inside or outside the isosurface.

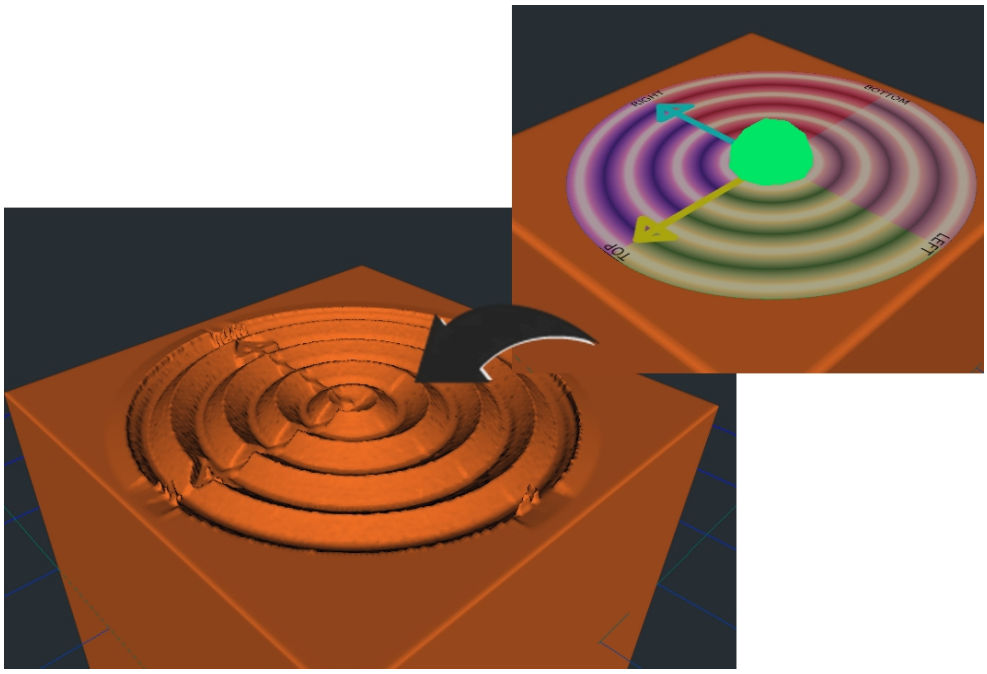


Figure 4.1: *Geometric implicit decal extends capability of an implicit decal, allowing to add texture-based features to an implicit surface.*

4.1.1 Geometric implicit decals‘ algorithm

An algorithm using geometric implicit decals is combining the field function evaluation with an implicit operation of $D_{call(q)}$.

$$P_{f(q)} + \sum_{i=0}^{n} D^i_{call(q)} \tag{4.1}$$

where $n \in \mathbb{N}$ is the number of decals

and $D^i_{call(q)} = D^i_{f(q)} + D^i_{c(q)}$

In more detail, when the polygoniser processes a cell‘s corner q :

1. It finds a value of $P_{f(q)}$ at q
2. Selects decals that are satisfying a rule (eg. are proxime to q)
3. Looping through all selected decals, polygoniser calls for their function $D_{call(q)}$ passing q as a parameter.

$$\sum_{i=0}^{n} D^i_{call(q)} \quad \text{where } n \in \mathbb{N} \text{ is the number of decals} \tag{4.2}$$

4. At each call of $D_{call(q)}$:

   (a) Coordinates of point q are used to find $[u, v]$ coordinates.

   (b) The colour value of D_t sampled at $[u, v]$ is found.

   (c) The resultant value is scaled by the confidence $D_{c(q)}$ and returned.

5. It combines the result of $P_{f(q)}$ and all $D_{call(q)}$ with a geometric operation.

6. That combined value is then used by the polygoniser to determine whether corner q is inside or outside the isosurface.

The rest of the polygonisation is executed normally. This as a result produces a mesh that approximates an implicit surface different from the original P_s, that effectively contains the textures of decals embedded onto it.

4.1.2 Modifying the $D_{call(q)}$

Basing implicit decals on decalcomania introduces an assumption that a decal would be used in relation with a texture map. However it is easy to observe that $D_{call(q)}$ may be performing any other calculation, not necessarily related with texture sampling.

- One such modification of the decal function would be to calculate its own field function, similar to one used within implicit surfaces. For example

$$D_{f(q)} = sin(dist) \tag{4.3}$$
$$\text{where} \quad dist = ||q - Dp||$$

  will embed a wavy pattern on P_s around the D_p. This can be seen in the attached video in section 3.2 *Geometric implicit decals* at 3:12.

- The concept of acquiring local parameterization - texture coordinates

or any other - can be used to sample any other form of data, such as a signed distance field or a vector field

- Essentially any calculation may be performed at this stage, which reveals true potential of implicit decals.

Furthermore, $D_{call(q)}$ facilitates calculation of a confidence function $D_{c(q)}$. Such confidence zone would scale the $D_{f(q)}$ in order to produce a certain modification to its normal result, as can be seen in Figure 4.2

An example of a confidence zone could be:

$D_{cp(q)}$

Planar confidence - the DY along with D_p may be used to create a plane W, used to diminish $D_{f(q)}$ as q becomes further away from W.

$D_{ci(q)}$

Implicit confidence - the $D_{call(q)}$ may accept the result of $P_{f(q)}$ as well as q as an attribute. That will allow to scale $D_{f(q)}$ as q becomes further away from the original surface P_s

$D_{cg(q)}$

Geodesic confidence - a geodesic distance between D_p and q can used to scale the $D_{f(q)}$ as the distance becomes larger

4.2 Implementation

4.2.1 Repolygonisation in detail

At repolygonisation, the marching cubes algorithm is looping through all voxels in its 3D space and calls P_s to calculate a value at each corner q of a voxel, similarly to how a regular polygonisation would execute. This call will now be referred to as $P_{call(q)}$. Differently from normal polygonisation, the algorithm will also calculate the results of $D_{call(q)}$ and combine it with the $P_{f(q)}$.

$$P_{call(q)} = P_{f(q)} + \sum_{i=0}^{n} D^{i}_{call(q)} \tag{4.4}$$

where $n \in \mathbb{N}$ is the number of decals

For each cell's corner q of the voxel grid, a parent surface P_s and a number of decals D parented to it:

1. The parent's field function $P_{f(q)}$ at point q is called and its result is stored as $valueP$
2. Because P_s could have been transformed, the point q is multiplied with the current transformation matrix of P_s and stored as $transQ$
3. Then, the algorithm loops through all children that are implicit decals (there could be other types, as is explained in a further section 5.2.2) and calls their $D_{call(q)}$ 4.2.2 function, passing $transQ$ and $valueP$ as attributes.
4. Results of all $D_{call(q)}$ are summed to a variable $valueD$
5. At the end the the function returns $valueP + valueD$

This result is back again processed normally - like in a regular polygonisation - to define q's location in relation to the isosurface.

After all voxels are processed, the produced mesh is an approximation of the isosurface running through the scalar field defined by P_s (ie. by $P_{call(q)}$). The tool developed for this project adds a final step to this process, converting the resulting mesh to an OpenMesh.

4.2.2 $D_{call(q)}$

On repolygonisation, geometric implicit decals extend the abilities of a polygoniser by allowing it to sample functions from implicit decals. This, as mentioned earlier in this chapter 4.1.1, is executed through a call of $D_{call(q)}$ function during the $P_{call(q)}$. Input from the $P_{call(q)}$ are two attributes: q and $valueP$.

The algorithm is executed in the following order:

1. **Distance check**

   - Vector between q and D_p is stored as

   $$diffVec = (q - D_p) \tag{4.5}$$

   - Its length is stored as

   $$diffD = ||diffVec|| \tag{4.6}$$

   - If $diffD$ is found larger than the radius D_r the function returns 0, otherwise carry on.

2. **Calculate polar angle theta and polar radius**

   $$x = DX \cdot diffVec \tag{4.7a}$$
   $$y = DZ \cdot diffVec \tag{4.7b}$$
   $$\theta = atan(y, x) \tag{4.7c}$$
   $$radius = \frac{||q - D_p||}{D_r} \tag{4.7d}$$

3. **Convert to UV**

   $$u = (radius * cos(\theta)) \tag{4.8a}$$
   $$v = (radius * sin(\theta)) \tag{4.8b}$$

4. **Sample D_t and find luminosity**

   - The texture D_t is sampled using bilinear sampling at $[u, v]$ to acquire the colour value.

   - From this, a *call* value is calculated through extracting the luminosity using the following equation.

   $$call = R * 0.375 + G * 0.5 + B * 0.125 \tag{4.9}$$

- Where R, G, B are the colour components of the sampled colour.

