Interactive Modeling of Heterogeneous Volumetric Objects

by

Kunio Miyazaki

Graduate School of Computer and Information Science
Hosei University
Email: i04t0019@k.hosei.ac.jp

March 2006
Abstract

We deal with interactive modeling of heterogeneous volumetric objects and issues of an interactive modeler based on the function representation (FRep) using HyperFun. At present, it is impossible to make heterogeneous volumetric objects interactively without loss of precision of a model with existing volume modelers based on voxel arrays. We provide specifications of an interactive FRep modeler using HyperFun in order to solve this problem. Then, we employ FLTK for implementation of a new modeler which helps users create function-based heterogeneous volumetric objects interactively.
Chapter 1

Introduction

At present, only homogeneous objects can be created by existing commercial interactive solid modelers in CAD, animation, and other application areas. Heterogeneous volumetric object is a 3D object with multiple attributes. An attribute is a mathematical model which has object property (material, photometric, physical, statistical, etc.) defined at any point of the point set. Existing interactive volume modelers require voxel arrays to deal with heterogeneous volumetric objects [1]. This causes loss of precision of a model. There is a need in a modeler which can create heterogeneous volumetric objects interactively without loss of precision. Therefore, we introduce interactive modeling of heterogeneous volumetric objects and deal with issues of an interactive FRep [2] modeler using HyperFun [3]. FRep can be used for definition of both geometry and attributes.

HyperFun was introduced in [3] as a high-level language for modeling FRep solids. In FRep, a geometric object is represented by a single real continuous function of several variables as \( F(x_1, x_2, x_3, ..., x_n) \geq 0 \). For example, "25 - x^2 - y^2 - z^2 \geq 0" defines a sphere of the radius 5. In the HyperFun language, we can describe this sphere as an object:

```cpp
my_model(x[3], a[1]) {
```

where \( x[1] \) is X coordinate, \( x[2] \) is Y coordinate, and \( x[3] \) is Z coordinate. HyperFun can define FRep solids using library primitives (algebraic and skeleton-based implicit surfaces, convolution surfaces, and distance-based models). Also set-theoretic operations have built-in special symbols in HyperFun. Thus, HyperFun has enough ability for modeling complex FRep models.

Nevertheless, quite hard manual labor of creating complex FRep models with the current HyperFun tools is a problem. In the current HyperFun tools, users are not allowed to interactively operate in the 3D view, they have to write HyperFun codes first, then compile it, and then render the model.

First, in order to solve problems mentioned above, we formulated requirements and developed a specification of an interactive FRep modeler using HyperFun
HyperFun has many primitive objects such as spheres, blocks, meatballs, convolution objects, ellipsoids, etc. The ideal modeler parses a model in HyperFun language and displays skeletons of primitive objects which are included in the model. When the user arranges primitive objects in 3D view, the modeler displays HyperFun code in text area. Also when users edit the model text, the modeler displays skeletons of primitive objects in the 3D view.

Second, we develop a prototype interactive heterogeneous volume modeler using HyperFun at section 4.2 and chapter 5. We concentrate our attention on the prototype modeler with convolution surfaces [4] as modeling primitives, similar to the work by Goto [5]. The reason of this selection is that skeleton-based convolution surfaces are most suitable for interactive modeling. In addition, the modeler can deal with skeletal primitives (blobby models and soft objects).

We employ FLTK for the implementation. FLTK is a cross-platform C++ GUI toolkit, provides modern GUI functionality and supports 3D graphics via OpenGL. With FLTK, we can make platform independent HyperFun tools. We implemented and experimented with a new modeler which can help users create heterogeneous volumetric objects interactively without loss of precision.
Chapter 2

Related works

As mentioned above, existing volume modelers use voxel arrays to deal with heterogeneous volumetric objects. The first interactive modeler related to continuous function based modeling was introduced by Schmitt et al [6]. It supports constructive modeling of heterogeneous volumetric objects using trivariate B-splines. A new trivariate B-spline primitive which can be used as a leaf in an FRep constructive tree is used as the basic model for both object geometry and attributes. The major weak point of the trivariate B-spline functions is that modeling a complex shape with a lot of details requires a huge number of control coefficients. The authors overcome this weak point using visualized "heat" color scheme which expresses values of the function in 3D views. However, not only B-spline based, but a more general interactive heterogeneous volume modeler is needed.

