A SCRIPTING INTERFACE FOR DOUBLY LINKED FACE LIST BASED POLYGONAL MESHES

A Thesis

by

STUART TOSTEN TETT

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2007

Major Subject: Visualization Sciences

A SCRIPTING INTERFACE FOR DOUBLY LINKED FACE LIST BASED POLYGONAL MESHES

A Thesis

by

STUART TOSTEN TETT

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2007

Major Subject: Visualization Sciences

ABSTRACT

A Scripting Interface for Doubly Linked Face List Based

Polygonal Meshes. (December 2007)

Stuart Tosten Tett, B.S., Iowa State University

Chair of Advisory Committee: Dr. Ergun Akleman

This thesis presents a scripting language interface for modeling manifold meshes represented by a Doubly Linked Face List (DLFL). With a scripting language users can create procedurally generated meshes that would otherwise be tedious or impractical to create with a graphical user interface. I have implemented a scripting language interface for the user to create stand-alone scripts as well as script interactively within a graphical environment.

ACKNOWLEDGEMENTS

Firstly, I would like to acknowledge all of those who contributed to TopMod. Without them, this thesis would not have been conceived. The lead developer, architect, and programmer of TopMod is Vinod Srinivasan. The concept was developed by Ergun Akleman. He and Jianer Chen developed the theoretical framework. Key contributors include Esan Mandal, Eric Landreneau, Zeki Melek, David Morris, Michael Stanley, Ozgur Gonen, and Brian Barran. Other contributors include Paul Edmundson, Fusun Eryoldas, Cansin Evrenosglu, and Xu Bei.

I would like to give special thanks to David Morris who took the initiative to design the new graphical interface for TopMod. The new interface was crucial to the development of my thesis. He has worked on TopMod along with me supporting my work and critiquing it.

I would also like to thank my committee. My committee chair, Ergun Akleman, has helped me to have a thorough understanding of the theory behind the Doubly Linked Face List structure and the minimal set of operators for manifold modeling. Also, thanks to my other committee members, Donald House and Daniele Mortari for their interest and support.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION AND MOTIVATION

A Doubly Linked Face List (DLFL) is an efficient data structure used to store 3D mesh data. The DLFL data structure is a representation of a graph rotation system, guaranteeing that the mesh is manifold [Akleman and Chen 2000]. A mesh is manifold if all of its edges belong to no more than two faces. This property is necessary for the mesh data to be physically realized by a 3D printer. This property is also important for working with subdivision surfaces.

DLFL uses linked lists for a mesh's faces, edges, and vertices. An additional type of mesh data is also stored as a linked list: corners. Corners represent two consecutive edges and the vertex connecting them. In most cases this corresponds to a face-vertex pair[1]. A simple illustration of the data can be seen in Figure 1. This example represents a tetrahedron. The items are labeled 'v' for vertices, 'e' for edges, and 'f' for faces. The arrows represent the links (pointers) between the data.

In order to construct models with the DLFL structure there exists a complete set of minimal operators. The operators include Create-Vertex, Remove-Vertex, Insert-Edge, and Delete-Edge [Akleman et al. 2003a].

Using this DLFL data structure a topological mesh modeling system, TopMod, was created in 2000. The application is implemented in C++ and provides a 3D viewer for modeling inside a graphical user interface. TopMod is well-suited for creating high-genus 2-manifold meshes. It has many operations built upon the minimal operators. These operations include different types of face extrusions, remeshing/subdivision schemes, and high-genus operations to create holes and handles.

Prior to this thesis, most users could not program their own procedures for TopMod. Most users

This thesis follows the style of *ACM SIGGRAPH Computer Graphics.*

[1]See the "Conclusion and Future Work" chapter for discussion of the cases where this is not valid.