5. **Scale *call* by confidences**

   D_{cp} - Planar confidence

   - Calculate projection of q on a plane W defined by D_p and DY and store it as qW
   - Calculate length of $qW - q$ and store it as qWD
   - Scale current value of *call* by the qWD

   D_{ci} - Implicit confidence

   - Calculate absolute value of $valueP$ and store it as $absoluteP$
   - Define a threshold in which *call* will be scaled down, by specifying $minP$ and $maxP$ range of $absoluteP$ in which scaling will happen.
   - If $absoluteP$ is smaller than $minP$ no scaling will happen. If it is greater than $maxP$, *call* will be scaled to zero. Otherwise, *call* will be scaled to smoothly diminish as the q goes away from the original implicit surface P_s.

Finally, resulting *call* value is returned.

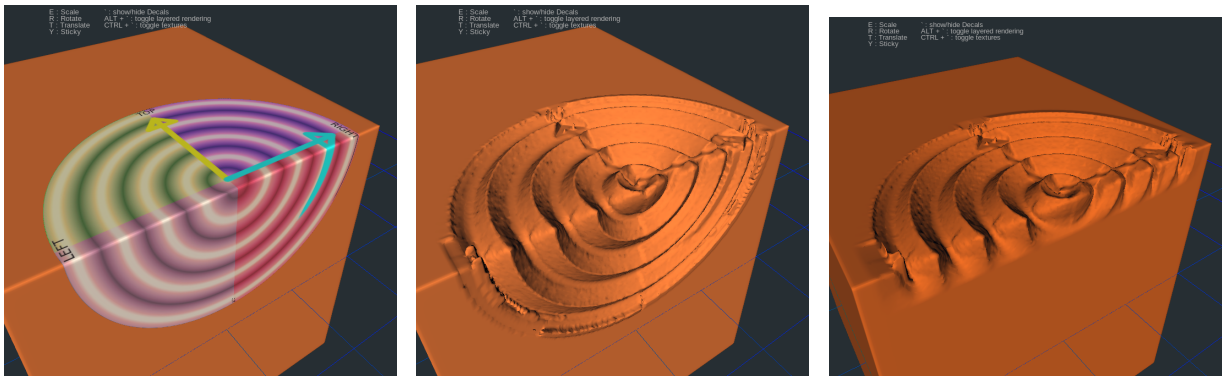


Figure 4.2: *Texture used, polygonisation with D_{ci} and D_{cp}*

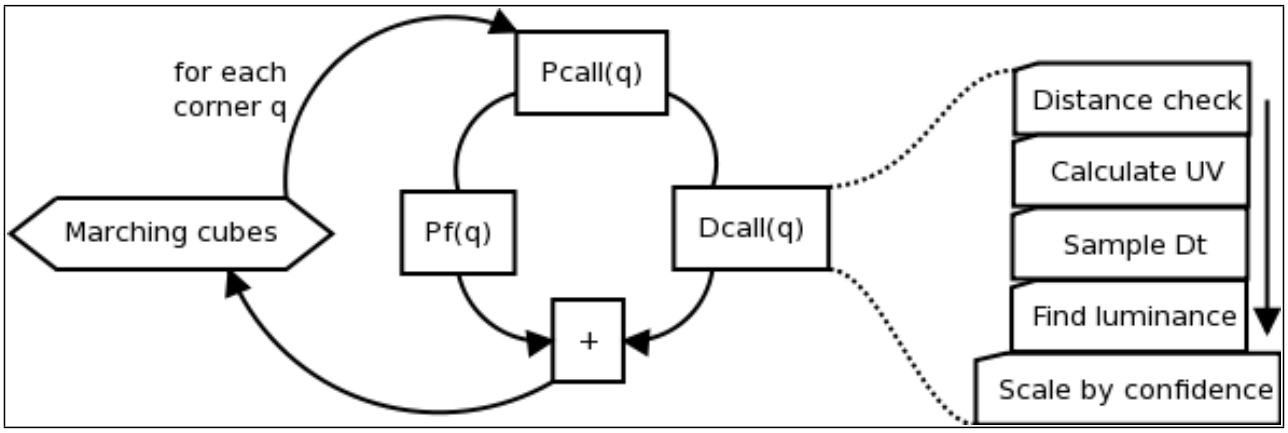


Figure 4.3: *Repolygonisation algorithm. Note that the combination of* $P_{f(q)}$ *and* $D_{call(q)}$ *is performed as a geometric operation.*

Rewriting the equation 4.4, above can be expressed as follows:

$$P_{call(q)} = P_{f(q)} + \sum_{i=0}^{n} \left(D^i_{f(q)} * D^i_{cp(q)} * D^i_{ci(q)} \right) \tag{4.10}$$

where $n \in \mathbb{N}$ is the number of decals

The repolygonisation could be executed in real-time on each user modification, if the polygonisation algorithm was replaced or enhanced with a faster alternative. This will be further discussed in the Results 6 section.

Chapter 5

Application

5.1 Geometric implicit decals tool

The tool developed for this project is a simple GUI-based application implementing the concept of geometric implicit decals. It is written in C++, and uses following libraries:

- OpenGL as the rendering API.
- Qt for graphical interface.
- OpenMesh to represent geometry.
- NGL for graphics-related functionality.
- ISM A to perform marching cubes polygonisation.

5.1.1 Order of usage

To better understand the workflow of the algorithm, let's first look at the outline of the order in which the program is expected to be used:

1. When program starts, it presents the initial parent surface P_s.
2. P_s, along with any other objects in the scene, may be repositioned using the transformation handles

3. Using the graphical interface user creates and modifies decals on the P_s
4. After being satisfied with the placement of decals, user selects the P_s and repolygonises it by pressing enter or clicking on Polygonise button. This essentially calls $P_{call(q)}$ function, combining its original output of $P_{f(q)}$ with results of decals' $D_{call(q)}$
5. Optionally, resulting mesh may be exported by selecting it and pressing the Export Mesh button

At any point user may modify the decals or their textures and repolygonise the P_s which will result in a new, refined mesh. This gives the user a live update on changes made, making the modification process simpler, quicker and more intuitive.

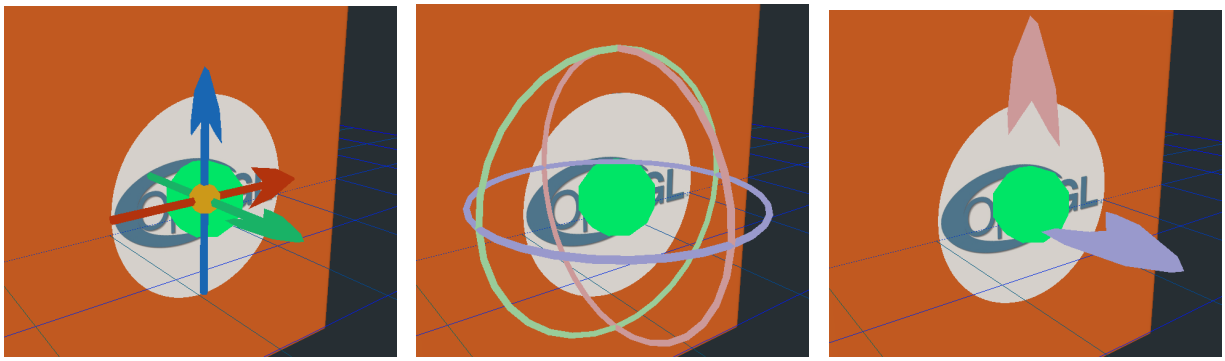


Figure 5.1: *Transformation, rotation and scale handles allowing for real time modifications.*

5.1.2 Real time shading

To aid the placement of decals on the surface, the tool also displays their textures on the mesh to make the process more intuitive. To perform this operation, all data from decals and their textures needs to be loaded to the shader each time the scene is modified.

- $D_p[\,]$ - array with positions of decals
- $DX[\,]$ - x - axes
- $DZ[\,]$ - z - axes

- $D_r[\,]$ - radiai
- texture id of D, defining the $D_t[\,]$
- $id[\,]$ of D
- total amount of decals
- all textures loaded to the program

A GLSL shader is used to display the textures on the mesh, The algorithm of the real time decal-based texturing is almost exactly the same as one used to calculate the $D_{call(q)}$ on repolygonisation. The only difference is that in this case q refers to a global position of the currently shaded point.

Since a number of information is required to be stored on the GPU, the total number of textures had to be limited to 20, while maximum number of decals is 80.