Goto developed an interactive modeler for convolution surfaces with an extendable user interface [5]. The main feature of the Goto's modeler is an extendable user interface. The modeler is extensible by convolution surface primitives, which become built-in primitives of the modeler. If convolution surface primitives using new kernels are added to the FRep library, it is possible to use those primitives without rewriting the source code of the modeler. Also, the proposed initialization file for binding convolution surface primitives is important. This file informs the modeler what primitives are used. The initialization file includes 4 blocks: HFModel, Parameters, Defaults, and Visual. A HyperFun convolution surface primitive is specified in the HFModel block. Declarations of primitive parameters are presented in the Parameters block. Default values of parameters are given in the Defaults block. And in the Visual block, a built-in primitive of the modeler is associated with a specific visual representation of a convolution surface primitive. The modeler uses this information to display skeleton models during interactive modeling, and to generate HyperFun files. Our interactive heterogeneous volume modeler is similar to the Goto's modeler in using convolution surfaces.
Chapter 3

Heterogeneous volumetric object

Before we discuss the interactive heterogeneous volume modeler, we explain some details and connection between FRep, HyperFun language, and heterogeneous volumetric objects.

3.1. Function representation

FRep is a more generalized representation of a geometric object than traditional implicit surfaces, constructive solid geometry (CSG), sweeping, and other solid models. In FRep, a geometric object is represented by a single real continuous function of several variables as $F(x_1, x_2, x_3, \ldots, x_n) \geq 0$. If the value of function is positive at a point, the point belongs to the object. The value of the function is zero on the entire surface of the object. This surface is called an "implicit surface." If the value of the function is negative at a point, the point is outside the object.

Operations such as set-theoretic, blending, offsetting, metamorphosis, sweeping, and others, are also applicable. These operations have been formulated for FRep in such a manner that they yield continuous real-valued functions as output. Operations are applied to primitive geometric objects (spheres, blocks, convolution objects, ellipsoids, etc.) to construct more complex ones.

3.2. HyperFun language

HyperFun language was proposed as the FRep based shape-modeling language. It allows for a parameterized description of functionally based multidimensional geometric shapes and supports all the main concepts of FRep. Primitives and operations are defined by an equation or by a procedure converting point coordinates into the function value.
HyperFun is similar to universal programming languages such as C or Java. It is a specialized high-level programming language for specifying FRep models. But unnecessarily constructions are removed for language simplicity.

HyperFun has some advantages such as multidimensionality, features as exchange protocol, extensibility openness, and others. Multidimensionality is especially important for heterogeneous volumetric objects.

### 3.3. Heterogeneous volumetric object

Heterogeneous volumetric objects can be regarded as "limited hypervolume objects" with 3D geometry only. In [1], FRep is used for modeling multidimensional point sets with multiple attributes (hypervolumes). FRep representation of a hypervolume object can be expressed as:

\[
\omega = (G, A_1, \ldots, A_k) : (F(X), S_1(X), \ldots, S_k(X))
\]  

(1)

where \( X = (x_1, \ldots, x_n) \) is a point in n-dimensional Euclidean space \( \mathbb{E}^n \), \( F : X \to \mathbb{R} \) is a real-valued defining function of point coordinates to represent the point set \( G \), and \( S : X \to \mathbb{R} \) is a real-valued scalar function representing an attribute \( A_i \) that is not necessarily continuous. The set of operations and relations are defined with this representation.

Texturing 3D objects (particularly implicit surfaces and FRep solids) is available using functionally hypervolumes. Both geometry and attributes are defined by FRep. The geometric point sets are assigned attributes by attribute functions. When applying solid texturing to an object, one has to create a space partition of the object space, where each subset contains different material property. Then, the shading parameters are computed for each point in space, where its position determines which subset it belongs to, and eventually the corresponding attributes are determined. A constructive tree which includes space partitions of geometry and attributes is useful for defining subsets. It is called a Constructive Solid Texturing tree.

