Implementing Sketch Based Implicit Blending

by

Ishmeet Singh Kohli
Bachelor of Technology, Guru Gobind Singh Indraprastha University, 2012

A Project Submitted in Partial Fulfillment
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Implementing Sketch Based Implicit Blending

by

Ishmeet Singh Kohli
Bachelor of Technology, Guru Gobind Singh Indraprastha University, 2012

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Andrea Tagliasacchi, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Andrea Tagliasacchi, Departmental Member
(Department of Computer Science)

ABSTRACT

Implicit models can be combined using composition operators, but they can produce undesirable characteristics in the results(e.g. unwanted bulging and discontinuities). Recent works in gradient-based operators have been able to address a majority of these characteristics but a major problem still remains: the inability of an artist to build new operators since it involves defining and tuning a large number of parameters requiring specialized knowledge. In this implementation, we use an automatic method for defining such an operator by first capturing sketches from the user around 2D primitives defining the desired blending behaviour and then non rigidly registering a generic template to these samples. The resultant surface interpolates all these samples and hence provides us with a general template for a blending operator. The final operator is derived from this template and can be applied to 3D as well as 2D shapes. I have worked towards developing a c++ based tool for this implementation which is the goal of this project.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dr Brian Wyvill for believing in me and giving me a chance to work with him.

Baptiste Angles for his mentorship, support, encouragement and patience.

and my sister, Harleen for supporting me through the highs and lows.

"If you keep doing what you're doing,
you'll keep getting what you're getting"
Stephen R. Covey

DEDICATION

For my mother, without whom I would not be here.

Chapter 1

Introduction

1.1 Background and Preliminaries

1.1.1 Implicit Objects

An implicit representation of a 3D object describes its surface as a set of 3D points on which a scalar function equals a prescribed iso-value [Bloomenthal and Wyvill, 1997]. Implicit means that the equation is not solved for x or y or z. In the current context, we regularly sample points for all the dimensions of the Implicit surface and solve it. We then polygonize the scalar field using one of the scalar values acting as a threshold to define our surface as shown in Figure 1.3.

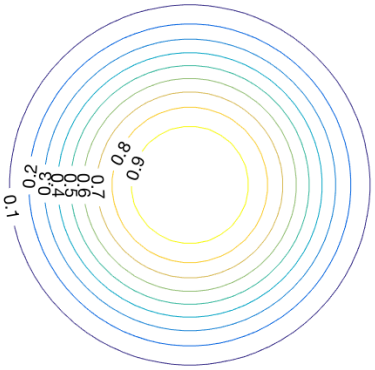


Figure 1.1: Isolines

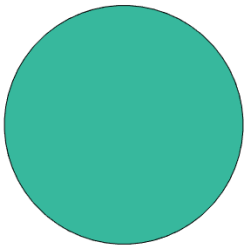


Figure 1.2: Contour at Isovalue 0.5

Figure 1.3: Implicit Circles

1.1.2 Support

The values for a scalar field for an implicit object decrease with the distance from the skeleton. The area where the scalar values are greater than 0 is called the support of an implicit object. The value of the scalar function outside the support is zero and the object has no influence on the composition operators (section 2.1) in this region.

1.1.3 Blending

A seamless surface can be generated by merging two input volumes. The new surface is calculated by combining the field functions of the input implicit objects and computing a new iso-surface [Bernhardt et al.]. A standard operator for generating smooth blends is the sum [Blinn, 1982; Ricci, 1973] as shown in Figure 1.4.

$$g(f_1, f_2) = f_1 + f_2$$

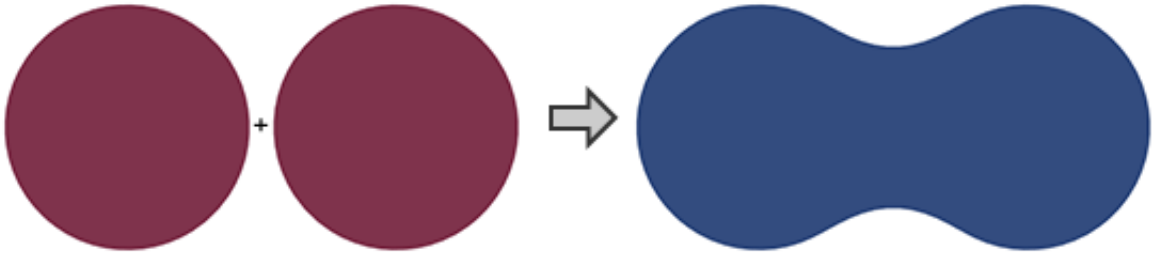


Figure 1.4: Blending using the sum operator

Unfortunately, it does not provide any control on the shape of the resulting blend, and can only be applied to local support field functions or to convolution surfaces[Gourmel et al., 2013]. Therefore, several binary blending operators were proposed over the years for field functions with global support, some of them in [Hoffmann and Hopcroft, 1985; Rockwood, 1989; Barthe et al., 2001].

1.1.4 Gradient

The Gradient of a function represents the slope of the tangent to the graph of the function. More precisely, it points in the direction of the greatest rate of increase

of the function, and its magnitude is the slope of the graph in that direction. As shown in Figure 1.5, the values of the function are represented in black and white, black representing higher values, and its corresponding gradient is represented by blue arrows.

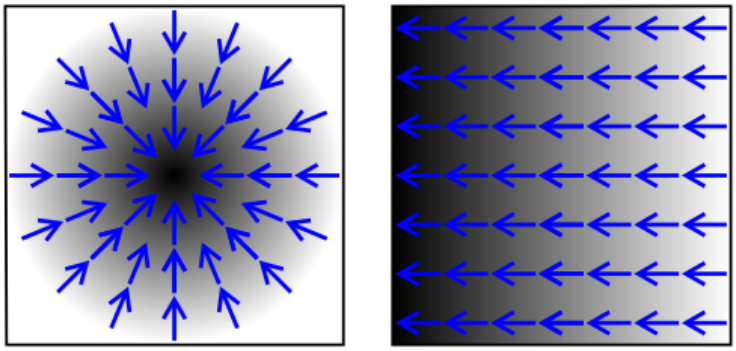


Figure 1.5: Gradients of two functions

1.1.5 Continuity

The most basic form of continuity is C_0 continuity, which ensures that there are no jumps in a function. Higher-order continuity is defined in terms of derivatives of functions.If the first derivative is defined at all points, and the three one-dimensional partial derivatives are each C_0, then the function is C_1. Informally, C_1 surface continuity means that the surface normal varies smoothly over the surface [Marschner and Shirley, 2016].

1.2 Outline of the Report

Chapter 2 describes the problem to be tackled together with its context, impact and the overall motivation for this implementation.

Chapter 3 highlights in detail, the various steps involved in the new method that is used to overcome this problem.

Chapter 4 is where the overall User Interface of the implemented tool is shown along with the various features that this tool provides the user with and how to use them.

Chapter 5 includes the implementation details outlining the various technologies, algorithms and approaches used to synthesize this tool.

Chapter 6 contains some of the results that were produced by the tool.

Chapter 2

The Problem to be Solved

Implicit modeling (also known as implicit surfaces) in computer graphics covers many different methods for defining models. These include skeletal implicit modeling, offset surfaces, level sets, variational surfaces, and algebraic surfaces.Despite the computational overhead of finding the implicit surface, designing with implicit modeling techniques offers some advantages over other modeling methods [Marschner and Shirley, 2016]. One of the first methods was proposed by Ricci as far back as 1973 [Ricci, 1973]. Many geometric operations are simplified using implicit methods including:

1. the definition of blends.
2. the standard set operations (union, intersection, difference, etc.) of CSG.
3. functional composition with other implicit functions.
4. inside/outside tests, (e.g., for collision detection).

2.1 Composition operators

Let us take into consideration scalar functions defined for two objects :

$$f_a(x), f_b(x) : \mathbb{R}^3 \to \mathbb{R}$$

And a binary composition operator, g which is used to define the resulting function :

$$f_c(p) = g(f_a(x), f_b(x)), g : \mathbb{R}^2 \to \mathbb{R}$$

This binary function can be used to combine the scalar fields from the initial two objects but still some of the desired results are not captured by using this approach like a smooth blend at a transition can cause a bulge in the resultant object or a bulge before even getting in contact with each other. These anomalies, i.e. bulging, locality, absorption and topology [Gourmel et al., 2013] are the major problems of this approach, although some of these might not be undesriable in certain cases.