5.2 Interface

The backbone of this tool is a flexible and universal rendering and scene management interface. It is based around the SceneGraph A and Renderer A classes, that along with NGLScene A introduce a robust way of manipulating objects, both in code and in the viewport.

5.2.1 SceneObjects and user input

The core of this system is the way in which objects are handled. Everything is defined as a subclass of SceneObject A, while a list of these is stored within the SceneGraph A, effectively defining the current scene.

To facilitate this hierarchical structure, SceneObject has a number of abstract functions, such as *moved*(), *released*(), *highlighted*(), etc. that allow each concrete class to handle user input individually. Below is a list of all available input events:

- selected
- deselected
- moved
- released
- highlighted
- dehighlighted
- update
- update_parent

To facilitate that system, each object is given an object type, used to define what concrete type of object is hidden behind the processed SceneObject.

For instance, whenever the user moves the mouse around the viewport, currently highlighted object is found using colour selection technique, as described in GPWiki (2012). Later, when a mouse click event is found, the currently highlighted objects *selected*() function is called. With each of such input events it is down to the objects specialisation to define how will the particular object handle the event.

When user clicks at an empty space an empty SceneObject is selected to perform deselection

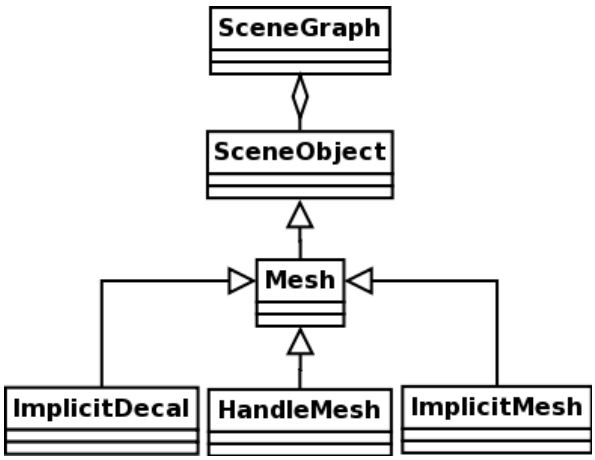


Figure 5.2: *Inheritance relationship*

5.2.2 SceneObjects and hierarchy

To make the interface more universal, SceneObjects are capable of being parented, effectively creating hierarchical trees. Current version of the system allows to define many children of an object, but only one parent.

This characteristic is used in many ways. For example, if a Mesh A has HandleMeshes A attached to it, repositioning of the Mesh would automatically reposition all the handles. And vice versa, on mouse event a HandleMesh is able to recognise its purpose and modify its parent Mesh appropriately.

Another example would be the way in which an ImplicitMesh A finds appropriate decals for repolygonisation. When a decal is positioned on a mesh using the sticky handle, its parent is automatically set to the surface it is sliding on. This way ImplicitMesh simply has to loop through all its children and find ones that have an object type defining them as decals.

5.2.3 Mesh

To represent geometry more generically, a Mesh class was implemented as an extension of SceneObjects. Since the project bases on OpenMesh, Mesh is mainly a wrapper and extension of some of the functionality that OpenMesh has to offer.

Inheriting from this class and using its OpenMesh allowed to easily experiment and change the nature of this project, as described in section *History of the project* 5.3. Since Mesh is subclass of SceneObject, it can be modified and used in the interface independently of the specialistic implementation of the program.

5.2.4 Renderer and RenderTasks

That genericity of the Mesh class is further exploited during render time. The Renderer is setup to process ngl::VAOs, while SceneGraph and the rest of the program uses OpenMesh to represent polygons and vertices.

To bridge that gap, Renderer is first preprocessing all renderable Meshes in SceneGraph and creates a RenderTask A for each unique find. The OpenMesh to VAO conversion is executed using the Converter class.

RenderTasks contains the shader, transformation matrix, current render layer, selection colour, ID and VAO, all directly related to the Mesh with the same ID.

After each frame, each RenderTask is set as inactive, and can be only restored if the same Mesh is once again preprocessed by the Renderer. This time, only certain attributes of the Mesh are updated, depending on whether the Mesh was modified.

If RenderTask remains inactive (ie. its Mesh has been either deleted or set unrenderable), it is automatically deleted before the next render frame.

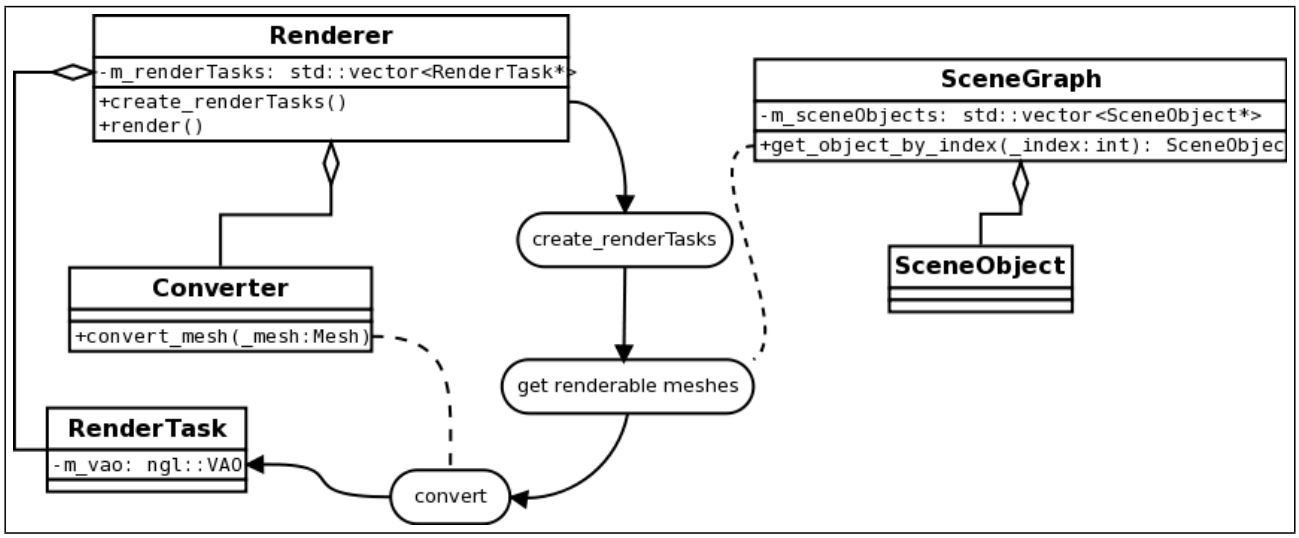


Figure 5.3: *Creating and updating RenderTasks. On each frame Renderer loops through all renderable meshes of the scene graph and updates the RenderTask queue appropriately.*

5.2.5 Render layers

To improve the graphical interface even more, Meshes can be put in different render layers. Each layer has an ID, and Meshes within the same layer are rendered in one depth buffer.

This means that by specifying layer number in which the Mesh is to be rendered, it is possible to put it on front or behind other objects.

The layer creation process is automated and happens after renderable Meshes are preprocessed to RenderTasks.

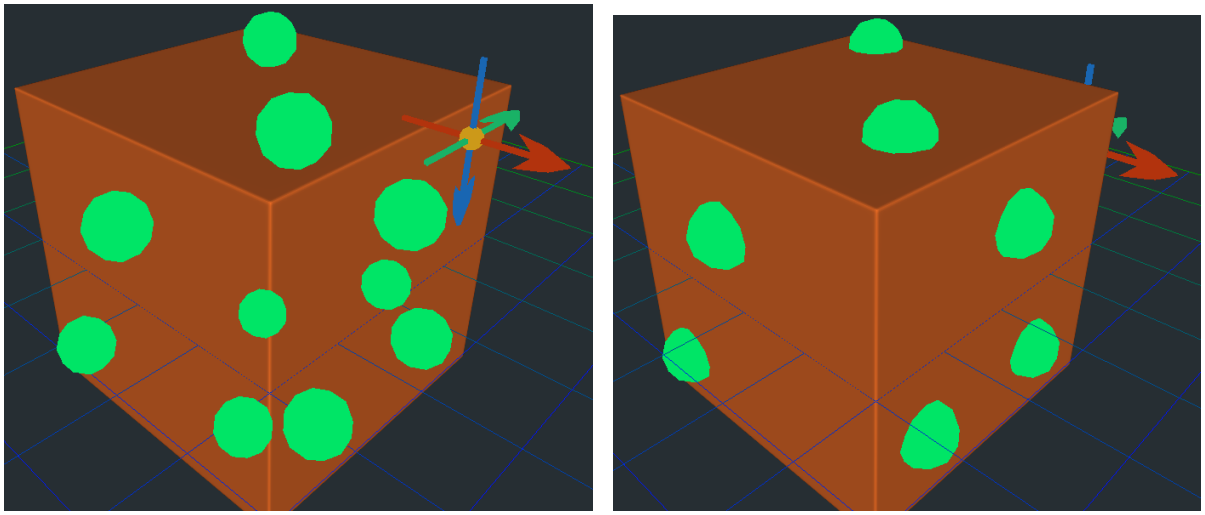


Figure 5.4: *Layered rendering can be toggled on and off from within the tool. All handles are by default put in a layer on front of the meshes.*

5.2.6 Render procedure

In short, the rendering process is as follows

1. Request a render from the NGLScene
2. Preprocess:
   (a) Renderer updates and creates RenderTasks from the current state of SceneGraph
   (b) Renderer creates render layers
   (c) Implicit Gallery uploads information about decals and textures to the shader
3. Render:
   (a) Clean inactive RenderTasks

(b) Loop through all RenderLayers, cleaning depth buffer on each loop. If layered rendering is off, all objects are processed as a single layer.