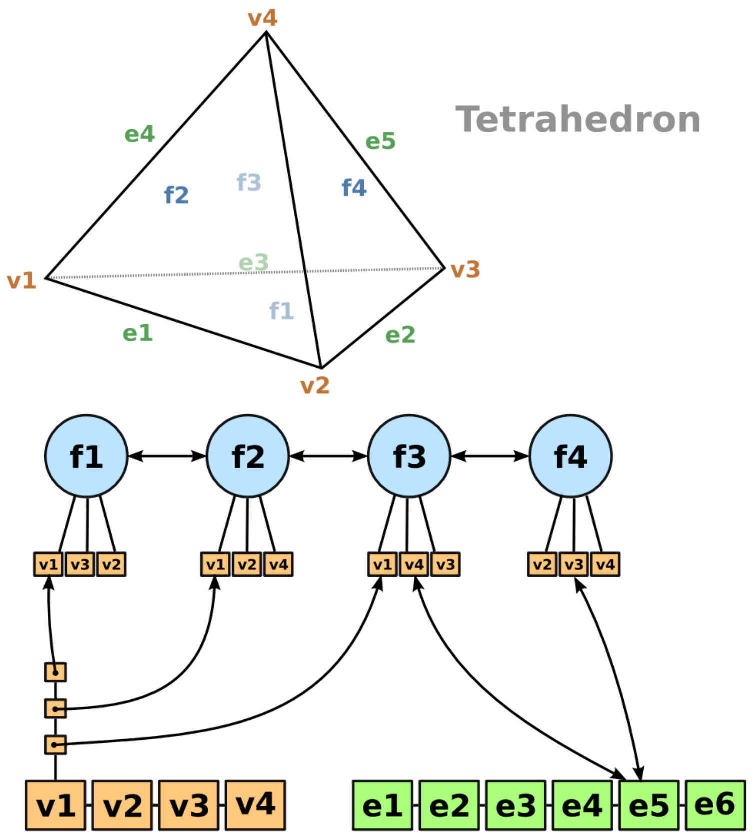


Fig. 1: Example of DLFL structure for a tetrahedron

do not have access to the source code. Even if a user has access to the source code, it is difficult for them to add their own procedures. The TopMod source code consists of a large number of files and it takes a significant amount of time to acquaint oneself with its structure. Many TopMod users are novice programmers (they have little or no programming experience). For those few users that do have the access and the ability to contribute to the TopMod source code, it is not practical to edit the code for every little task. It takes time to expose the new code to the graphical interface and also to compile the code. Adding several of these custom tasks would cause the code to become bloated very quickly.

This inability to write code in TopMod limits users. The users cannot perform repetitive or logical tasks efficiently. 3D meshes can be constructed/manipulated with a sequence of operations. Operations that are performed several times can be condensed into a loop. A loop simply says: "perform this several times." Without code, a user has to manually perform that operation each time. Working this way is tedious for the user, especially when the user wants to alter the operations parameters for each execution.

Users cannot control the parameters of TopMod's operations with functions. For example, 3D modelers often want to control an aspect of a mesh by a mathematical function. Useful mathematical functions include Sine, Cosine, Random, Maximum, Minimum. By using these functions along with the standard arithmetic operators, modelers can use their creativity and mathematical knowledge to create interesting shapes.

In addition, users do not have access to low-level operations. While TopMod's graphical interface provides access to the most useful operations, it has functions in the source code which are not exposed to the user. These operations serve as helpers to the high-level operations. Hiding access to these operations keeps the software interface simple. This is desirable for most users. However, hiding these operations also limits the user's creativity.

In computer graphics packages, all of these features are useful. Users often perform tasks that follow a logical pattern in order to achieve a certain data set (i.e., an image, a mesh). The pattern could be as simple as performing the same task multiple times. It could also be much more complex, but still be fashioned from a series of logical steps. Performing these monotonous tasks by hand is inefficient and often impossible in a reasonable period of time.

Many computer graphics packages also give the user the ability to control operations' parameters with functions. For example, most animation packages allow you to assign the rotation, translation, or scale of an object to a mathematical function such as a sinusoidal wave. Describing animation with a function is an alternative to explicitly setting each frame by hand.

The most powerful computer graphics packages allow access to some important low-level operations. Access to these is generally hidden from basic users, but is available for experienced users interested in utilizing the potential of the software.

The goals of this thesis are to provide a solution to these problems. TopMod users should be able to write their own code. Since contributing to the source is not an option for users, this feature should be provided via alternative means.

TopMod users should be able to efficiently perform repetitive tasks. If a user wants to extrude a face fifty times, for instance, then the amount of work to do this should be minimal. The users should not have to perform fifty mouse clicks. Users should be able to control the parameters with functions.

A user should not have to edit the parameters of an operation by hand to conform to some function. This is extremely inefficient. If the user desired that the fifty extrusions scale conform to a sinusoidal wave, then the user should not have to use hand calculation or external software to compute the parameters.

Users should have access to useful low-level operations. Users should not be limited by the graphical interface. For example, a user may need to perform a subdivision scheme unavailable in TopMod. The user should be able to access TopMod's low-level operations in order to perform that scheme.

Scripting is an effective solution to these problems with TopMod. Scripting languages are computer programming languages that are interpreted (as opposed to compiled) and can be used interactively from a keyboard. Scripting allows users to execute a series of commands utilizing computer logic. Quick feedback from interactive scripting facilitates rapid prototyping.

One advantage of scripting languages is that users need not be experienced programmers. Scripting enables users with domain knowledge to perform programming-like tasks without having knowledge of the underlying program structure. Scripting languages are designed for readability. They more closely resemble the way humans communicate. For this reason, scripting has become a very powerful tool for software designed for a non-programmer constituency.