2.2 Gradient based blending operator

To address these anomalies, gradient-based composition operators [Gourmel et al., 2013] have been introduced which select the operator at each point based on the angle, θ between the gradients of the two scalar functions. This class of operators deals with functions of the form:

$$f_c(p) = g(f_a(p), f_b(p), \theta)$$

$$\text{where} \quad \theta = \measuredangle(\nabla f_a(p), \nabla f_b(p))$$

$$\text{and} \quad g : \mathbb{R}^3 \to \mathbb{R}$$

Following this approach addresses most problems mentioned above by using different configurations of the operators for different values of the angle between the gradients. But the main challenge is designing such an operator since it requires core technical knowledge and is complex for artists or modelers to design due to the large number of parameters that have to be tuned.

Baptiste Angles et al. addressed this problem in [Angles et al., 2017] but only built a prototype implementation which did not provide an adequate user interface. In this work, we implemented a c++ version of the software using angles' matlab and other code as a starting point. Due to the gain in efficiency, we were able to produce an interactive system.

2.3 Solution overview

The tool designed in the current work is capable of capturing user sketches over the 2D projections of constituent primitives and then automating the optimization (based on

Section 2.2) to produce the desired operator, without letting the user worry about the intricacies of the underlying implementation. The resultant operator can be applied to both 2D or 3D contexts whenever such a behaviour is required. We achieve this through the following steps:

1. Create a template which forms the basis of the desired operator.
2. Map the user sketches into the operator space.
3. Fit the template to these mapped samples using Non rigid surface registration and by applying necessary regularizers.
4. And finally generating the suitable operator by using the registered template.

Again, these steps are based on [Angles et al., 2017].

2.4 My Contribution

As mentioned, I have based the implementation of this tool on the research work done by Baptiste Angles - articulated in [Angles et al., 2017]. When I started working on this project, some code was available to me in Matlab compatible format. This code outlined the basic workflow for the process in bits and pieces and served as a proof-of-concept for the method and no more. I re-implemented the entire workflow in c++, and I also created a user interface for it for manipulation and visualization. The objective behind this implementation and its advantages are discussed in the following sections.

2.4.1 Speed

Because the earlier implementation was in matlab, the performance was not at par with the desired usability of this tool. A user would expect a real-time output from such an implementation, but the matlab version took significant time for a basic workflow. Since c++ is much more efficient and optimized, I was able to achieve improvements in the performance.

2.4.2 Ease of use

The user inputs and options in the matlab implementation were hard coded and one would have to make changes to the code if they needed a different behavior as an output. Additonally, all the outputs were scattered in multiple output plots and it was difficult to visualize the results. The new implementation makes it much easier for the user to adjust the input parameters for the tool by encapsulating features like moving the input primitives by dragging, drawing the user sketches manually using strokes, having buttons for various process transitions like mapping, registration and generation. It also features modes to visualize each step in the process with a user controllled 3D camera to move around. All these features will be discussed in detail in chapter 4.

Chapter 3

Method

3.1 Capturing User Sketches

The users are provided with a 2D canvas with two 2D input primitives placed on it. These primitives are visualized by closed surfaces polygonized by using an isovalue of 0.5 over the scalar fields of these primitives. We also show the support of the primitives by drawing a thin closed line drawn around the supports of these primitives. The users are able to freely move these objects on the canvas and place them anywhere as per their wish. The user is then able to sketch freely around the scalar fields of these primitives using single or multiple sketches/strokes as shown in Figure 3.1.

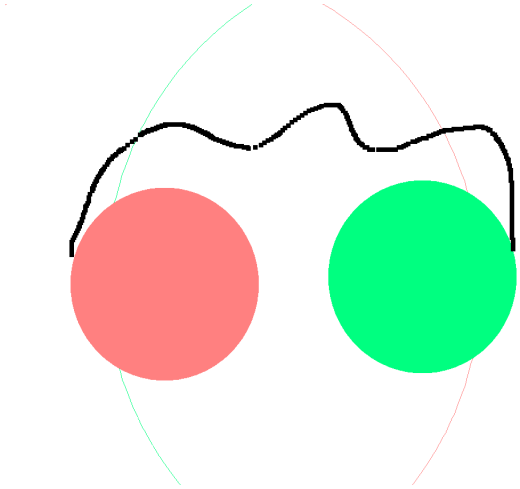


Figure 3.1: User Sketches

3.1.1 Mapping User inputs

We extract a set of n samples $\{s_1, ...s_N\}$ from the collected user sketches where each sample $s_n \in \mathbb{R}^2$ specifies a position that the user wants the final model to interpolate. We then map each to these samples to a corresponding sample $\bar{s}_n$ in the operator domain $\mathbb{D}$, where the x and y components of the operator are given by the values of scalar fields for first and second primitive respectively at s_n and the z component is given by the angle between the gradients of both the scalar fields at s_n :

$$\bar{s}_n = \begin{pmatrix} a_n \\ b_n \\ \theta_n \end{pmatrix} = \begin{pmatrix} f_a(s_n) \\ f_b(s_n) \\ \measuredangle(\nabla f_a(s_n), \nabla f_b(s_n)) \end{pmatrix} \tag{3.1}$$

3.2 Fitting

The samples collected in the last step only define behaviour of our function g in a small subset of $\mathbb{D}$, however we want the 3D blending operator, g defined over its entire space $\mathbb{D}$. Also we need to apply some constraints on the general shape, continuity and boundary conditions (which are discussed in the following subsections) which we can use to deduce the overall extent of the operator over its entire space. We do this by identifying the 0.5 values of g, as a parametric surface (Third order B-spline surface) embedded in $\mathbb{D}$, by fitting it to the samples and applying the constraints simultaneously to the shape of the operator.

3.2.1 Operator Template

We represent g as a surface, P embedded in $\mathbb{D}$ representing the iso-value 0.5. it is created by combining two third-order B-Spline patches, each with 5 X 5 control points, which are stitched at the boundary as shown in Figure 3.2. The whole of the surface is completely defined by the positions of control points $p^a_{i,j}$ and $p^b_{i,j}$. There are some boundary conditions and regularizers that are applied to this template which we will discuss in the following subsections. The continuity of P at the junction of both the patches is enforced by :

$$\forall i \in [1..5] : p^a_{i,5} = p^b_{i,5}$$

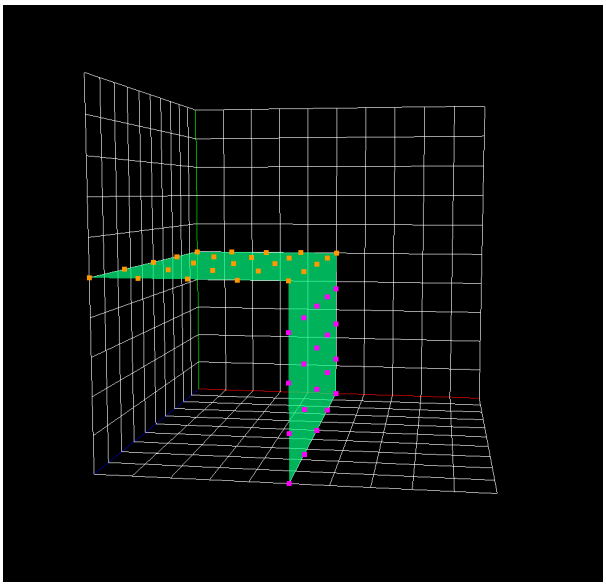


Figure 3.2: Operator Template

3.2.2 Boundary conditions

To achieve C^0 and C^1 continuity on the boundaries, we constrain the control points on the opposite ends of the template to lie on the two line segments at the boundary of $\mathbb{D}$, which are $(0.5, 0, \theta)$ and $(0, 0.5, \theta)$. The control points neighboring the boundaries of P are constrained to be vertically/horizontally aligned to the control points. Also, we constrain each of the rows of the control points to lie at a constant, equally spaced θ values.

3.2.3 Surface registration

To fit the template surface P to the mapped samples $\bar{s}_n$, we start with an initial guess which corresponds to a union operator, i.e. the constituent B-spline patches are planar and orthogonal to each other as shown in Figure 3.2. Then the template is non-rigidly registered onto the samples following [Bouaziz et al., 2016]. This process consists of alternation of two optimization steps.

Local step

This step computes the closest point projections for each sample onto the template surface. This is done by first triangulating P, employing a resolution of 40 X 40 and using a hierarchy of axis-aligned bounding boxes to accelerate closest point queries

from $\bar{s}_n$ to the triangles.