(c) Loop through RenderTasks of the current layer

(d) Load shader of the current RenderTask

(e) Render the RenderTask. This is equal to binding, drawing and unbinding the VAO of the task.

(f) Set the current RenderTask as inactive

5.3 History of the project

This project is a continuation of the *Texturing Decals* 2014 program that I submitted for my CGIT project as part of MSc Computer Animation and Visual Effects course at Bournemouth University.

Moreover, before reaching its final shape, a number of different applications of implicit decals were considered and researched (see attached video, Section 3. *Work in progress* at 2:03)

Below is a chronological outline of the steppes that lead to the current form.

5.3.1 Starting off Texturing Decals

Texturing Decals was a program implementing the algorithm described in the *Background* 3.2 section, based on the paper by de Groot *et al.* called *Implicit Decals*. Different from this project, *Texturing Decals* - as described by De Groot - were using a more complex function for calculating the texturing coordinates. This was present to facilitate deformable functions, a concept which is not implemented in geometric implicit decals, hence the procedures are simplified. Other than that, the algorithm was the same.

The program was also written in C++, using OpenGL, Qt, OpenMesh and NGL. It presents an interface that allows to create and modify a number of decals on a surface of a mesh, essentially defining its texture.

Texturing Decals was used as a basis for this project also because of its scene management and rendering interface. Most of this system was extended in the current version of the program, and is discussed in the Interface 5.2 section of this thesis.

The classes inherited from *Texturing Decals* are included in their original form with this project in folder *Work/other/Texturing Decals(CGIT project)*

Initial aims

At the beginning, the direction of the project was set on using decals to add detail onto existing geometry. It was not decided which exact target would it take at the end. One of such possibilities was an application similar to *GeoBrush* Takayama *et al.* (2011) that would be used to clone existing geometric detail between meshes. Otherwise, decals could be used to modify an implicit surface, where one of considered implementations was what the project has concluded as.

Independently of the final destination, there was a number of things that needed to be introduced and improved at the start of this project.

Core and NGLScene

Due to enhancements in other areas of the program, a big amount of specialistic code could have been put in different classes, leaving only most generic parts, such as object selection and highlighting.

Core in different iterations of the projects held other geometry, not only the ImplicitMesh.

Both of these classes now also facilitate an ability to restart the Core, bringing it to its original state from when the program was run.

SceneGraph, SceneObject and Mesh

Similarly to this project, in the heart of data management was the SceneGraph class which worked with abstract SceneObjects and mesh handling class Mesh. These were extended to be more generic and therefore allocate specialistic functionality in appropriate classes, what reduced code in NGLScene and Core.

Furthermore, code used to generate primitive geometry (spheres, tubes, arrows, etc.) was moved to a MeshShelf A class, containing all mesh construction functions.

Handles

Texturing Decals contained only a simple control interface in form of three handles. Sticky - one that slides a decal on the parent surface P_s, scale - to define decals radius, and rotation - allowing it to twist around its DY. The y-axis itself would be automatically calculated from the P_s normals when decal was moved.

To extend flexibility of modifying decals three-axial handles were implemented for translation, rotation and scale. Moreover, a camera-aligned translation handle was added, while the sticky handle was kept in almost its original form. These can be seen in Figure 5.1 and in the video in the section 1 *Demonstration of use* (0:13).

Therefore a HandleMesh - inheriting from the Mesh class - was introduced to wrap all range of usabillty related to handles.

Renderer and RenderTask

RenderTask was moved to a separate file and given some updating and initialising procedures, that were previously managed by the Renderer.

Renderer‘s general efficiency was improved, while the class was also extended with layered rendering functionality, normals rendering and support for new shaders.

5.3.2 Mesh-modelling decals

With the interface enhanced, the first aim of the project was to implement a simple modelling tool that would allow a decal to modify polygons of a mesh it was defined on.

This was executed by finding vertices proxime to D_p and displacing them according to the result of $D_{f(q)}$

This can be seen in the attached video in the Section 3.1 *Mesh-modelling decals* at 2:03.

Nevertheless it was useful and interesting, this implementation was fairly basic and considered not very innovative, hence a different path was chosen to be researched for the next iteration of the project.

5.3.3 Implicit-tree decals

Having that quite generic modelling tool implemented, the project was pushed towards the direction of implicit surfaces.

In this version of the program, each decal was treated as another ImplicitMesh and contained its own field function and isosurface‘s mesh representation.

By defining decals on surfaces of the parent surface P_s and other already existing decals, it was possible to create a hierarchical tree of implicit surfaces.

Each decal would have its own way in which it would be combined with its parent, such as union, intersection, subtraction, etc.

On repolygonisation, the algorithm would go down towards the leafs of the tree, and polygonise the surfaces back to the trunk, creating the final implicit surface built from the tree.

Again, this was found not innovative, more widely known as a FRep Tree, hence the direction of the project was altered once again.

5.3.4 Geometric implicit decals

Finally, the aim was set back on using textures, this time to refine the implicit surfaces. This was what the project concluded as, however this step has only clarified what other implementations could be applied using the concept of implicit decals. This will be further discussed in the *Results and discussion* 6 section.

Chapter 6

Results and discussion

6.1 Results

To put the tool in practice, some of the meshes made with geometric implicit decals were exported and rendered using SideFX Houdini and Mantra. These can also be found in the attached video in Section 2. *Results* at 1:25.