Non-programmers can easily use the highly readable scripting languages. The most highly readable scripting language might read, “repeat this command twenty-five times.” Although scripting languages aren’t quite this close to English, they are almost as easy to understand.

In the hands of an experienced programmer, a scripting language can yield much more power. While an interpreted language cannot run as fast as a compiled one, this is not where the “power” comes from. The power comes from the interactivity. The scriptor can develop quickly within the environment and receive very quick feedback. The scripting language extends the software for the experienced programmer beyond what the average user can achieve with it.

Many computer graphics packages have their own specific scripting language. Autodesk Maya, for example, is powered by the Maya Embedded Language (MEL). With MEL, users can script modeling, rendering, animation, and custom interface dialogs. Likewise, Autodesk 3D Studio Max has MaxScript and Side Effects Houdini has HScript. These custom scripting languages were most often developed because no existing language existed that could fulfill the needs of the software when it was developed.

With the increase in quantity and power of existing scripting languages, more recent packages make use of an external scripting language. Scripting languages usually have much extensibility with which developers can create bindings to that package. Rhinoceros, a 3D modeling tool for Windows, uses VBScript. Another example of an external scripting language is in the open source 3D content creation suite, Blender. In Blender, users script with Python, an increasingly popular language for scripting. Python is also used in the computer graphics package Softimage XSI, and has been adopted recently in Maya 8.5. This gives Maya users a choice in their scripting language.

Using an external scripting language like Python has many benefits. Users that are already familiar with Python would not have the hurdle of learning a new language. Users that are new to Python would be learning a popular language that they could use with many other programs.

It is also beneficial on the development side. Creating even a simple language is extremely difficult and requires significant development resources. Using an existing scripting language allows developers to forego having to design the language structure, building an interpreter, and debugging the system. Instead, the developers can concentrate on how the scripting system is going to interact with the software. This is a much more important part in developing a powerful scriptable application. Furthermore, quality scripting languages exist. It is a waste of good resources to essentially duplicate code in existence.

Supplementing the graphical user interface with a procedural scripting interface is also important in allowing the user to manipulate deep structure within the software. The graphical user interface has a high compatibility with input and output devices. Graphical interfaces present the user with graphical metaphors (such as icons). These provide a straightforward way in which the user can interact with the software via mouse and/or keyboard, and receive feedback from the display.

It also has a high compatibility with manipulation of surface structure. It is very efficient for users to work with both 3D and 2D data in this way. The mouse is well suited to manipulate vertices, faces, and other data – especially on a local level.

However, the graphical user interface has a very low compatibility with the manipulation of deep structure [Fitzmaurice and Buxton 1998]. Significant research has been done to offer a better deep structure manipulation via the graphical user interface. Maya's Hypergraph is an effective visual representation of deep structure (Figure 2). The graph represents scene data and illustrates the scene's hierarchy. It also shows how data has been manipulated by operators.

However, the power of scripting still outweighs that of these interfaces. This is why Maya and other computer graphics software still provide user programability. While scripting has a low compatibility with manipulating surface structure, it has a very high compatibility with manipulating deep structure. Scripting provides efficient means for performing global operations on mesh data. It also provides access to the underlying structure of the system.