In the extended HyperFun specification [1], the multidimensional array \( s[] \) is used for attributes. We show an example of a colored solid ball in Fig. 1 and Fig. 2, which are screenshots taken in our new modeler. Fig. 1 is the half-purple, half-yellow ball. The scooped ball in Fig. 2 shows that the attributes are applied not only on the surface but also inside the entire volume. The model of the colored solid ball with the array \( s[] \) holding RGB component attributes is as follows:
my_model(x[3], a[1], s[3]) {
    s = [0.5, 0, 0.5];
    if(x[3] > 0 ) then s = [1, 1, 0];
    endif;
}
Chapter 4

Interactive FRep modeler

4.1. Specification of an interactive FRep modeler

HyperFun is designed to be as simple as possible in order for non-specialist users to create models of complex geometric objects. Also users can use many flexible primitives in the HyperFun FRep library. Although, as mentioned in the Introduction, there is a problem with modeling complex FRep models with current HyperFun tools. Generally, HyperFun users use built-in primitives such as spheres, blocks, convolution objects, ellipsoids, etc., to create their own models. Additionally transformations (blending, twisting, tapering, etc.) are also applied to modify or create more complex models. If users can operate FRep primitives directly in a 3D view, intuitiveness and interactivity are attained.

The HyperFun parser parses HyperFun codes, then the interpreter returns values of the function at given points. The HyperFun polygonizer employs these values for generating a polygonized model and most HyperFun tools display polygonized models generated by the HyperFun polygonizer for real-time rendering. This polygonized model is not suitable for interactive modeling because polygonization takes a lot of time and there are no controllable points. In order to parse primitives and display primitive's controllable skeletons, new a parser is needed.

The FRep library contains functions representing geometric primitives and transformations. Each function has its own set of arguments, all of which are functional expressions. For example, a sphere mentioned in the Introduction can be written as:

```plaintext
my_model(x[3], a[1]) {
center[3];
center = [0,0,0];
my_model =hfSphere(x,center,5); }
```

where hfSphere expresses a primitive object of a sphere (solid ball). Each primitive has its own name. If the parser finds a primitive, then it seeks arguments.
The parser informs interactive FRep modeler about primitives’ information (types, arguments, etc.). The interactive FRep modeler displays primitives' skeleton using this information.

The outline of the interactive FRep modeler features is the following:

**Basic elements.** The modeler is based on the FRep and supports all its basic elements
- primitives
- operations
- relations
- attributes

**New parser.** This new parser parses primitives written in HyperFun codes.

**Extendable user interface.** Users can add convolution surface primitives to the modeler automatically without rewriting its source codes when primitives are added to the FRep library.

**Multimodal user interface.**
- graphical widgets
- construction tree view for geometry and all the attributes
- text editor
- command line

**Functional modules for the modeler.**
- real-time graphical display
- interactive generator of 2D and 3D meshes
- text editor
- graphical widgets for 3D manipulation and editing
- import of existing HyperFun objects
- export of created HyperFun objects.

### 4.2. Interactive heterogeneous volume modeler

We implemented an interactive heterogeneous volume modeler as the first prototype of the general interactive FRep modeler. We employ convolution surface primitives and skeletal primitives of the FRep library.

**4.2.1. Modeler procedures and system architecture**

First, the modeler reads new primitive data file format "hfcm" which is
mentioned in a later section. Second, the user operates with the skeletons and attributes. At this step, the modeler outputs HyperFun model source in the text area and the user can obtain polygonized model. Finally, the user saves created models to the primitive data file and HyperFun file.

The system architecture is shown in Fig. 3.

\[\text{Figure 3. System architecture}\]

### 4.2.2. Convolution Surfaces

\[\text{Figure 4. Skeleton of a Convolution Surface}\]

\[\text{Figure 5. Convolution Surface}\]
Convolution surfaces are one type of implicit surfaces. They are defined by skeletal elements and kernel functions. Fig. 4 is a polygonal mesh skeleton of a convolution surface, and Fig. 5 is the convolution surface made from the skeleton.

Each skeletal element can have weight (kernel width) and threshold. Convolution surfaces and their skeletons enable users to operate interactively similar to modeling of polygonal meshes.

4.2.3. Primitive data file

We should prepare new simple file format for our interactive heterogeneous volume modeler because it can not directly parse HyperFun codes as skeleton data source. The new primitive data file "hfcm" consists of primitive type, points, indices (depends on primitive type), and attributes.

4.2.4. Polygonization and rendering

The modeler is constructed on the basis of the HyperFun project tools. It includes the interpreter and the polygonization engine, which returns vertex coordinates, indices, and color information for each vertex. The modeler employs this information for real-time rendering with OpenGL. With this feature, users can render their modeled heterogeneous volumetric objects whenever they want to see.