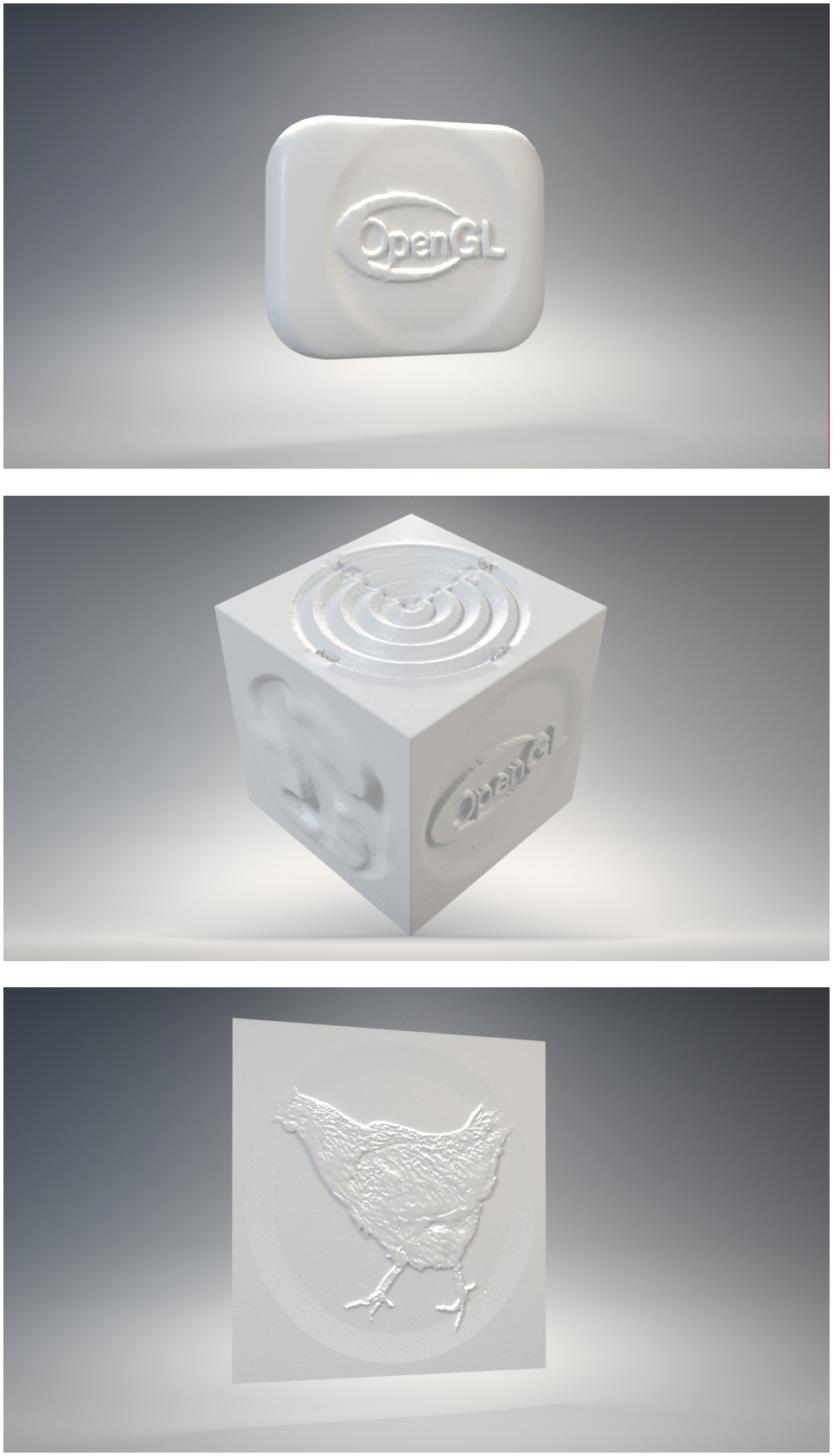


Figure 6.1: *Geometric Implicit Decals in use*

6.2 Evaluation

The project was implemented in C++, built and tested on an Intel Xeon E5-1650 @3.20GHz CPU with 12 threads, 32Gb RAM, NVIDIA Quadro K2000 2Gb GRAM machine.

In the test scene a parent surface was repolygonised with a different number of decals on different resolutions of the voxel grid. In this algorithm, the field function $P_{f(x)}$ alone was found to be taking approximately 58.3%, 48.8%, 29.6% of the total time on resolutions of 128, 256, and 512 accordingly.

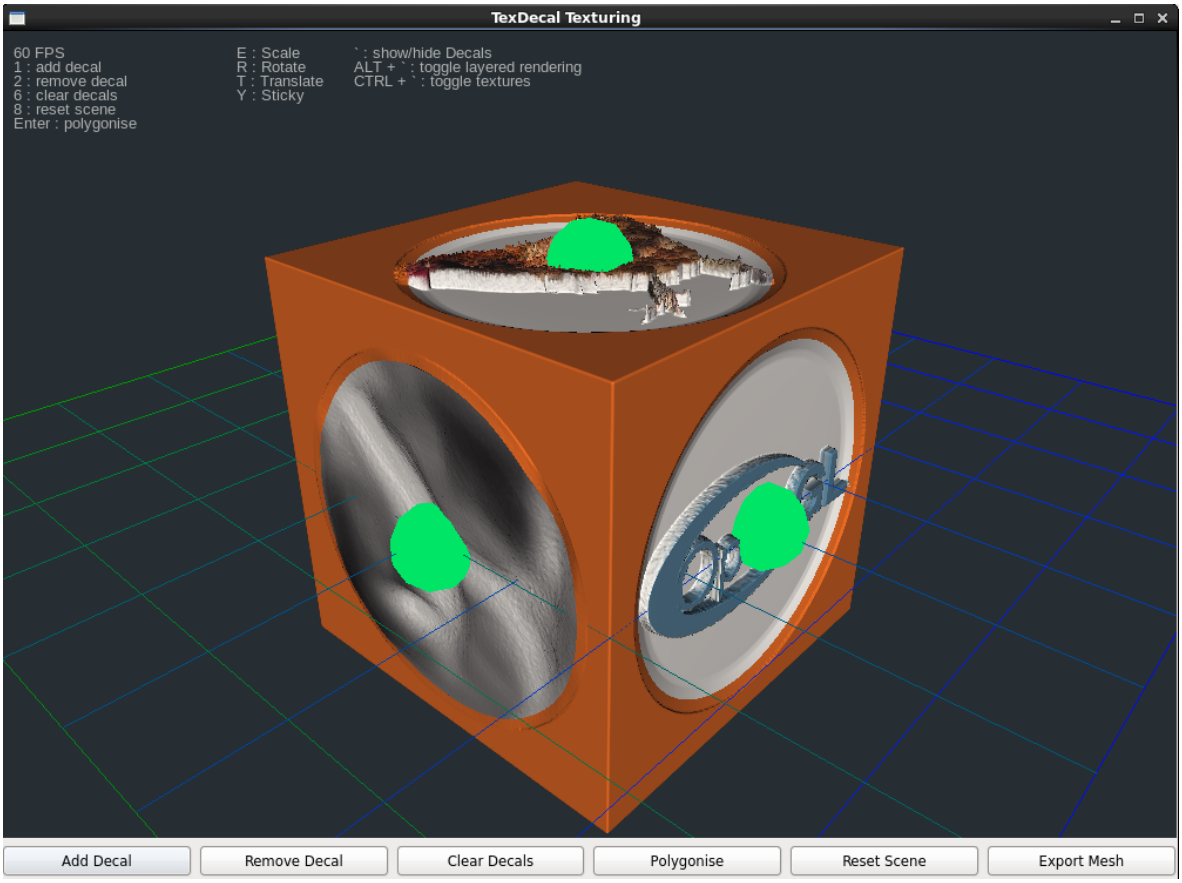


Figure 6.2: *Test scene*

Res.	# of decals	# of faces (approx.)	Repolygonisation time (approx.)	Estimated time of $D_{call(q)}$
128	0	86k	0.18s	0s
128	1	88k	0.21s	0.03s
128	6	99k	0.36s	0.18s
128	30	143k	1.08s	0.9s
256	0	350k	1.2s	0s
256	1	359k	1.4s	0.2s
256	6	408k	2.45s	1.25s
256	30	638k	7.23s	6.03s
512	0	1.395m	6.62s	0s
512	1	1.433m	10.45s	3.83s
512	6	1.655m	18.08s	11.46s
512	30	2.734m	53.89s	47.27s

Figure 6.3: *Polygonisation times for different resolutions and numbers of decals*

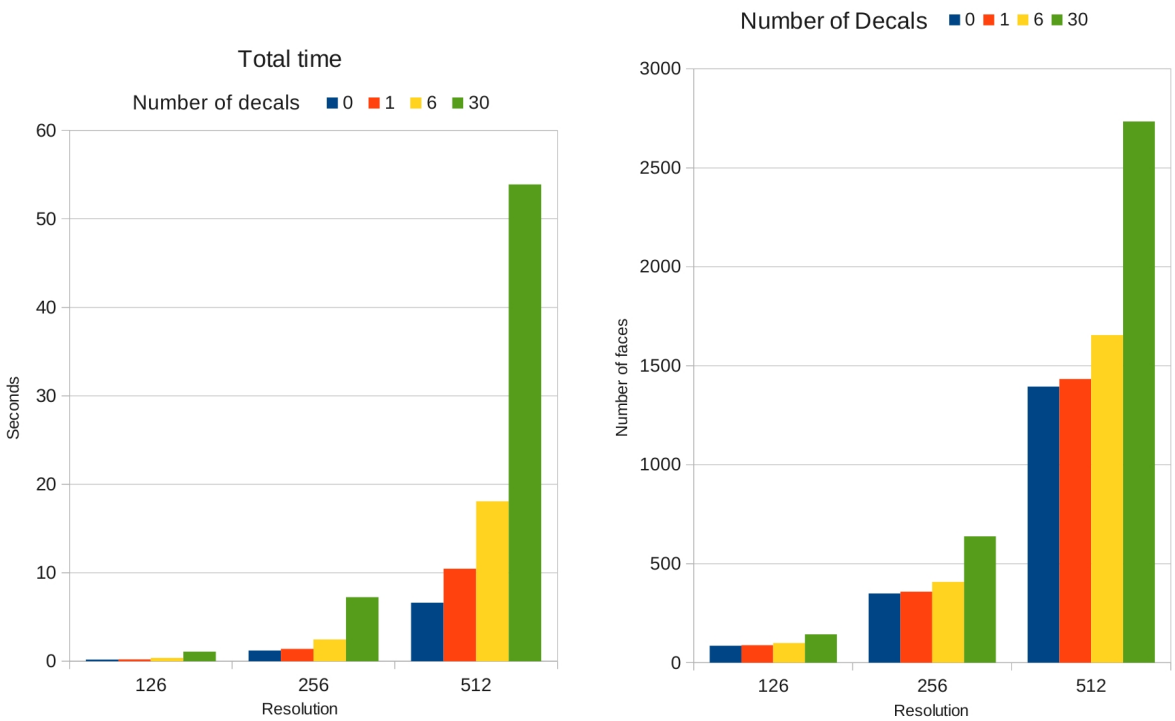


Figure 6.4: *Total time of repolygonisation and resulting number of faces (in thousands)*