Global step

This step is implemented as a Least Squares minimization step, where the surface control points are modified globally as per the data-to-model error combined with regularizers' energies (Section 3.2.4).The data-to-model error is computed as the averaged squared distances of samples from the tangent plane of their projections on P. We use point-to-plane distances to calculate the error in each iteration which leads to a better convergence as compared to point-to-point errors [Angles et al., 2017].

3.2.4 Regularizers

Laplacian regularization

We penalize oscillations in the surface with a laplacian operator defined on the control points. It ensures a smooth interpolation in an under-sampled area, prevents overfitting and also regularizes our optimization since the sketches only provide a sparse sampling.

Tikhonov regularization

Along with the other energies in the global step, we avoid overshooting with a Tikhonov energy by setting its weight to as low as 10^{-3} which penalizes the displacements from the previous solution.

3.3 Lattice Filling

Once the surface P is fitted to the samples, the next step is to sample the domain $\mathbb{D}$ and assign scalar values for each cell to deduce an operator to be applied to the input primitives. We first assign the voxels surrounding P with signed distances from either of the two patches, by rasterizing over the 3D grid, slice by slice. If a voxel receives two values, it is set to the higher one. We then assign the boundary values for each slice as per the constraints mentioned in Section 3.2.2. Once this is done, the assigned values are diffused over the remaining portions of D.

3.4 3D Synthesis

After we have an operator in place from the previous steps, we calculate the respective 3D scalar fields corresponding to the 2D primitives and the angles between their gradients for a regularly sampled 3D grid acting as (a, b, θ) components of the final 3D model. We then apply tri-linear interpolation to the 3D lattice obtained in the previous step for each point in the grid using the three components and synthesize the final 3D result. The same process can be applied to obtain 2D results as well by using just one slice of the input primitives.

Chapter 4

User Interface and Features

4.1 User Interface

The entry point of the application is a 2D canvas with two primitives already placed on it shown along with their fields. A tab switcher and a toolbar is shown on the screen which can be used to toggle between different modes of the application or to use various features as shown in Figure 4.1.

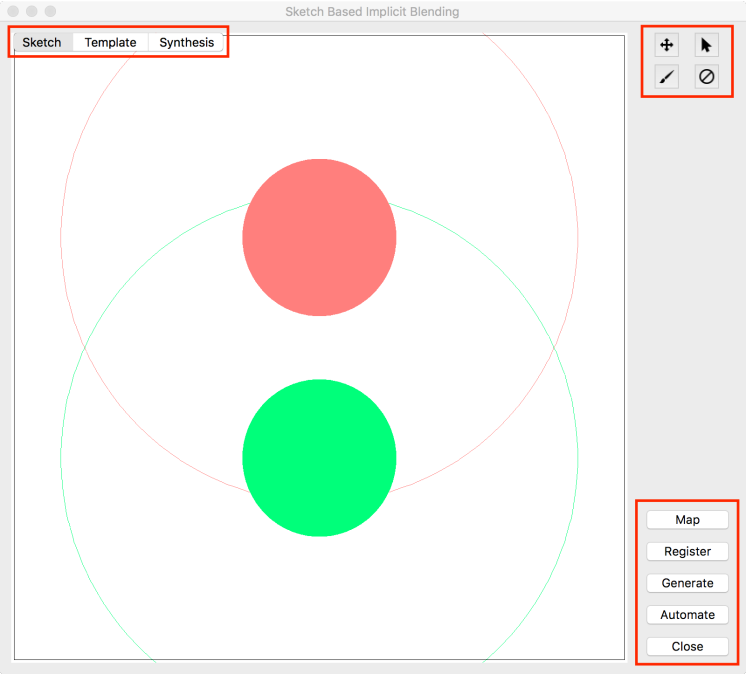


Figure 4.1: Primary User Interface

4.1.1 Modes

The application supports three modes which render the different stages of the process pipeline and can be changed using the tab switcher shown in Figure 4.2.

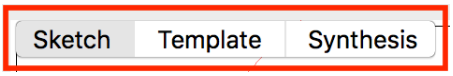


Figure 4.2: Tab Switcher

Sketch

This mode is where the user is able to freely sketch over the 2D primitives specifying the behaviour of the resulting operator and the nature of the desired blending type as shown in Figure 4.3

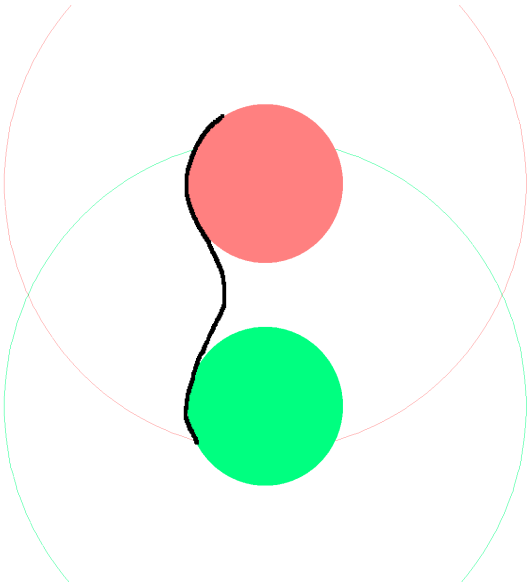


Figure 4.3: The Sketch Mode

Template

In this mode, the user can see how the samples from sketch mode are mapped to the operator domain and can also visualize the process of non rigid surface registration embedded in a 3D grid as shown in Figure 4.4.

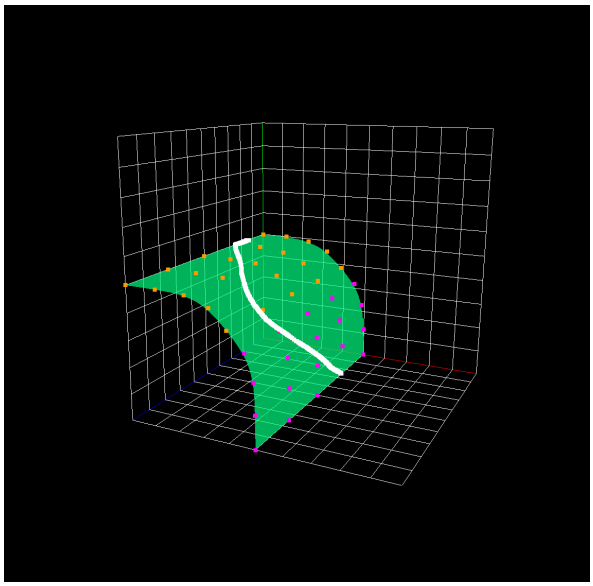


Figure 4.4: The Template Mode

Synthesis

The user can see the generated 3D model from the two input primitives and as per the user defined blending behaviour in this mode as shown in Figure 4.5.

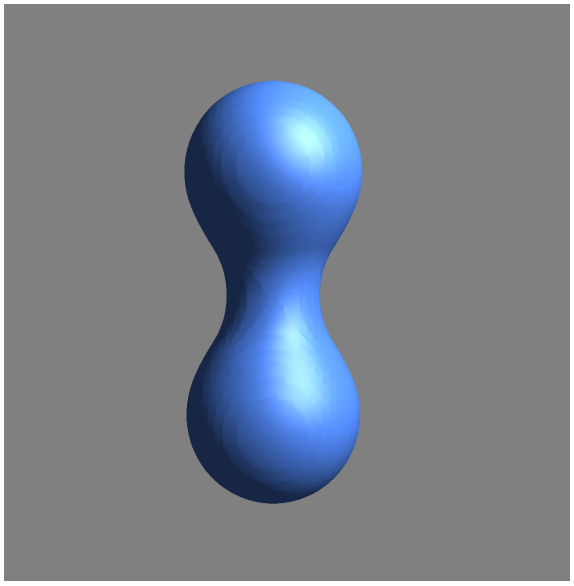


Figure 4.5: The Synthesis Mode

4.2 Features

4.2.1 Moving 2D primitives around canvas

The user is able to move input primitives around the 2D canvas, while in the sketch mode, by clicking on the button shown in Figure 4.6.

Figure 4.6: Move Button

The user can then move any of the primitives by clicking and dragging them around the canvas. In the backend, I have implemented this by adding the position delta of the mouse while it is being dragged, according to the axis it is dragged in, to the vertices of the primitive as shown in Figure 4.7.