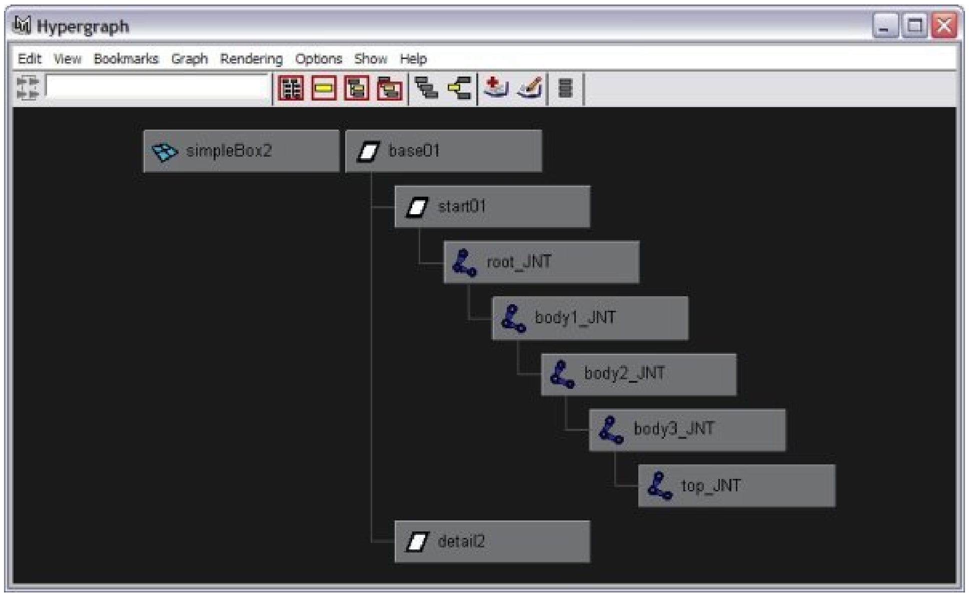


Fig. 2: Example of data visualized by Maya's Hypergraph

CHAPTER II

PRIOR WORK

Topological Modeling

I have developed a prototype scripting language to create manifold meshes represented by the DLFL data structure [Akleman and Chen 2000], a conceptual framework that guarantees creation of topologically correct 2-manifold meshes. Using the DLFL data structure, Akleman, Chen and Srinivasan introduced a minimal set of operators for the development of robust manifold mesh modeler [Akleman et al. 2003a]. The minimal set of operators are INSERT-EDGE, DELETE-EDGE, CREATE-VERTEX, and REMOVE-VERTEX and using these operators it is possible to create all and only manifold meshes.

TopMod, a robust topological mesh modeling system that allows users to create high genus 2-manifold meshes was developed based on this minimal operator set and the DLFL data structure [Akleman et al. 2003a].

This software is used primarily for architectural and sculptural design. Figures 3, 4, and 5 show examples of models built with TopMod.

Because TopMod guarantees manifold meshes it is well suited for these disciplines; both require that the mesh be physically real. With the technology available to print 3D data with a variety of materials, it is necessary that the data be structured correctly. Other uses such as computer games and film production can make use of this data as well, but it is not usually necessary for meshes to be manifold.

Since the introduction of TopMod, many researchers and students have expanded on its functionality [Akleman et al. 2004a; Akleman and Srinivasan 2002; Akleman et al. 2002; Akleman et al. 2004b; Landreneau et al. 2006; Landreneau et al. 2005; Mandal et al. 2003; Srinivasan and Akleman 2004; Srinivasan et al. 2005]. Because of these contributors and its availability to the internet community, TopMod has become a powerful software and gained attention from users across the globe.

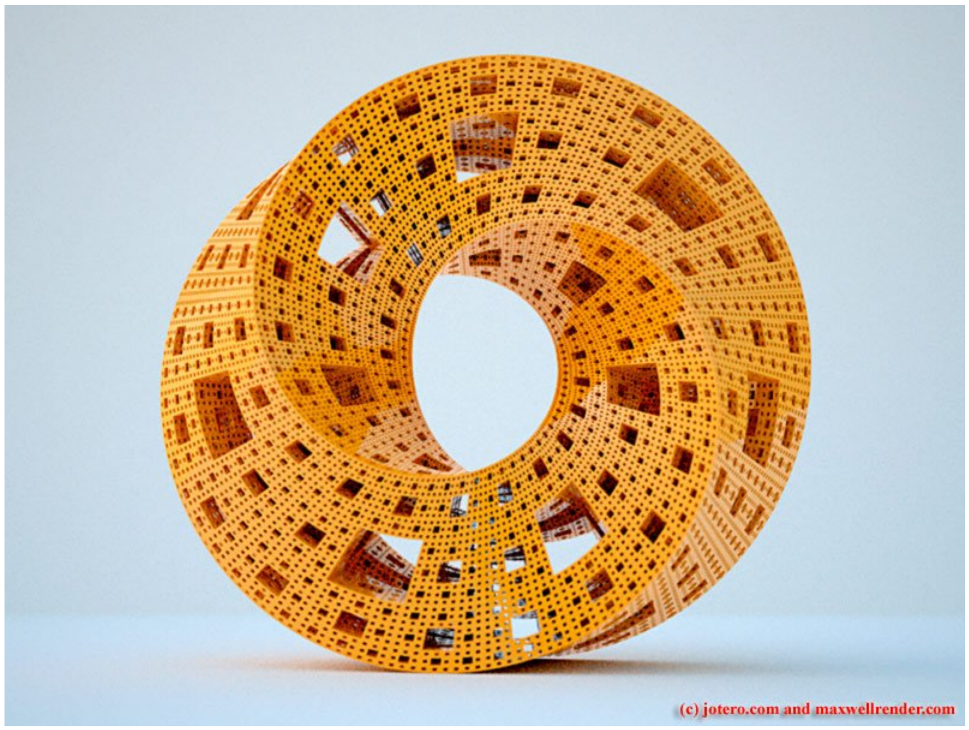


Fig. 3: A render of a sculpture designed with TopMod by Torolf Sauerman

TopMod now provides a plethora of operations built upon the minimal set of operators. These high level operations include extrusion, remeshing, and high-genus operations.

Many different face extrusion methods have been added, such as Doo-Sabin extrusion (Figure 6).