4.2.5. 3D views

Current HyperFun tools have only one 3D view. On the other hand, the modeler has four 3D views. This provides comfortable interaction to users. Upper left window is a perspective view and other three windows are orthogonal views arranged in the order of the cube unfolding to the plane.

4.2.6. Platform independency

The modeler should be a platform independent tool so that many users can use the modeler. Hence we employ cross-platform 3D API OpenGL, and the GUI toolkit FLTK.

4.2.7. Export to various file formats

There are various file formats for 3D objects such as VRML, STL, Autodesk
DXF, and vlib [7]. Exporting these file formats is important for users' comfort. Compatibility between heterogeneous volumetric objects modeled in HyperFun language and vlib is especially optimal. Maeda developed software which exports heterogeneous volumetric objects written in HyperFun to vlib [8].
Chapter 5

Implementation

In this chapter, we show implementation methods of the interactive heterogeneous volume modeler.

5.1. FLTK

For building cross-platform 3D software with GUI, we chose FLTK as a GUI toolkit. FLTK is a cross-platform C++ GUI toolkit for UNIX/Linux, Microsoft Windows, and MacOS X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL and its built-in GLUT emulation.

We compared FLTK with Fox Toolkit and Tcl/Tk. FLTK is more active project than Tcl/Tk, and more simple and compatible with OpenGL than Fox Toolkit.

5.1.1. Building FLTK libraries

FLTK has several libraries and header files. These libraries and header files are necessary to build a GUI application with FLTK. FLTK's source codes and documents are available at the official web site of FLTK. Not only source codes and documents, but also makefiles and project files for Borland C++, Microsoft Visual C++ 6.0, Visual C++ .Net, and others are included. Also there are many demo programs for learning FLTK.

In order to build FLTK 1.1.6 libraries on Visual C++ 6.0, open "fltk.dsw" file in "visuale" directory first. This workspace includes many demo programs and the FLTK libraries. Next, open "set active project" window from "Build" menu in menu bar. Select "demo - Win32 Release," then build it. Visual C++ 6.0 builds all demo programs and FLTK libraries. FLTK libraries "fltk.lib," "fltkforms.lib," "fltkgl.lib," "fltkimages.lib," "fltkjpeg.lib," "fltkz.lib" will be in "lib" directory, and "fltpng.lib" will be in parent directory. If some errors occur, check the project's settings. Probably there is one miss in project settings of "fltkdll." At ignore list of
libraries in "Link" tab, remove "msvcr" because it is necessary to build FLTK's DLL.

Next, select "demo - Win 32 Debug" and build it. Debug version of FLTK libraries and demo programs are built. With this debug version of libraries, we can gain debug information and a console window for debug while program with FLTK runs.

After building libraries, put them to Visual C++ library directory. In addition, put "FL" directory which includes header files to Visual C++ "Include" directory.

5.1.2. Developing 3D software with FLTK

In order to develop 3D software with FLTK in Visual C++, make an empty work space and projects first. Then open setting window from "Project" menu in menu bar. At "Link" tab, add fltk.lib fltkgl.lib glu32.lib opengl32.lib to object/library modules for Win32 Release, and fltkd.lib fltkgld.lib glu32.lib opengl32.lib for Win32 Debug. These are all of preparations for developing 3D software with FLTK on Visual C++.

There are three ways for developing it - GLUT emulation of FLTK, Fl_Glut_Window class, and Fl_Gl_Window. We chose the Fl_Gl_Window class because combination of FLTK and Fl_Gl_Window is better than other ways.

With Fl_Gl_Window class, all OpenGL windows is subclass of the Fl_Gl_Window class. These windows are redrawn by "redraw()" methods. When "redraw()" is called, it calls "draw()" methods. These methods should be called while program runs.

5.2. HyperFun tools

There are some HyperFun tools - HyperFun for Windows, HyperFun Applet, HyperFun polygonizer, and others. Each is constructed with its own program project, interpreter, and polygonization engine. Our modeler is based on the HyperFun polygonizer technology with attributes. In the Microsoft Windows system, we developed the modeler using Microsoft Visual C++ 6.0.

HyperFun for POV-Ray and HyperFun ray tracer use ray-tracing technique for rendering FRep models. On the other hand, other HyperFun tools use polygonized models for real-time rendering.
5.3. Interactive modeler

We made the modeler with FLTK and the HyperFun polygonization engine. The whole program is written in C++. The modeler has four main classes - "reading hfcm file and outputting HyperFun code class", "polygon model generator class", "perspective window class" and "orthogonal window class." In addition, many classes and methods for buttons, sliders, mouse control, text window, callbacks, and setting windows are defined. Fig. 6 is a screenshot of the modeler.