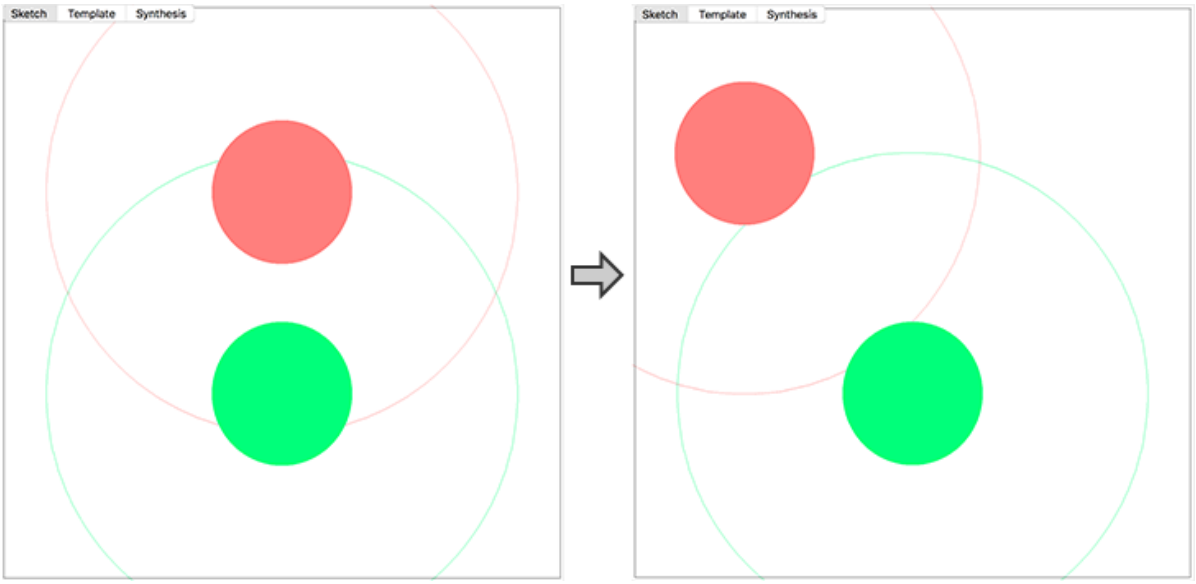


Figure 4.7: Moved primitive

4.2.2 Sketching freely over 2D primitives

The user can sketch freely around the absolute positions of the primitives defining the desired blending behaviour, while in the sketch mode, by clicking on the button

shown in Figure 4.8.

Figure 4.8: Sketch Button

The user can then start the sketch by clicking anywhere on the canvas and dragging the mouse along the desired shape to be interpolated by the final model as shown in Figure 4.3. On the backend, I discretize these sketches into samples to be mapped to the operator domain.

4.2.3 Mapping sketches to operator domain

Once the user has finished sketching in the sketch mode, the collected samples can be mapped to the operator domain as per equation 3.1 using the Map button (Figure 4.1). The user needs to change the mode from Sketch mode to Template mode to visualize the samples in operator domain drawn within a 3D grid as shown in Figure 4.9.

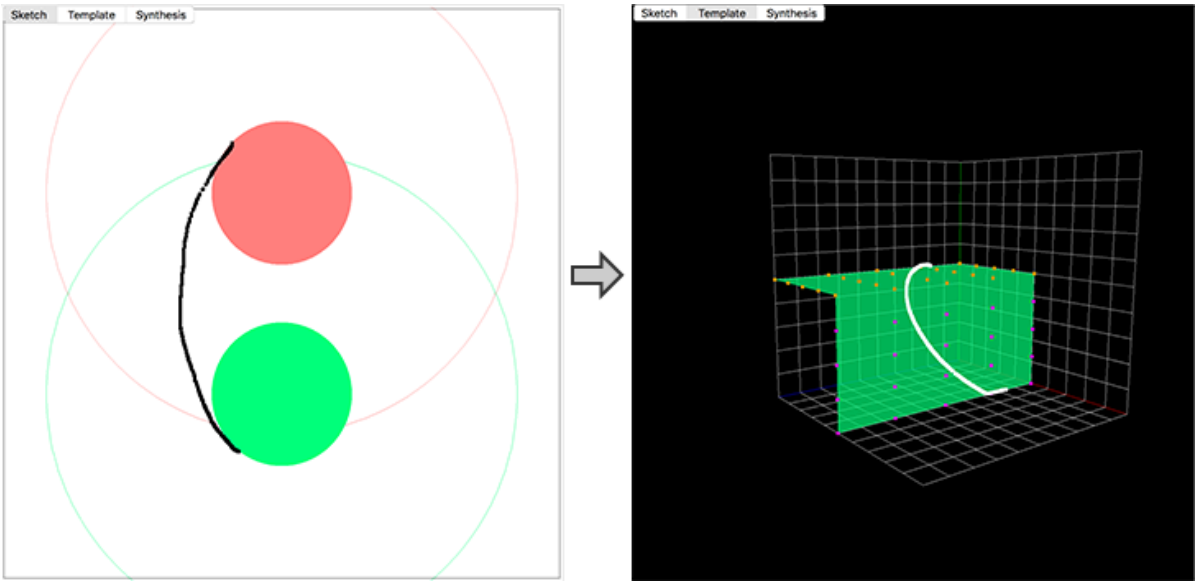


Figure 4.9: User Sketches mapped to Operator Domain

4.2.4 Surface Registration

After mapping the samples, while in the template mode, user can visualize the step by step registration of the template to the mapped points by repeatedly pressing the Register button (Figure 4.1). With each iteration of the register command, template is non rigidly registered to the mapped samples which is visualized in 3 dimensions as shown in Figure 4.10.

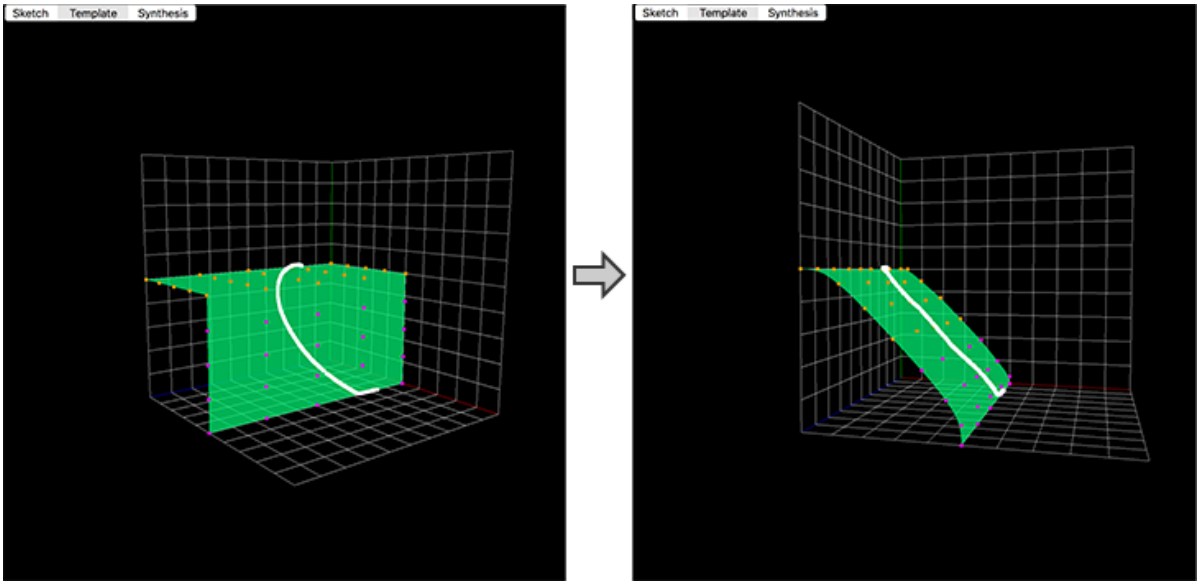


Figure 4.10: Template Registration to the Mapped Samples

4.2.5 3D Synthesis

After making sure the surface is effectively registered to the mapped samples, user is ready to visualize the final result. This can be done by switching the mode to Synthesis mode and clicking on the Generate button (Figure 4.1) once, which is near realtime in this implementation on a 2012 MacBook Pro (retina) to render the results as shown in Figure 4.11.