No other known software has as many subdivision schemes as TopMod. Figure 7 shows Catmull-Clark, a common scheme found in 3D software. Most of the other schemes are not found in other software. Schemes include: Doo-Sabin, Honeycomb, Pentagonal, Loop, Checkerboard, Fractal, and many others.

High-genus operations result in meshes with a significant number of holes or handles. For

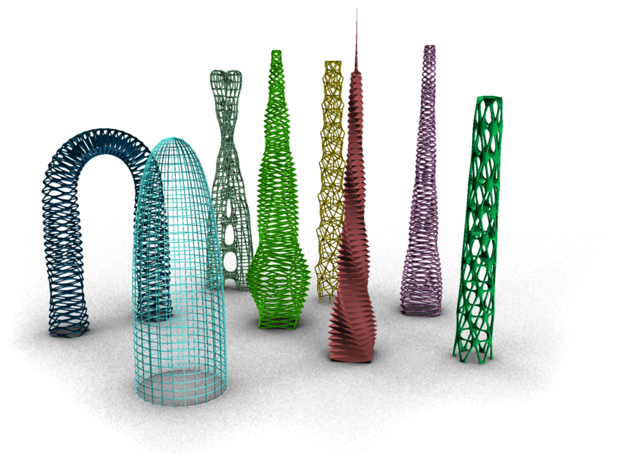


Fig. 4: Several architectural creations by David Morris

example, rind modeling (Figure 8) is the process of converting a mesh into a shell or crust, followed by peeling or punching holes in the crust [Akleman et al. 2003b]. Another example is handle creation. Given two corners (each belonging to a unique face), TopMod can use interpolation to create a handle (Figure 9).

For users to create unique and interesting shapes, many operations require repetitive tasks including face, edge, and vertex selection. For some complex shapes this can require hundreds of clicks. Some shapes that a user envisions could require too many clicks to be feasible. A scripting system could greatly expand on the capabilities of what users could create with TopMod.

Python

My choice for a scripting language is Python. Python was created in 1990 by Guido van Rossum. It is derived from a previous language van Rossum contributed to, called ABC. Python is in the public domain so there are no restrictions on distributing it. It is available on all major platforms (DOS, Windows, Linux, Macintosh, UNIX, etc.) [Lutz 1996].