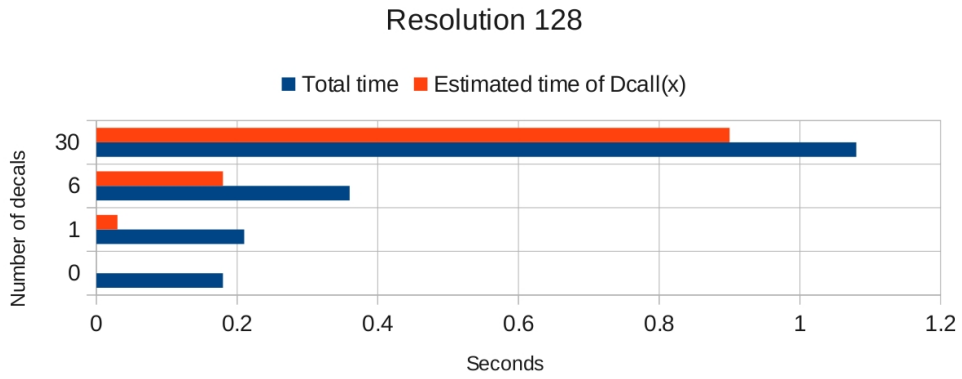
The results of the program was strongly dependant on the complexity of the scene. The biggest bottleneck was the process of repolygonisation, which by its nature is slow. Above breakdowns, seem to confirm a clear observation, that the complexity of the algorithm rises linearly as the number of decals increases.

# of decals	% of time
0	0%
1	14.2%
6	50%
30	83.3%

Figure 6.5: *Percentage of time spent on* $D_{call(q)}$

Interestingly, not only the number of decals but also their size affected the time taken to repolygonise. This is due to the fact that each $D_{call(q)}$ is processed only if q is within D's radius, hence larger radii cause the algorithm to run slower.

The speed obstacle on repolygonisation is difficult to overcome due to complexity of currently used polygonisers. Using GPU to perform the polygonisation greatly accelerates the procedure, even up to real time. Adaptive remeshing could also improve the interface, as in most cases only a small part of the geometry is changed at a time. Moreover, the algorithm could use spatial partitioning to reduce the number of unnecessary checks happening at the beginning of each $D_{call(q)}$ calculation.



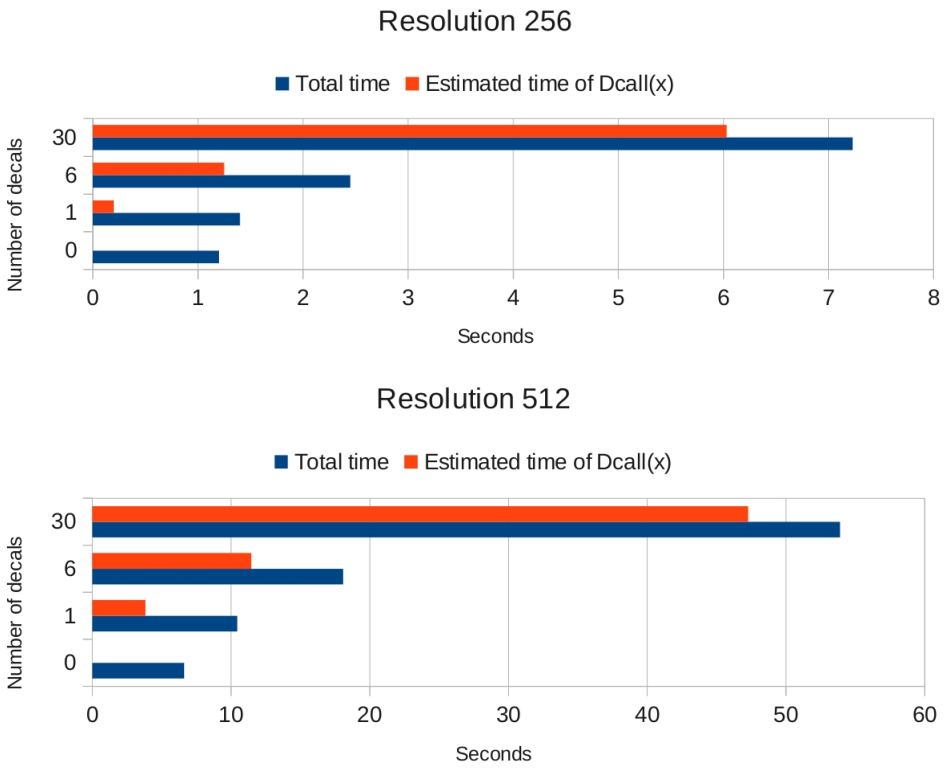
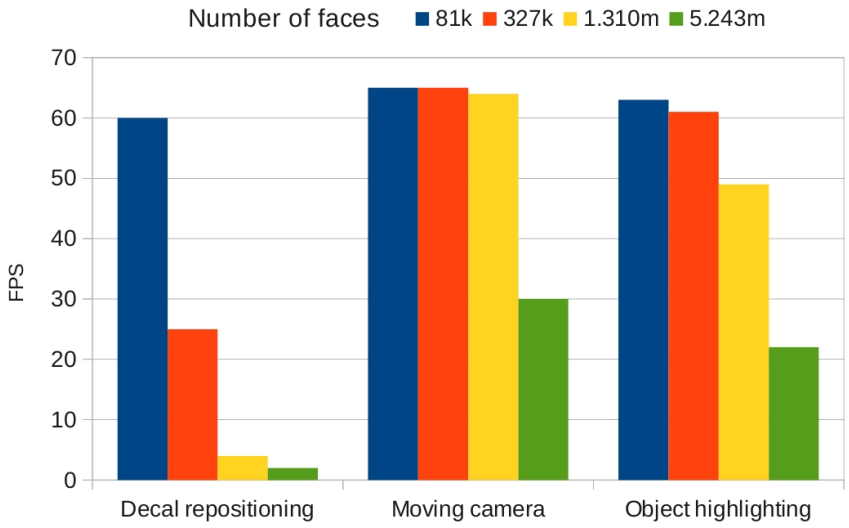


Figure 6.6: *Total time taken compared against the time of* $D_{call(q)}$

Furthermore, many other areas of the geometric implicit decals‘ tool, mainly related to processing the OpenMesh, were also parallelised using OpenMP OpenMP. Resulting interface was able to run smoothly even with meshes of high topology. Below is a table presenting resulting frame rates in different cases of usage. Mesh used for testing was an icosahedron in consecutive levels of subdivision. The figures presented are expressed in frames per second (fps) with a top limit of 65fps.

The noticeably slower results on repositioning decals is caused by the ray-casting algorithm, needing to loop through all existing objects faces, find points of intersections with the mouse-click ray and return the closest point. This can be improved by modifying the algorithm to check colour selection prior to looping all existing faces, or by performing full checking only through colour selection, where each triangle would be given a unique selection colour, however this method would have limitations of its own.



# of faces (approx.)	Decal repositioning	Moving camera	Object highlighting
81k	60	65	63
327k	25	65	61
1.310m	4	64	49
5.243m	2	30	22

Figure 6.7: *Frames per second during different operations*

The way in which decals confidence D_{ci} and D_{cp} is calculated is key to prevent discontinuities and therefore reduce further artefacts. In the current version the confidence falls-off linearly, however a type of filter fall-off function could be applied to improve the continuity. This may be a necessity in a different variation of the $D_{call(q)}$ however in its current form it proves to be suffice.

Moreover, due to the nature of marching cubes algorithm, the resulting polygonal surface contains visible artefacts, particularly present on smooth rendering in non-smooth areas such as corners. Again, this is strongly related with the polygonisation technique and some different algorithms propose solutions to this unwanted outcome. The effects of this downside can be scaled down through postprocessing the mesh by relaxing the vertices and reducing the coplanar faces. A simple way of executing this procedure is to export the mesh and fix it in a 3D package such as Autodesk Maya or SideFX Houdini.