4.2.6 Additional Features

Automate pipeline

If the users do not wish to view the internal working of sample mapping and surface registration after they are done with sketching, they can use Automate button (Figure

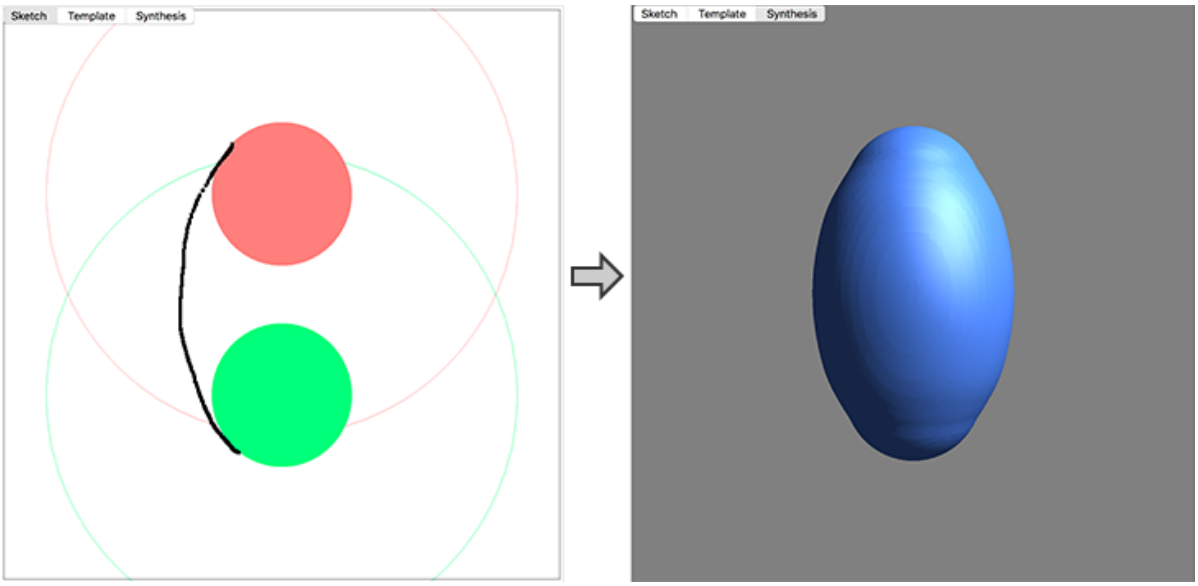


Figure 4.11: Blended 3D Model

4.1) which automates the whole process and shifts the mode to Synthesis mode to visualize the results.

360 Camera fly

For Template and Synthesis modes, which are essential to be visualized in 3D, I have added a simple camera to these modes to move around the space and to visualize the objects from every angle and depth. The controls for the camera are outlined in Table 4.1

Key/Event	Action
W	Move Forward
A	Move Left
S	Move Backward
D	Move Right
Z	Move Up
X	Move Down
Shift + Mouse Move	Look around as per mouse movements

Table 4.1: Camera Controls

Resetting user sketches and surface registration

If users want to reset the current drawn sketches and the surface registration, they can use the button shown in Figure 4.12 to reset the tool to the initial state. This will remove the user sketches from the sketch mode, the mapped samples from the template mode and will reset the shape of the template to initial configuration, i.e. to resemble union composition operator.

Figure 4.12: Reset Button

Chapter 5

Implementation

5.1 Technology Stack

5.1.1 Technologies

Technology	Description
C++11	It is a version of the standard for the programming language C++ which was approved in 2011 which preferred changes to the libraries over changes to the core language. The areas of the core language that were significantly improved include multithreading support, generic programming support, uniform initialization, and performance.
OpenGL 4.1	Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. Its 4.1 version was released in 2010.
GLSL	OpenGL Shading Language (abbreviated: GLSL or GLslang), is a high-level shading language with a syntax based on the C programming language. It was created to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages.

Table 5.1: List of Technologies used

5.1.2 Libraries

Library	Description
Eigen 3	Eigen is a high-level C++ library of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms. Eigen is an open source library licensed under MPL2 starting from version 3.1.1.
CGAL	CGAL is a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library. CGAL is used in various areas needing geometric computation, such as geographic information systems, computer aided design, molecular biology, medical imaging, computer graphics, and robotics.
libIGL	Libigl is an open source C++ library for geometry processing research and development. Dropping the heavy data structures of tradition geometry libraries, libigl is a simple header-only library of encapsulated functions. This combines the rapid prototyping familiar to Matlab or Python programmers with the performance and versatility of C++.
GLM	OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.

Table 5.2: List of Libraries used

5.1.3 Tools and frameworks

Tool	Description
QT	Qt is a cross-platform application framework that is used for developing application software that can be run on various software and hardware platforms with little or no change in the underlying codebase, while still being a native application with native capabilities and speed. Qt is currently being developed both by The Qt Company, a publicly listed company, and the Qt Project under open-source governance, involving individual developers and firms working to advance Qt.
QT Creator	Qt Creator is a cross-platform C++, JavaScript and QML integrated development environment which is part of the SDK for the Qt GUI Application development framework. It includes a visual debugger and an integrated GUI layout and forms designer. The editor's features include syntax highlighting and autocompletion. Qt Creator uses the C++ compiler from the GNU Compiler Collection on Linux and FreeBSD.
GDB	The GNU Debugger (GDB) is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java and partially others. It offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

Table 5.3: List of Tools and frameworks used

5.2 Application Structure

Due to large number of classes in the application, I segregated the classes as per their functionalities into five sections (Figure 5.1) details of which are given in the following subsections.

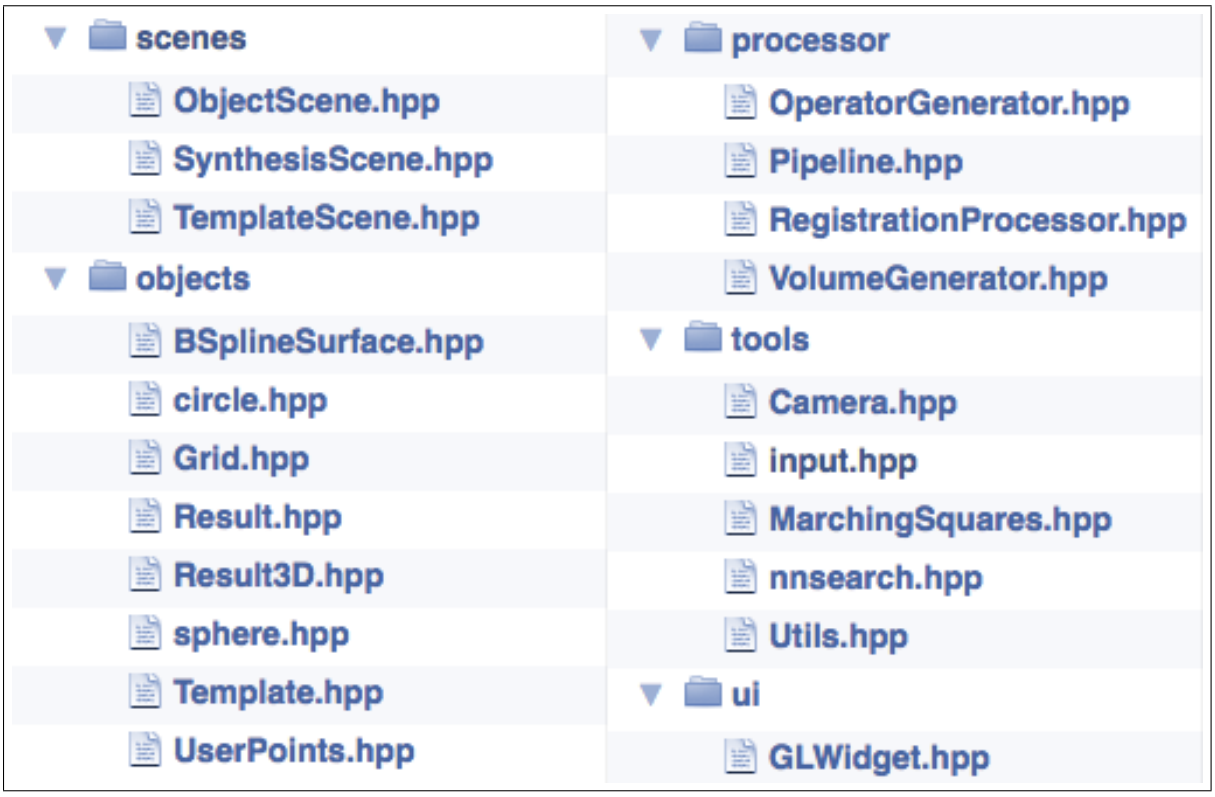


Figure 5.1: Application Structure

5.2.1 Scenes

ObjectScene

This class handles the rendering and flow management logic for the operations related to the Sketch mode. It is responsible for capturing user sketches and handling the primitive movements.

TemplateScene

This class includes the logic for rendering and flow management of the operations related to the Template mode. It is responsible for handling the creation of a generic template, mapping the sampled data from user sketches to the operator domain, registering the template to the mapped points and visualizing this in a 3D setting with a movable camera.