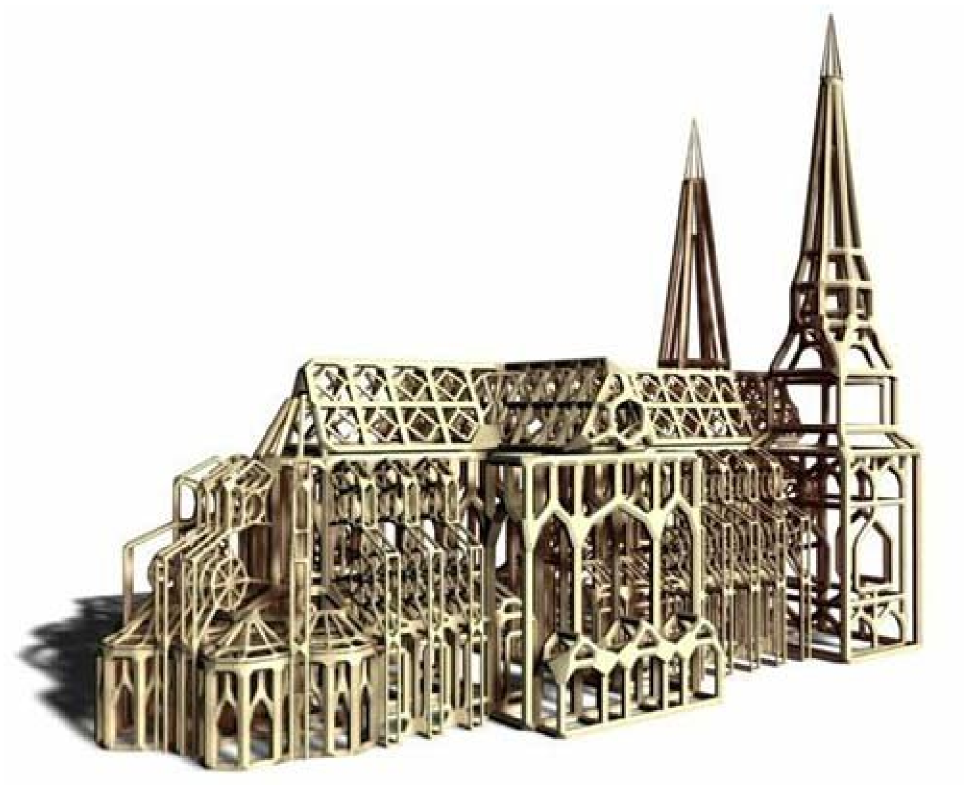


Fig. 5: An architectural model made with TopMod

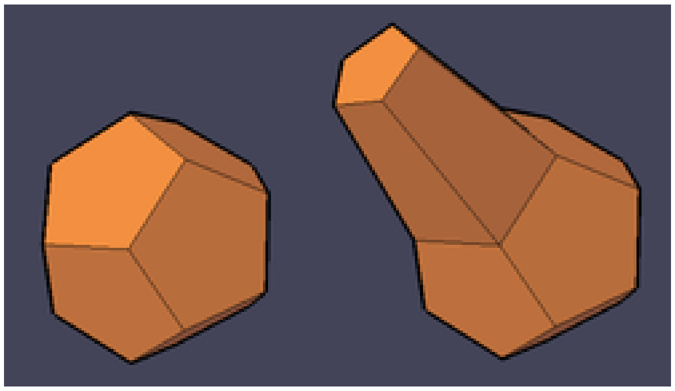


Fig. 6: A simple Doo-Sabin extrusion (Left: before extrusion; Right: after extrusion)

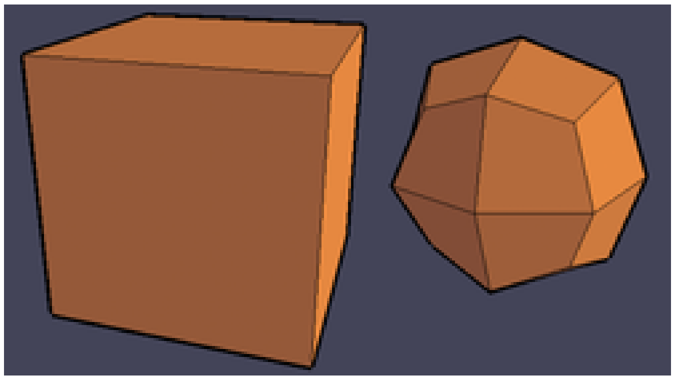


Fig. 7: A simple example of a remeshing scheme using Catmull-Clark subdivision (Left: original cube; Right: subdivided cube)

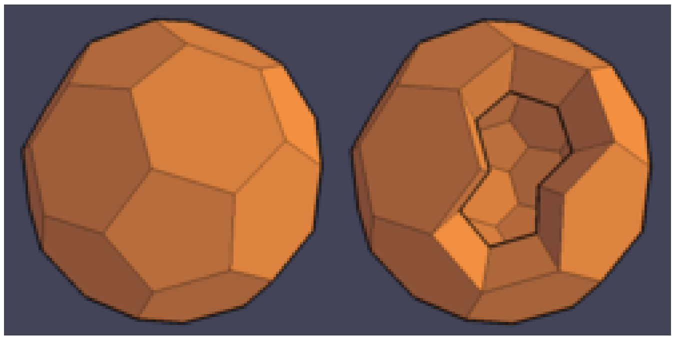


Fig. 8: A simple rinding (Left: before peeling; Right: after peeling)

Python is a strong choice both for stand-alone rapid-development and scripting. As a rapid-development language, Python supports object-oriented programming and has a similar workflow to Java [Lutz 1996]. Python source files can be compiled and often approaches run-time speeds near that of C++.

As a scripting language, no compilation or linking is necessary. Python has automatic memory management and garbage collection. It is well suited for developing system, graphical user interface, database, and networking tools [Lutz 1996].

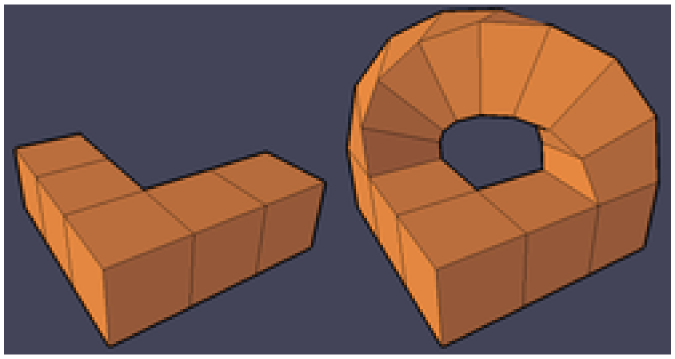


Fig. 9: A simple handle creation (Left: before handle; Right: after handle)

A key feature is the Python C Application Programmers Interface (API). This enables programmers to extend C with Python, by creating bindings to C functions. This also enables programmers to embed Python inside a C program. For instance, creating a built-in Python interpreter inside a C++ program. The existence of this feature was paramount in the feasibility of this thesis.