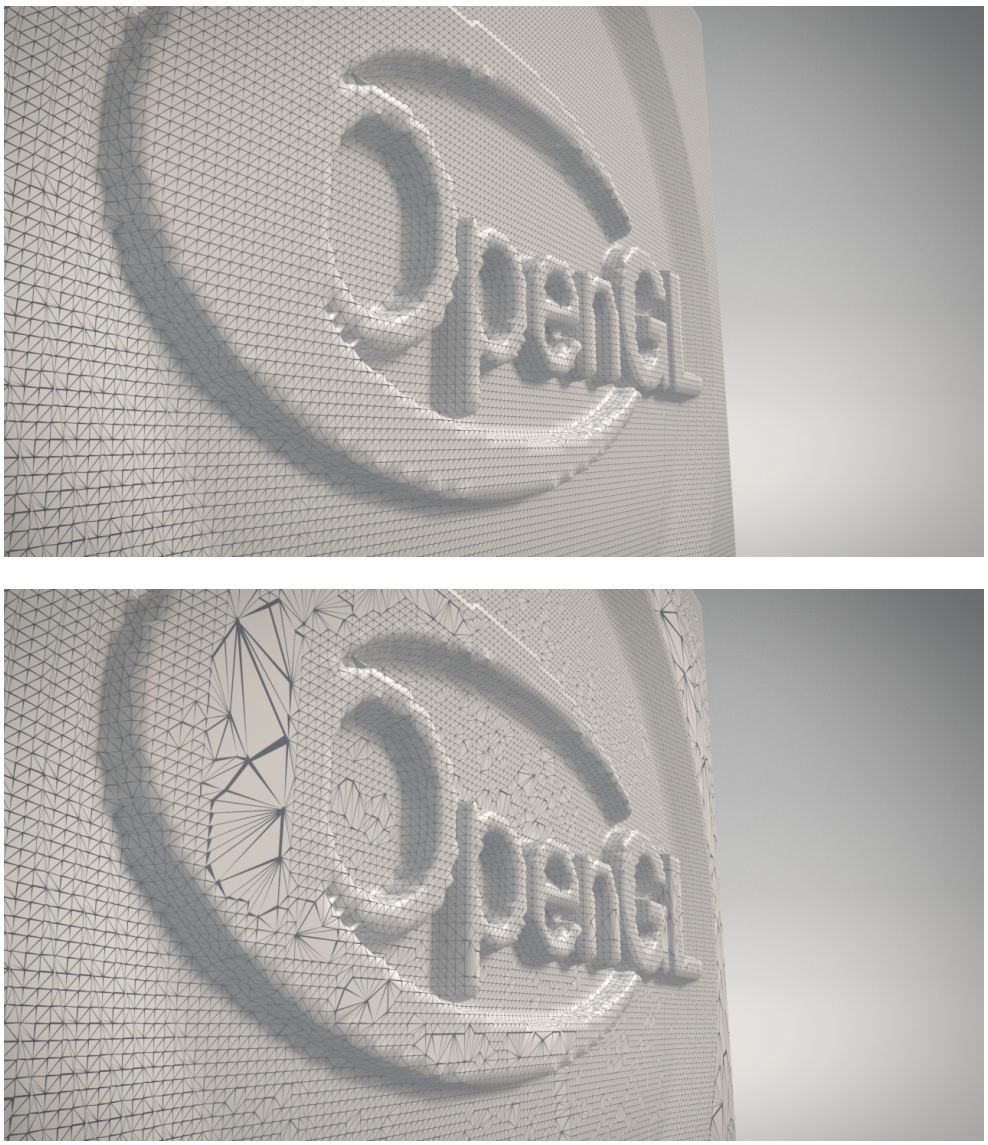


Figure 6.8: *Topology of an original and postprocessed mesh*

6.3 Known limitations and issues

Numbers of decals and textures are limited

In the current version, the program limits the number of decals to 80 and textures to 20. This is due to memory capabilities of the GPU of the machine used for testing.

Polygonisation algorithm is not quick

In cases of low polygonisation resolution time required to compute

oscillates below one second, however as the resolution grows, the complexity of the algorithm causes program to perform increasingly slower. This however, does not refer to the time used on each turnaround, outlined as a problem in the Introduction 1, which has been in overall improved.

Texture maps work planarly

Using texture maps to add small scale features works only in or close to a planar context. In other words, there is currently no wrapping of the textures along the curvature of the surface. Using D_{cp} allows to prevent artefacts caused by this problem, however it also greatly limits the usability of the program. On the other hand, D_{ci} serves as a compromise, producing wrapped confidence zone, whilst the texture is stretched.

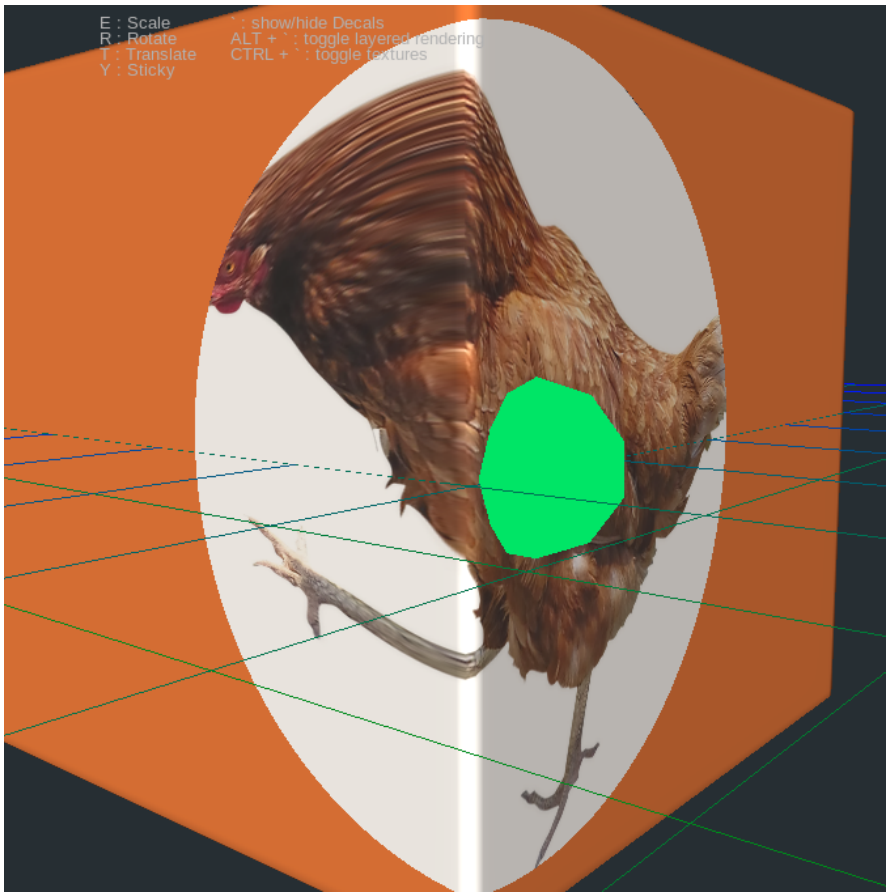


Figure 6.9: *Stretched texture due to planar texturing*

6.4 Future work

1. One of the solutions to the problem mentioned in the last paragraph would be an introduction of geodesic based confidence, as well as calculating the texture coordinates from the geodesic distance, rather than euclidian. This could be executed through implementation of exponential maps as described by Schmidt *et al.*.

2. Continuing from the thought in *Modifying the* $D_{call(q)}$ 4.1.2, the algorithm used on each $D_{call(q)}$ could be extended. An interesting application would be to use 3D data, instead of 2D. This way, a geometrical feature could be captured (using a sign distance field for instance) and reproduced by a decal. That could push the project towards a tool similar to a cloning/stamping known from 2D packages such as Adobe Photoshop.

3. That in fact, would be similar to a solution proposed by the Geobrush 2011, what also suggests that decals could be designed to work with polygonal meshes, rather than implicit surfaces. In such variation, a decal could be used to first capture polygonal mesh within its radius, and then stamp it onto another area of the mesh. In this case, similarly to Geobrush the geometry could have been parameterized using an exponential map and wrapped along the surface curvature using FFD or Green Coordinates. An advantage of such technique over Geobrush would be that stamping decals could have been distributed using a scatter algorithm or a particle system, while the function used to acquire the 2D or 3D coordinates could be deformed, as suggested in *Implicit Decals* by de Groot *et al.*.

4. Looking more at the interface itself, the rendering system could be altered. Since currently used desktop computers come equipped with multithreaded CPUs there was no need to further improve the rendering workflow. However, if it was to be ported to a more-limited machine, such as a mobile device, the interface could use a number of improvements.