SynthesisScene

This class holds the logic for rendering and flow management of the operations related to the Synthesis mode. It is responsible for handing the generation of the final model along with rendering and shading it in a 3D setting with a movable camera.

5.2.2 Objects

Grid

This class encapsulates the variables and logic related to the 3D grid that is rendered in the Template mode which provides a reference point for the relative positions and sizes of various objects present in the scene.

UserPoints

This class acts as a container for the collected user points from the Sketch mode in a mapped form. It also contains the logic for rendering these points in the scene.

BSplineSurface

This class holds the logic for creation of a B-spline surface by accepting the number of control points and order for the parametric surface. It also provides a public method to evaluate the surface for any specific input points, which is used while generating the operator.

Template

This class includes the BSplineSurface class to create a generic operator template as explained in section 3.2.1 using two B-Spline patches of order 3 with 5X5 control points each. It also includes methods to render and modify the template as per the inputs.

Circle

This class is the container for 2D primitive circles which are used in Sketch Mode. It includes methods to calculate the scalar fields and gradients, to polygonize the primitive using marching squares algorithm, to change the vertices according to primitive movements and to render it in 2D.

Sphere

This class is the container for the corresponding 3D models for the 2D circle primitives. This is used just for calculation purposes, calculating the 3D scalar field and gradients with respect to the position of the corresponding 2D primitive which is used while generating the final model.

Result3D

This class acts as a container for the final 3D model that is generated by combining the scalar fields of the constituent models as per the generated operator.It also includes methods to polygonize the model using marching cubes algorithm and to render it in 3D.

Result

Although in the current version of the tool, we do not show a 2D result but it has hooks to be able to do that as well. This is the class that acts as the container in case we want to see a 2D version of the result.

5.2.3 Processor

RegistrationProcessor

This is the class where all the heavy lifting for the non-rigid surface registration is done. It is responsible for carrying out the iterations for the fitting process to register the generic template to the mapped samples until convergence is reached.

OperatorGenerator

This class is responsible for using the registered template to create an operator that will be used to calculate the overall shape of the final 3D model.

VolumeGenerator

The purpose of this class is to use the generated operator in the OperatorGenerator and calculate the scalar field of the final 3D model using the scalar fields of the two constituents.

Pipeline

This class is a wrapper around the whole process around which this tool is built. It encapsulates RegistrationProcessor, OperatorGenerator, VolumeGenerator and manages the flow of data from various input sources to the output modes.

5.2.4 Tools

Camera

This class holds the view and projection matrices for the modes that require camera movements and is responsible for all the calculations related to camera emulation. It handles responses to specific user inputs and mouse movements.

Input

This class holds an enum for various input states and is reponsible for maintaining the current state of the input, according to which various responses to user inputs are executed. It also holds variables for mouse position and mouse delta.

MarchingSquares

This class is an implementation of Marching squares for 2D models. It is used to create a 2D visual representations for primitives and 2D results. It takes in scalar field for the object and returns the vertices and triangles for the tessellated object.

nnsearch

This class holds the logic for using the nearest neighbour algorithm from the CGAL library and is useful while registering the template to the mapped points.

Utils

This is a global utility class which holds a number of static methods which are used all over in the application to carry out specific atomic tasks like writing the values of a matrix to a csv file and slicing a Tensor into a matrix along the specified direction and index.

5.2.5 UI

GLWidget

This is the central widget for the whole application which holds the references to ObjectScene, TemplateScene and SynthesisScene. It initializes the OpenGL context and acts as the entry point to the render and update cycles, which in turn call the appropriate functions of the respective scenes to render and update objects as per the user inputs.

5.3 Pipeline and Dependencies

5.3.1 Process Pipeline

I have created a singular pipeline for the complete process of the application which can be altered according to user needs by providing the appropriate inputs. The basic structure of the pipeline is maintained and executed within the Pipeline class as mentioned in section 5.2.3. A single instance of this class is shared between all the Scene classes to enable smooth transfer of data from one mode to another. It encapsulates all the processors which perform the computations required for the whole process and hence facilitates seamless execution of respective functions for each of these processors in order of their relevance. Once the user has done with drawing the sketches, we have the basic inputs needed for our process pipeline to start. The initiation of the pipeline is triggered when the user clicks the Automate button. The primary functions used in the high level pipeline execution are briefly explained in section 5.4. A high level diagram of the pipeline is shown in Figure 5.2.

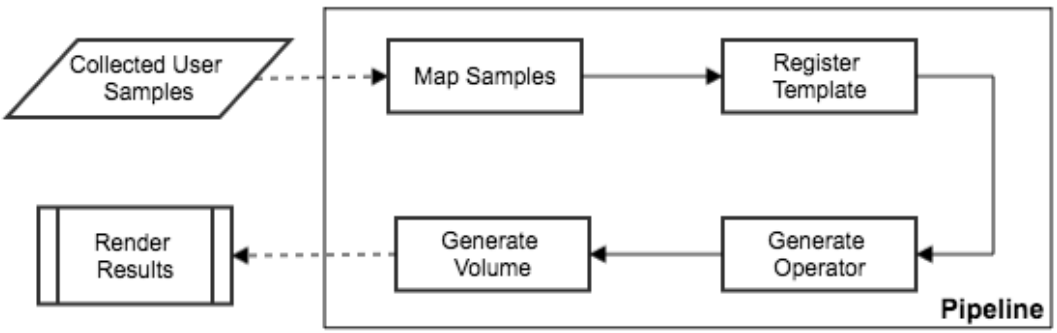


Figure 5.2: Process Pipeline

5.3.2 Dependencies

To highlight how each of the classes defined in the previous section are dependent on each other, I have included a dependency graph in Figure 5.3. The arrows pointing towards a class portray a has-a relationship with the class from which the arrow originates.