Figure 6. Screenshot of the modeler.

In this section, outline of modeler’s behavior at each modeling step is described. Here, we make a list in the order of a typical modeling session by the user - opening files, modeling, watching polygonized model, and saving files.
5.3.1. Opening file and drawing skeletons of convolution surfaces

After the program starts, the user makes a new object or opens an existing file. The modeler can open the primitive "hfcm" file. The modeler reads primitive type, vertex data, index data, and attribute data from the file. Then, the skeletons of the convolution surfaces are drawn in four 3D views.

5.3.2. Modeling

Controllable points are drawn for each vertex of the convolution surface. Users can move these points or add points to modify objects. Also attributes (one color or preset attributes) are applicable.

5.3.3. Generating HyperFun code

The modeler generates HyperFun code when the user pushes "To HF" button. The generated HyperFun code is outputted to the text area. This code is used for the polygonization and exporting HyperFun files. The user can edit the HyperFun code in the text area and polygonize the code. However, when the "To HF" button is pushed, the edited code will disappear because the modeler ignores what is written in the text area.

5.3.4. Polygonal models generation

The modeler employs the HyperFun's new polygonization engine which can calculate attributes at the vertices of the generated mesh. The "polygon engine class" and the "interpreter class" are defined in the "polygon model generator class." When the user pushes "Polygonize" button, the new "polygon model generator class" object is created. Then the class gives to the interpreter the HyperFun code written in the text area. If there are any errors, an alert window will appear. If no errors, data of the polygonized heterogeneous volumetric object is created in the "polygon engine class." With this function, users can see results of their heterogeneous volumetric object modeling whenever they want to.

5.3.5. Rendering 3D objects with OpenGL

Skeletons of convolution surfaces, axes, bounding box, and the polygonized heterogeneous volumetric object are rendered by OpenGL in real-time. The modeler's renderer reads vertex data from the "polygon engine class." Each vertex
has data of attribute color. The renderer sets the color as material for each vertex, then renders polygon objects.

5.3.6. Saving files

The modeler can save the created model to "hfcm". Also, exporting "hf" (HyperFun code file) is available. The HyperFun code written in text area is used for exporting "hf" file.

5.4 Program Structure

In this section, we explain the program structure.

5.4.1. Source files and header file

Our modeler's program is constructed on HyperFun Polygonizer technology and its project for Microsoft Visual C++ 6.0. The program uses not only FLTK and OpenGL libraries but also HyperFun's C++ library, primitive library, interpreter library, and polygonization engine library.

The program consists of seven C++ source files and one header file. In the header file, class definitions, external global variables and instance objects, including files, prototypes, and macros are defined. On the other hand, substances of classes are defined in source files. All source files include the header file.

5.4.2. Main method

The main method initializes some class objects first. Second, it calls "makeform" method which constructs form of GUI. An instance object of the main window is created, and then instance objects of other windows, 3D views, buttons, sliders, menu bar, and others are thrown into the main window. Third, the main method initializes values of sliders, buttons, and windows.

After that, the main method runs a "while" loop which involves "Fl::wait()" method. This method waits for any event of the program and keeps running the program. In addition, the "while" loop involves "redraw()" methods for each 3D view and events of buttons and sliders. While the program is running, all 3D views are redrawn whenever any event for 3D views happen.
5.4.3. Menu bar

In FLTK, the menu bar is handled as a group of callbacks. FLTK employs the array of Fl_Menu_Item for the list of the menu bar. This array involves callback indicators of each menu. When the user selects a menu item, a callback method which is made for the menu is called.

We show one example of callback methods. The "Attribute" menu sets the option for the modeler of using a polygonal model with attributes or a polygonal model without attributes. The "Attribute" menu calls "attribute_cb" method. All callback methods of FLTK receive "Fl_Widget" and void (for any type of variables) as arguments.

First, the "attribute_cb" method creates an instance object of the window class. The window class for "attribute_cb" method involves some variables for attribute settings. Second, the method puts two buttons into the window. Then the method shows the window. When the user pushes the "Apply" button, the method calls a method which is made for applying the button as a callback method. The "Attribute apply button" method receives Fl_callback and the window's class which is created by "attribute_cb". The window class is used for setting variables and closing the window after the "Apply" button is pushed.