5. The rendering system is working on a basis of a single RenderTask queue, waiting for a queue update before each render. If a number of objects was to increase, the rendering could have been noticeably slowed if the updating algorithm was more complex. To overcome this, a concept similar to a game loop could be applied, introducing a double queue system. In such version, two versions of RenderTask queues would be present - a render queue and an update queue. The renderer would always render the one currently attached to it, which would be unmodifiable while used for rendering. The game loop would independently perform updates on the update queue, and exchange it with the renderer after a logic calculation is finished. This would require some synchronisation, however it would increase the speed of execution, give a better overall experience and mark the interface ready for less powerful devices.

6. Other than that, to improve the tool's usability, a number of improvements could be added to the graphical interface:

   - Changing textures of decals during the program execution.
   - Adding and removing textures from the program.
   - Modifying the resolution of polygonisation.
   - Modifying the operation used to combine a result of a particular $D_{call(q)}$ with the $valueP$.
   - An additional mesh attached to each decal that would indicate its y axis.

7. Finally, the program could be converted to a plugin for one of widely used 3D packages, such as 3ds Max, Maya, or Houdini, what would further extend the usability of the project.

Chapter 7

Conclusion

This work manages to achieve a number of goals:

- It is an innovative extension of implicit decals described by de Groot *et al.*. It discusses a number of new ways in which they could be used in various applications.
- It proposes a novel method of applying small scale features to implicit surfaces and outlines possible further research in the area.
- It improves the downsides of implicit surfaces stated in the Introduction 1, being an intuitive system allowing for quick turnaround.
- The interface implemented is quick, universal and extendable.
- Geometric implicit decals are a generic concept that can be assigned to work with different polygonising algorithms.

Proposed technique is solely an iteration of the idea of using locally-derived parameterisation to add detail onto implicit surfaces. Further research in the area could lead to solidifying this promising concept and result in a tool that would propose a strong, innovative alternative to mesh-based computer 3D modelling.

Bibliography

Boost. http://www.boost.org/.

Bomfim D. All about opengl es 2.x - (part 2/3). "http://blog.db-in.com/all-about-opengl-es-2-x-part-2/".

Bourke P., May 1994. Polygonising a scalar field. "http://paulbourke.net/geometry/polygonise/".

de Groot E., Wyvill B., Barthe L., Nasri A. and Lalonde P., 2013. Implicit decals: Interactive editing of repetitive patterns on surfaces. In *COMPUTER GRAPHICS forum*, 1–11.

Geiss R., August 2007. Gpu gems. http://http.developer.nvidia.com/GPUGems3/gpugems3 textunderscore ch01.html.

Gerdelan A. Mouse picking with ray casting. "http://antongerdelan.net/opengl/raycasting.html".

GPWiki , 2012. Opengl selection using unique color ids. "http://content.gpwiki.org/index.php/OpenGL_Selection_Using_Unique_Color_IDs".

OpenMP A. Openmp (open multi processing). http://openmp.org/wp/.

Pedersen H. K., 1995. Decorating implicit surfaces.

Sanchez M. Iso surface modelling library.

Schmidt R., Grimm C. and Wyvill B., 2006. Interactive decal compositing with discrete exponential maps. In *SIGGRAPH*.

Shirley P., MARSHNER , M. A., M. G., N. H., G. J., T. M., E. R., K. S., B. T. H. W., P. W. and B W. Y., 2009. *Fundamentals of Computer*

Graphics, 3rd edition. CRC Press, 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742.

Sunday D. Distance between 3d lines & segments. "http://geomalgorithms.com/a07-_distance.html".

Takayama K., Schmidt R., Singh K., Igarashi T., Boubekeur T. and Sorkine O., 2011. Geobrush: Interactive mesh geometry cloning. In *EUROGRAPHICS*, volume 30, 2.

theowl84 , 2011. Bilinear pixel interpolation using sse. http://fastcpp.blogspot.co.uk/2011/06/bilinear-pixel-interpolation-using-sse.html.

Turner W., 2012. Applying decal. "http://www.flamesofwar.com/hobby.aspx?art_id=857".

user_123abc . Line and plane intersection in 3d. "http://math.stackexchange.com/questions/83990/line-and-plane-intersection-in-3d".

Zankowski W., July 2014. Texturing decals. Submitted as coursework for MSc Computer Animation and Visual Effects.

Zanni C., Bares1 P., Lagae A., Quiblier M. and Cani M., 2012. Geometric details on skeleton-based implicit surfaces. In *EUROGRAPHICS*, volume 49, 52.

Appendix A

Class descriptions

NGLScene

Initialises the program, handles input from the user and executes appropriate functions responding to the QApplication calls, such as updating the view or handling the timer events.

Core

A placeholder and manager for program-specific data, holding objects from the parent implicit surface P_s, through ImplicitGallery A, to Renderer A. Moreover, it contains some utility procedures, such as ray-casting, ray-triangle intersection, as well as commands for Renderer.

SceneGraph

Data structure managing objects in the scene.

SceneObject

An abstract class used to represent an element of the SceneGraph A. It contains all required functionality for transformations, rendering, converting, inheritance and for hierarchical structure of objects in the program.

Mesh

A generic representation of a mesh object in the program. Mesh contains an instance of the OpenMesh, along with a number of functions to modify it, such as flipping normals, vertex transformations,

mesh merging, subdividing, etc. Each Mesh also has an ability of having handles attached to it.

HandleMesh

A subclass of Mesh A, wrapping all procedures required to respond to user‘s object manipulation.

MeshShelf

Contains code used to generate primitive geometry such as spheres, tubes, arrows, etc.

Renderer

A class serving as the rendering interface of the program. Its primary workflow is to directly use the SceneGraph A, converting all declared renderable objects to appropriate RenderTasks A. Furthermore, it supports layered rendering, normals rendering, and contains all shader initialisation.

RenderTask

A representation of a single rendering command. By default each object in the scene has a respective RenderTask, however if needed one object could create more than one RenderTask.

Converter

Converts OpenMesh geometry to ngl::VertexArrayObjects used by the RenderTasks A.

ImplicitMesh

Inheriting from Mesh A, it is used to represent an Implicit Surface in the scene, or in other words the parent surface P_s. It is set up to work with ISM A and with the IsmConverter A. When user defines decals on its surface, they are automatically added to its children list, and later used during repolygonisation.

IsmConverter

Used to convert a mesh generated by ISM A to a Mesh A.

ImplicitDecal

Inheriting from Mesh A, it represents an Implicit Decal in the scene.

ImplicitGallery

A manager class for ImplicitDecals A and their textures. It also contains functionality to load decal and texture information to the shader. These functions are called in the preprocessing stage of rendering (see Interface 5.2).

Texture

Extends ngl::Texture with bilinear texture sampling

Other definitions

DecalShader

The shader used to display decals'textures on their parent geometry. This shader was taken in an almost intact shape from the *Texturing Decals* project (see History of the project 5.3)

Render layer

A collection of RenderTasks A that is given its own clean buffer depth. Render layers are rendered in order of their IDs, allowing to set up the geometry to be displayed in a layered fashion. By default all objects are rendered from a single layer.

ISM

Iso Surface Modelling library, courtesy of Sanchez. It is employed along with ImplicitMesh A to create the approximation of the isosurface using the marching cubes algorithm. It requires an object - in this case ImplicitMesh - that has the *operator*() (here meaning the $P_{call(q)}$) overridden with field function calculation. To achieve quicker results, ISM uses Boost boo parallel processing.

List of symbols

$\boldsymbol{D}$	Decal; Geometric implicit decal; Implicit decal
$\boldsymbol{D_p}$	Coordinates of a decal

$\boldsymbol{D_t}$ Texture of a decal

$\boldsymbol{D_r}$ Radius of a decal

$\boldsymbol{D_{ci}}$ Implicit confidence of a decal

$\boldsymbol{D_{cp}}$ Planar confidence of a decal

$\boldsymbol{D_{f(q)}}$ Private function of a decal, calculating and using the local parameterisation

$\boldsymbol{D_{call(q)}}$ Calling function of a decal, combining distance checks, $D_{f(q)}$ and confidences.

$\boldsymbol{DX, DY, DZ}$ X Y Z axes of a decal

$\boldsymbol{P_s}$ Implicit surface; Parent surface

$\boldsymbol{P_f(q)}$ Field function used to define an implicit function P_s

$\boldsymbol{q}$ Currently processed point

List of Figures

Acknowledgements

I would like to thank Mathieu Sanchez, Tuomo Rinne and Oleg Fryazinov for their aid, advice and supervision during this project.