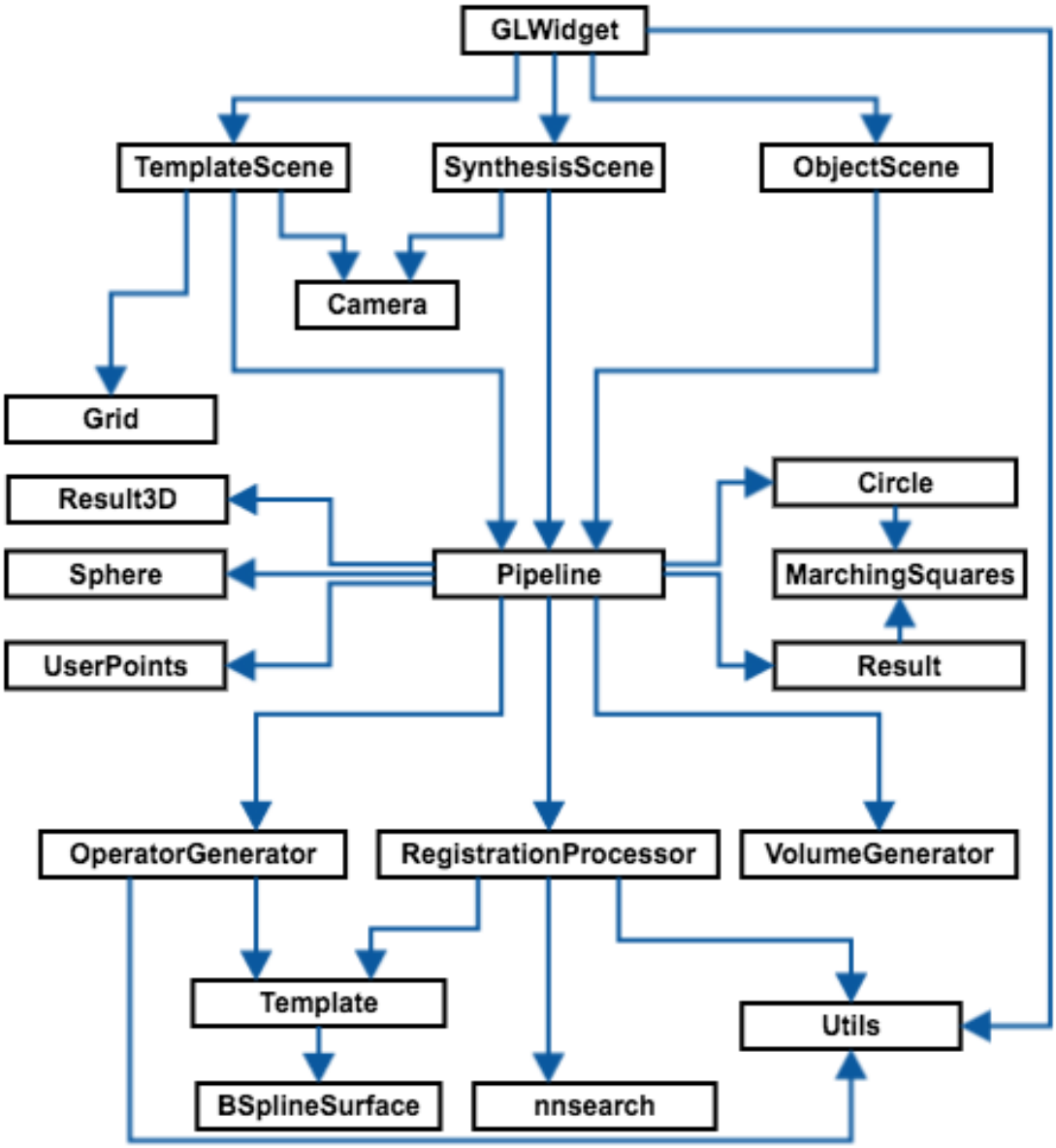


Figure 5.3: Dependency Graph

5.4 Important functions

5.4.1 Pipeline::map()

```
void Pipeline::map (
                    vector<Vector2f>& samples )
```

Parameters

samples	user samples to be mapped to the operator domain

This function accepts the 2D user samples collected from the sketch mode, calculates the distance fields at those points for both the primitives, calculates the angle between the gradients of both the primitives at those points, and maps them to the operator domain using equation 3.1.

5.4.2 RegistrationProcessor::registerPoints

```
bool RegistrationProcessor::registerPoints (
                    MatrixXf &inQueries )
```

Parameters

inQueries	Mapped user samples to the operator domain

Returns

bool	a boolean indicating if convergence has reached yet or not.

This function accepts the mapped user points in the operator domain and carries out one single iteration of surface registration on the template. It creates two matrices with the preconditions defined as per the proximity of mapped points to the surface of the template along with energies of various regularizers, assigning weights to each of the components and solving them using a linear solver. It also checks and returns if convergence has reached or not, by checking if the difference between the control points before the iteration and after the iteration is negligible. This is done by finding the matrix difference between two sets of control points and comparing the norm of the resultant matrix against a threshold value.

5.4.3 OperatorGenerator::fillGrid()

```
void OperatorGenerator::fillGrid (
                    int S,
                    int sampleCount,
                    int Nfactor,
                    MatrixXf X,
                    MatrixXf Y,
                    VectorXf V,
                    Tensor3f &G,
                    Tensor3i &mask,
                    Tensor3i &Gs )
```

Parameters

S	resolution of the operator to be generated
sampleCount	resolution for the rasterization of each slice of the grid
Nfactor	numeric flag to specify if evaluation is for patch1 or patch2
X	regularly sampled mesh grid for x component
Y	regularly sampled mesh grid for y component

Output

G	tensor storing the distance of each grid point from both the patches
mask	tensor storing flags for each grid point specifying if it is assigned value or not
Gs	tensor storing the sign of distances in G

This function rasterizes over the 3D grid, calculating the signed distance from either of the patches. This is done for the entire grid slice by slice. If the grid receives two values, it is set to the one which is higher and hence a regularly sampled grid having a scalar value on each cell of its lattice is calculated to be used in the further calculation of the operator.

5.4.4 OperatorGenerator::solve()

```
void OperatorGenerator::solve (
                    Tensor3f &G,
                    Tensor3i &mask,
                    int S )
```

Parameters

G	tensor storing the signed distance of each grid point from both the patches
mask	tensor storing flags for each grid point specifying if it is assigned value or not
S	resolution of the operator to be generated

This function is responsible for diffusing out the values assigned to the scalar grid. It achieves this by solving a bi-harmonic fairing optimization [Angles et al., 2017] using a Sparse Solver. This is a very complex step since the the lattice is cubic in nature hence this step takes significant amount of time to decompose the values.

5.4.5 OperatorGenerator::generateOperator()

```
Tensor3f OperatorGenerator::generateOperator (
                    int S )
```

Parameters

S	resolution of the operator to be generated

Returns

Tensor3f	a tensor containing the final values for the generated operator.

This function firstly finds the appropriate points on the grid where slicing is to be done, to evaluate the operator slice by slice, then it calls the fillGrid() method for both of the patches. The results received from both the patches are merged using the mask tensor which specifies which grid points are filled for both the patches. If the grid receives two values, it is set to the one which is higher. It then calls the solve method to diffuse the values over the entire grid after setting the appropriate boundary values.

5.4.6 VolumeGenerator::generate()

```
VolumeGenerator::generate (
                    Tensor3f &df1,
                    Tensor3f &df2,
                    Tensor3f &alpha,
                    Tensor3f &G )
```

Parameters

df1	distance field of the 3D representation for the first primitive
df2	distance field of the 3D representation for the second primitive
alpha	angles between the gradients of the two 3D models
G	a tensor for the generated operator

Returns

Tensor3f	distance field of the generated 3D result.

This method accepts the values of all three components in the operator domain as per the location of primitives and approximates the value for each point in the grid by using tri-linear interpolation on the values of the generated operator and returns the scalar field for the final 3D model.

5.4.7 Pipeline::generate()

```
Pipeline::generate (
                    QOpenGLShaderProgram *program )
```

Parameters

program	instance of the current shader program

This function is used when the user manually executes each step of the process pipeline and clicks the Generate button (Figure 4.1) after drawing the sketches in the sketch mode and mapping the samples to the operator domain. It carries out the task of executing the rest of the process steps by first calling the generateOperator() to get the operator, calculating the distance fields and the angles between the gradients for the 3D representation of the input primitives, then using this data collected in

the previous two steps to generate the final 3D model using generate() method of VolumeGenerator.

5.4.8 RegistrationProcessor::automaticRegistration()

```
bool RegistrationProcessor::automaticRegistration (
                    MatrixXf &inQueries )
```

Parameters

inQueries	Mapped user samples to the operator domain

This function iteratively calls the registerPoints() function until convergence is reached and finally sets the iteration count to zero. This is done to automate the process of non-rigid surface registration when the user is interested in automating the whole process by clicking on the Automate button (Figure 4.1)

5.4.9 Pipeline::automate()

```
void Pipeline::automate (
                    vector<Vector2f>& samples,
                    QOpenGLShaderProgram *program )
```

Parameters

samples	user samples to be mapped to the operator domain
program	instance of the current shader program

This function is the entry point of the full process cycle. It accepts the user samples collected from the sketch mode and calls the map() function, followed by automaticRegistration() function and finally its own generate() method to generate the final 3D model.

Chapter 6

Limitations and Future Work

I have a basic engine ready for the calculations and visualization of the final results but still there are some features which can be added to the tool to make it more usable and friendly. Also there are some features which the original paper talked about which are not implemented in this implementation since this is a subset of the original solution. All these features are enumerated in Table 6.1.

1	The tool does not support asymmetric contacts right now and all the results generated are symmetric to both the primitives, asymmetric contacts can be added as an additional feature.
2	All the results generated right now are smooth in nature and if the user desires sharp creases or discontinuities in the final result, there is no way to do it using the UI which can be an improvement.
3	More primitive types can be added to the tool apart from spheres which are the only primitive type supported right now.
4	Support for blending more than two primitives can be added to the tool.
5	The user can only provide inputs using a free form sketch tool, a configurable curve can be added for the same.
6	The resolution of the operator can be made configurable by adding an editable field for it.
7	Support for exporting the final generated mesh can be an enhancement.
8	There is a glitch in the shading of the final result which can be reproduced by rotating the camera around the final result. This can be improved

Table 6.1: Limitations and Future work

Chapter 7

Results

I compared the performance of the older version of the code base against the c++ version and found a significant improvement. All the tests were performed on a basic use case of blending two spheres together in a smooth fashion as depicted in the fourth entry of Table 7.4. I have specified the hardware used in these tests in Table 7.1 and the performance of both the versions are compared in Table 7.2 for multiple configurations.