5.4.4. Reading the hfcm file

An example of the hfcm file is following:

```
primitive [ mesh ]
point [ 8.0 0.0 0.0,
  0.0 8.5 0.0,
  ...
  0.0 -8.5 0.0,
  0.0 0.0 -9.0 ]
coordIndex [ 1 2 3,
   1 2 6,
   ...
  4 3 5,
  -1 -1 -1, ]
color [ 0.5 0.5 0.5, ]
```
The modeler uses the "reading hfcm file and outputting HyperFun code" class (HFCM_data class) for reading the hfcm file. When the user chooses "Open File" from the menu bar, a new instance object of the class is created. After that, the parser method which reads the hfcm file is called.

The parsing method uses keywords for finding data. If the method finds a keyword such as "point", "coordIndex", it starts to read data. The read data are put into variables in the "HFCM_data" class. Also some counters for the number of primitives, indices are counted. These data are used for many purposes of the program.

5.4.5. Generating HyperFun source codes

A HyperFun source code shown below is a result which was created by the modeler's HyperFun source code generator from the hfcm file mentioned in 5.4.5.

```
my_model(x[3], a[1], s[3])
{
array vect[18];
array tri[21];
array weight[7];
vect = [ 8.000000, 0.000000, 0.000000,  
      0.000000, 8.500000, 0.000000,  
      0.000000, 0.000000, 9.000000,  
      -8.000000, 0.000000, 0.000000,  
      0.000000, -8.500000, 0.000000,  
      0.000000, 0.000000, -9.000000 ];

tri = [ 1, 2, 3,  
       1, 3, 5,  
       1, 5, 6,  
       4, 2, 3,  
       4, 2, 6,  
       4, 3, 5 ];

weight = [4.000000, 4.000000, 4.000000, 4.000000, 4.000000, 4.000000, 4.0];
s = [0.500000, 0.500000, 0.500000];

mymesh = hfConvMesh(x,vect,tri,weight,2.0);

my_model = mymesh;
```
The HyperFun source generator outputs a source code to the text buffer of an instance object of the editor window class (text area) when the user pushes the "To HF" button.

The process of the generator is as follows. First, the generator distinguishes which kind of primitive is used. In this section, we assume the type of primitive is a convolution object. Second, the generator erases all text in the text area. Third, the generator outputs several sentences to a temporary string array. These sentences involve some variables. For example,

```
array vect[18];
array tri[21];
array weight[7];
```

these declarations involve three variables: vect[18] is an array of vertices, tri[21] is an array for connecting triangle vertices, and weight[7] is a convolution parameter for triangles. These variables are counted and recorded by HFCM_data. Fourth, the generator adds these temporary strings to the text area. The HyperFun source generator repeats third and fourth processes.

5.4.6. Polygon model generator

The outline of the polygon model generator is given in 5.3.4. First, the polygon model generator employs the HyperFun interpreter. The interpreter parses the HyperFun source code written in the text area. We can employ the interpreter's library as a class. An instance object of the class is created when the new instance object of the "polygon model generator" class is created. If there are errors, the polygon model generator receives error messages from the interpreter and displays an alert window using the FLTK's alert window method. The type of the message from the interpreter is "STL string." However, the FLTK's alert window method can handle only "char." The conversion with "c_str()" is required.

Second, the polygon model generator employs the HyperFun polygonization engine. We can employ this engine as a class too. Before the polygonization engine starts, the polygon model generator initializes variables which are used in the polygonization process. Then, it starts to create a polygon model. Data of vertices, normals, indices, colors are put to the polygon model generator.

There are two Boolean variables to check the polygon model generator - "polygonerror" and "polygoncheck"; "polygonerror" is used for the process of polygonization. If there are errors, "polygonerror" switches to true, and later
process doesn't run; "polygoncheck" switches to true when the process of polygonization succeeds.

5.4.7. 3D views and drawing methods

The "redraw()" methods for each Fl_Gl_Window class object (3D view class) in the main method call "draw()" methods declared in the 3D view class. This means each 3D view class must declare the "draw()" method. The drawing method must be named "draw()." The 3D views class for the perspective window and the orthogonal windows are almost the same, only the projection setting is different.