Python Language Overview

Python's syntax is often considered minimalist and it emphasizes readability. Python requires very little "boilerplate code" (sections of code that have to be included in many places with little or no alteration). For example, compare the minimum code required to print "Hello, world!" in Python (Figure 10) versus C++ (Figure 11).

```
print "Hello, world!"
```

Fig. 10: Python "Hello, world!" code

Python's code syntax paradigm makes it a perfect language for novice programmers. It makes it simple for users to quickly achieve their modeling goals (whether novice or expert).

```
#include <iostream>

int main(void) {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Fig. 11: C++ “Hello, world!” code

Python does not use curly braces like C++ and many other languages. Nor does it use any other character/keyword combination to represent the begin and end of a block. Instead it relies on whitespace. The indentation levels tell Python where a particular statement belongs. After a function definition, all statements that belong inside a function definition are indented in. At the end of the function, the indentation level decreases. See Figure 12 for a clear example on Python syntax.

```
def add56(x):
    return x+5,x+6

for i in range(0,5):
    if( i%2 == 0 ):
        print add56(i)
    else:
        a,b = add56(i)
        print i**2,a
```

```
(5,6)
1 6
(7,8)
9 8
(9,10)
```

Fig. 12: Example Python code with results

Python supports the standard built-in data types that most programming languages do: strings, integers, floating-points, and booleans. For organizing data it has lists, tuples, sets, and dictionaries.

Lists are mutable structures for storing sequences of data. Tuples are similar, but are immutable. Sets are unordered immutable structures useful for removing duplicates from sequences and performing boolean operations (union,intersection,etc.). Dictionaries store mappings between a key and its value. Given a key, the dictionary returns it's value. Figure 13 shows the basic use of dictionaries in Python.

```
>>> mydict = { 'hey' : 24580, 'joe' : 09564, 'jimi' : hendrix }
>>> print mydict
{'jimi': 'hendrix', 'hey': 24580, 'joe': 9564}

>>> print mydict['jimi']
hendrix
```

Fig. 13: An example of a dictionary in Python

Python does not have static typing. A variables type is determined by what it is assigned. However, Python has strong typing. You cannot for example, add an integer and a string together.

Python functions cannot modify the variables passed in as arguments. To compensate, Python has the ability to return multiple variables as seen in Figure 12. It is actually returning the data packed in a tuple. Tuple elements do not have to be of the same data type.

Maya Embedded Language

This scripting system design is modeled after the Maya Embedded Language (MEL). MEL is a powerful, flexible language and has proved its usefulness within a three dimensional modeling environment.

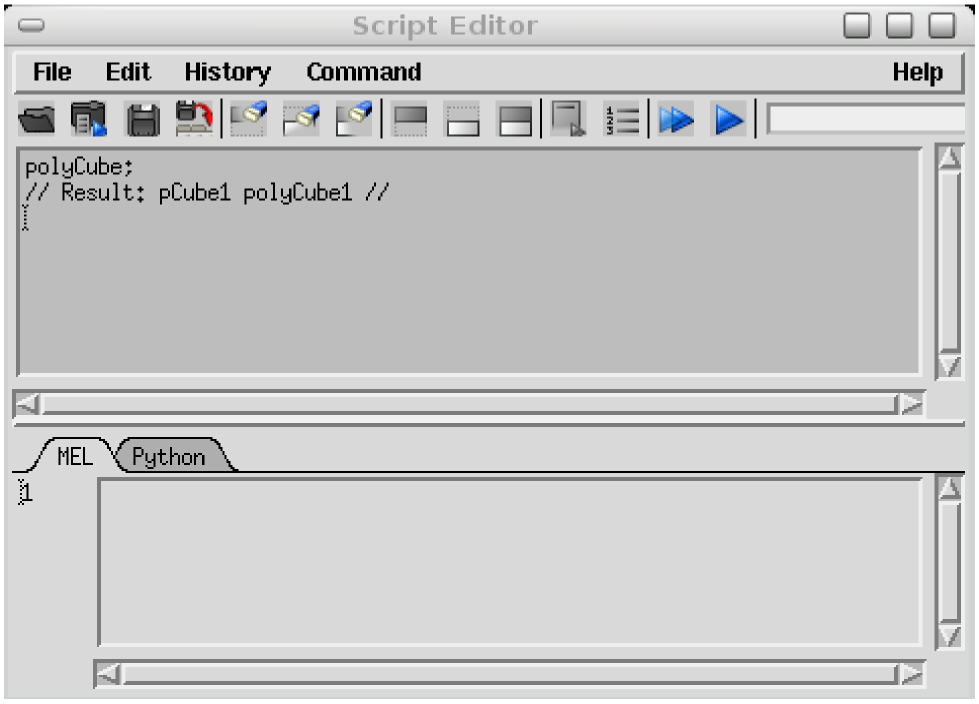
The first version of Maya was released in 1998. In early development Maya used Tcl as its scripting language. After comparing different scripting languages for use in Maya, developers opted to create a custom scripting language. MEL is an interpreted language and is syntactically similar to Perl. See Figure 14 for an example of what MEL code looks like.