Hardware	**Specifications**
MacBook Pro	Retina, 13-inch, Late 2013
Memory	8 GB 1600 MHz DDR3
Graphics	Intel Iris 1536 MB

Table 7.1: Hardware Used

Operator Resolution	**Matlab Latency**	**C++ Latency**
20 x 20 x 20	85.91 sec	**0.81 sec**
50 x 50 x 50	94.32 sec	**1.73 sec**
100 x 100 x 100	109.38 sec	**12.08 sec**
200 x 200 x 200	164.72 sec	**146.03 sec**

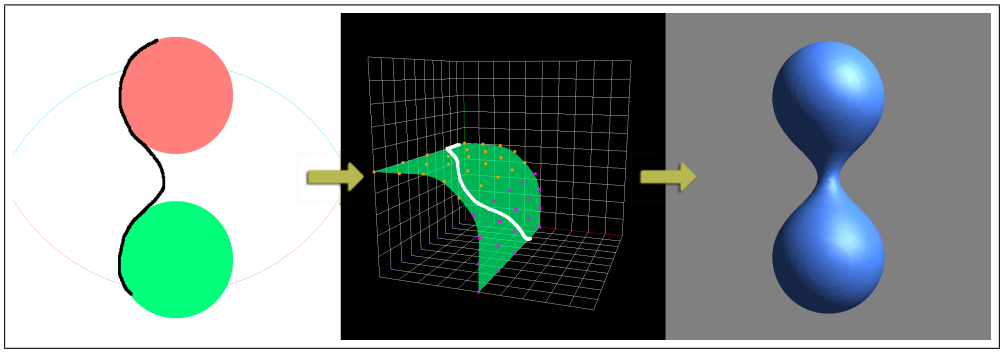
Table 7.2: Performance Comparison

Resolution	**20x20x20**	**50x50x50**	**100x100x100**	**200x200x200**
Mapping Samples	0.0001 sec	0.0001 sec	0.0001 sec	0.0001 sec
Registration	0.3613 sec	0.3685 sec	0.4131 sec	0.3894 sec
Operator Generation	0.0587 sec	0.9301 sec	11.2397 sec	145.083 sec
Volume Generation	0.349 sec	0.3911 sec	0.3913 sec	0.5124 sec
Polygonization	0.0425 sec	0.0445 sec	0.0407 sec	0.0412 sec
Total Time	**0.813 sec**	**1.7351 sec**	**12.0864 sec**	**146.031 sec**

Table 7.3: Time Breakup

As it is evident from the Table 7.3, a major chunk of the time was consumed by operator generation step, which was the primary bottleneck for generating the final results. We could not significantly improve the performance for this step in the current implementation due to the fact that this step was already implemented in c++ in the protoype implementation given the complexity of this step. There was an external call to a c++ function from the matlab code just for completing the operator generation step.

I have compiled the results in response to various user sketches capturing the states in all three modes using the tool which are compared in the Table 7.4.



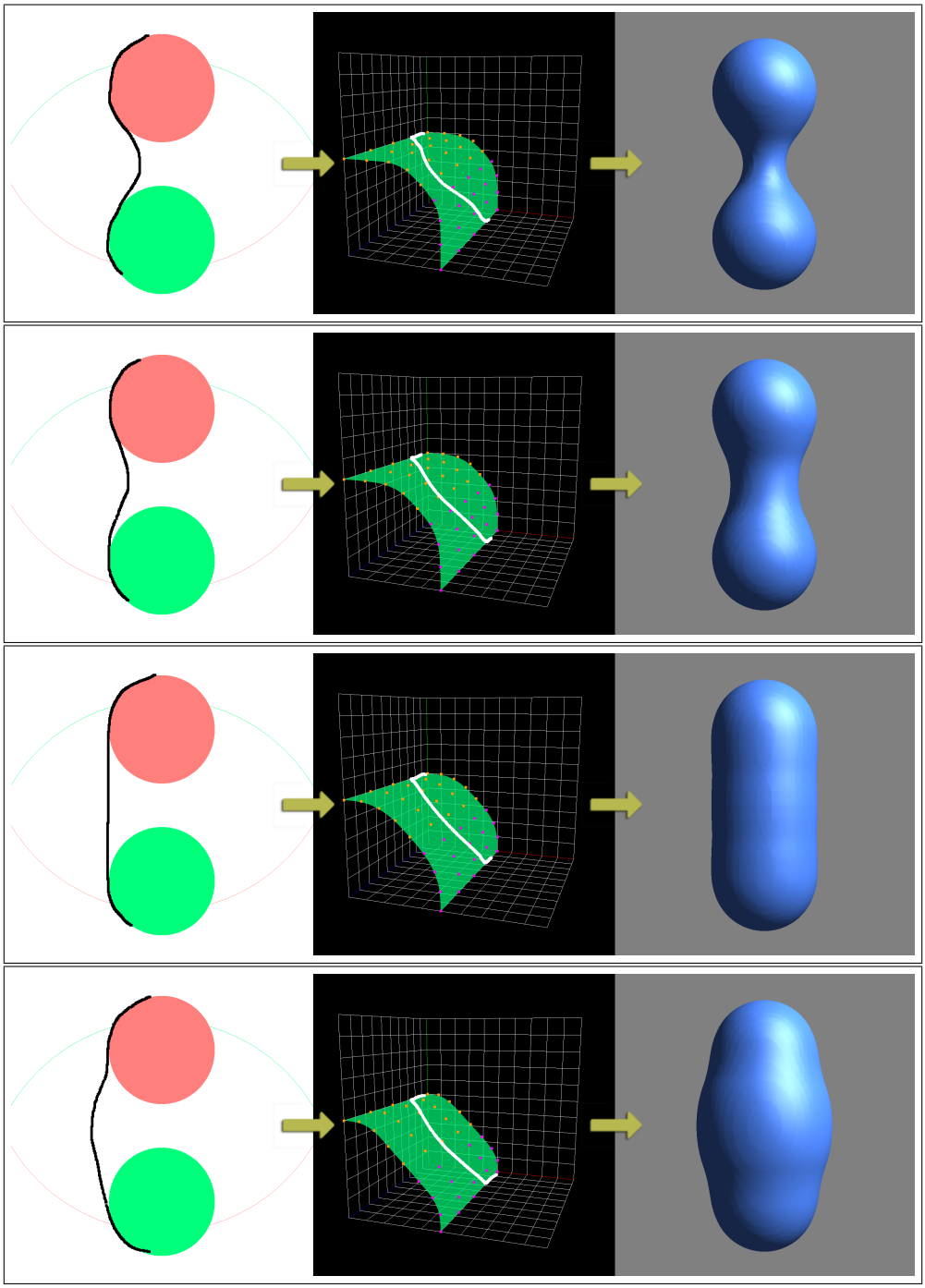


Table 7.4: Results

Bibliography

Baptiste Angles, Marco Tarini, Brian Wyvill, Loc Barthe, and Andrea Tagliasacchi. Sketch-based implicit blending. 01 2017.

L. Barthe, V. Gaildrat, and R. Caubet. Extrusion of 1d implicit profiles: Theory and first application. *International Journal of Shape Modeling*, 7:179–199, 2001.

Adrien Bernhardt, Loic Barthe, Marie paule Cani, and Brian Wyvill. Implicit blending revisited.

James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, July 1982. ISSN 0730-0301. doi: 10.1145/357306.357310. URL `http://doi.acm.org/10.1145/357306.357310`.

Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 155860233X.

Sofien Bouaziz, Andrea Tagliasacchi, Hao Li, and Mark Pauly. Modern techniques and applications for real-time non-rigid registration. In *SIGGRAPH ASIA 2016 Courses*, SA '16, pages 11:1–11:25, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4538-5. doi: 10.1145/2988458.2988490. URL `http://doi.acm.org/10.1145/2988458.2988490`.

Olivier Gourmel, Loic Barthe, Marie-Paule Cani, Brian Wyvill, Adrien Bernhardt, Mathias Paulin, and Herbert Grasberger. A gradient-based implicit blend. *ACM Trans. Graph.*, 32(2):12:1–12:12, April 2013. ISSN 0730-0301. doi: 10.1145/2451236.2451238. URL `http://doi.acm.org/10.1145/2451236.2451238`.

Christoph M. Hoffmann and John E. Hopcroft. Automatic surface generation in computer aided design. Technical report, Ithaca, NY, USA, 1985.

Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics, Fourth Edition*, chapter 22 Implicit Modeling. A. K. Peters, Ltd., Natick, MA, USA, 4th edition, 2016. ISBN 1482229390, 9781482229394.

A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160, 1973. doi: 10.1093/comjnl/16.2.157. URL + http://dx.doi.org/10.1093/comjnl/16.2.157.

A. P. Rockwood. The displacement method for implicit blending surfaces in solid models. *ACM Trans. Graph.*, 8(4):279–297, October 1989. ISSN 0730-0301. doi: 10.1145/77269.77271. URL http://doi.acm.org/10.1145/77269.77271.