The drawing method's source codes are similar to general OpenGL program source codes. First, it sets view port, lights, depth buffer, projection matrix, then calls drawing meshes methods. The GLUT's perspective method is used for perspective window's projection. On the other hand, GL's orthogonal method is used for orthogonal window's projection.

5.4.8. Operations with mouse and zoom sliders

In 3D views, users can rotate a model in the perspective window and move camera's position in orthogonal window with dragging the mouse. Also, the user can handle zoom up/down with zoom sliders.

Mouse events are handled by FLTK's "handle()" method. This method has to be declared in each 3D view class. The name must be "handle()."

Rotating objects with dragging the mouse in the perspective view is done by the following procedure. First, when the user pushes the mouse button to drag, the program records window's coordinate of the pushed point:

```
clickedpoint_w = Fl::event_x0;
clickedpoint_h = Fl::event_y0;
```

Second, the program saves current degree of rotation:

```
savedrot_x = rotation_x;
savedrot_z = rotation_z;
```

Degree of Y isn't saved because it doesn't change. Third, when the user starts dragging, the program starts to calculate the degree of rotation:

```
rotation_x = ( Fl::event_y0 - clickedpoint_h ) + savedrot_x;
rotation_z = ( Fl::event_x0 - clickedpoint_w ) + savedrot_z;
```
where the rotation_x and rotation_y variables are used in the drawing method.

Moving camera's position with dragging the mouse in the orthogonal view is done by the following procedure. In this section, we assume the case of the top view. First, when the user pushes the mouse button to drag, the program records window's coordinate of the pushed point:

\[
\begin{align*}
\text{orthoclick}_w &= \text{Fl}::\text{event}_x(0); \\
\text{orthoclick}_h &= \text{Fl}::\text{event}_y(0);
\end{align*}
\]

Second, the program saves the current position of the camera:

\[
\begin{align*}
\text{f_savemv}_\text{width} &= \text{f_widthmovevalue}; \\
\text{f_savemv}_\text{height} &= \text{f_heightmovevalue};
\end{align*}
\]

Third, when the user starts dragging, the program starts to calculate the degree of rotation.

\[
\begin{align*}
\text{f_widthmovevalue} &= \text{f_savemv}_\text{width} + (\text{Fl}::\text{event}_x(0) - \text{orthoclick}_w); \\
\text{f_heightmovevalue} &= \text{f_savemv}_\text{height} + (\text{Fl}::\text{event}_y(0) - \text{orthoclick}_h);
\end{align*}
\]

where the f_widthmovevalue and f_heightmovevalue variables are used in the drawing method. The model does not move. However, "the orthogonal viewport box", namely camera's position, is moved by changing these values.
Chapter 6

Experiments

Here we show an example of a heterogeneous volumetric object created by the modeler. Fig. 7 is a screenshot which is taken in our modeler. This model is one object which has forty vertices of the convolution primitive with the mesh skeleton. One of modeler’s preset attribute is applied to the model. Fig. 8 is a voxelized object image rendered using the vlib volume graphics API.

Figure 7. Screenshot taken in the modeler

Figure 8. Image rendered by vlib
Chapter 7

Conclusion and future work

We analyzed existing interactive volume modelers, specified an interactive FRep modeler for heterogeneous volume objects, and succeeded in developing a prototype modeler based not on the discrete voxel model, but on continuous functions both for geometry and attributes. However, there are problems we should solve and future works.

7.1. Technical problems

First, we will modify movement of points and objects because movement of points and objects doesn't match mouse drag. Also limitation of adding points and objects should be modified. This problem is caused by the data structure in our program. Second, the modeler has memory leak problems while creating models. Third, rendering speed is an important matter. The program redraws all windows (one perspective view, three orthogonal view, and a text editor window) while program runs. Fourth, the modeler cannot show the progress of polygonization. The percentage value of progress is available only in the polygonization engine, so the modeler cannot obtain this value. Fifth, we have to improve text editor for comfortable editing.

7.2. Future works

The next step is to implement modeling of time-dependent and higher dimensional objects. It means that the modeler will be able to create hypervolume objects and render animation.

In its full capacity, the modeler should support all primitives and operations of HyperFun. Furthermore, automatically extensible user interface for new primitives similar to Goto's work [5] is needed.
A constructive tree viewer is a useful component to add. This viewer visualizes the constructive trees for geometry and each of attributes. Supporting immersive navigation and carving [9] will provide higher power of expression.
References