```
global proc polyNormalRandomize() {
    string $faces[] = `getFaces`;

    string $fVtx[] = `polyListComponentConversion -ff -tvf $faces`;
    for ($vtx in $fVtx) {
        float $normal[3] = `polyNormalPerVertex -q -xyz $vtx`;
        float $rnd = `rand -0.1 0.1`;
        polyNormalPerVertex -xyz ($normal[0]+$rnd)
                                 ($normal[1]+$rnd)
                                 ($normal[2]+$rnd) $vtx;
    }
}
```

Fig. 14: Sample MEL code which randomizes the normals of all the faces

MEL plays a huge role in the Maya interface. All of the graphical user interface elements are created with it. In addition, Maya's internal settings can be controlled using MEL. For example, the command `window` creates a new window. This can be filled with all kinds of controls. The entire default Maya interface is created with MEL commands [Gould 2003]. This gives users the ability to customize and extend Maya in almost any way. Scripting with MEL can be done with a stand-alone script written in a text editor and loaded into Maya; it can also be done interactively in Maya's script editor (Figure 15).

MEL is also responsible for editing the scene data. Scripts can control modeling, rigging, animation, effects, lighting, and rendering. Most operations in Maya can be performed with a MEL command [Gould 2005]. Data structures, control structures, and all the standard features of a scripting language are available in MEL. This allows users to create control loops that perform an operation over and over, for instance.



```
polyCube;
// Result: pCube1 polyCube1 //
```

Fig. 15: Autodesk Maya 8.5 with embedded MEL script editor

CHAPTER III

METHODOLOGY AND IMPLEMENTATION

My approach to implementing a scripting system for TopMod involved embedding a Python interpreter into TopMod, reorganizing the core operations into a library, creating Python bindings to the core library, reorganizing the auxiliary operations into a library, and creating Python bindings to those operations.

Embedding a Script Editor

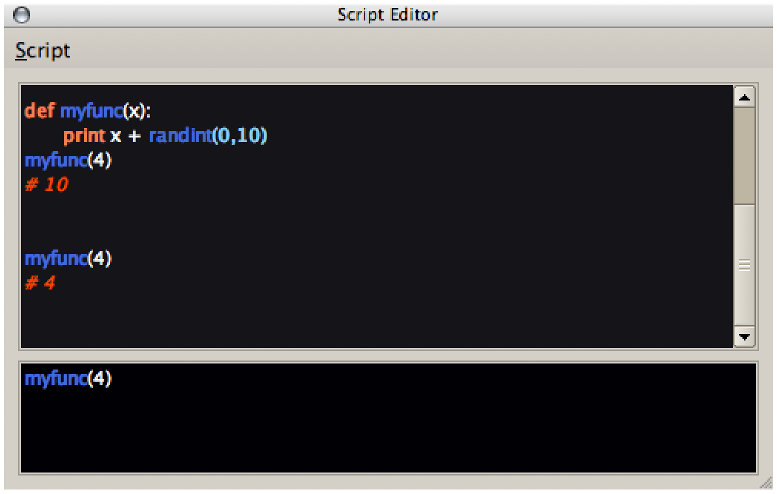
It is often useful for a script to be written in an external text editor and saved to disk. However, not having a script editor embedded in the software poses a big limitation on the user's experience. Users generally utilize both the scripting and graphical interfaces in conjunction. They need to jump back and forth often. They need to quickly see the results of the commands given. Script development usually takes some adjustments and error-correcting to achieve the desired results.

Thus, the primary step in developing a scripting system for TopMod was to create an embedded interface for working with and executing script commands. The goal was to develop an easily accessible interface that provides the user with an efficient means of performing operations.

Maya's script editor served as the basis for the design of TopMod's script editor.

The script editor has two text boxes: one for command input and one for command output (Figure 16). Obviously, input is a requirement for the script editor. The user will type the desired Python commands into the input text box. Some software just have single line input. However, I wanted TopMod to have full scripting capabilities. A user should be able to do all the scripting development within TopMod. This requires a multi-line text editor since most scripts are more than one line of code.

The "Enter" key is a logical choice for submitting scripts to execution. However, multi-line



```
def myfunc(x):
    print x + randint(0,10)
myfunc(4)
# 10


myfunc(4)
# 4
```

```
myfunc(4)
```

Fig. 16: TopMod's embedded Python script editor

input requires use of the `Enter` key. Therefore, I have chosen `Ctrl+Enter` (`⌘+Enter` on Macintosh) to execute scripts in TopMod. After the script is executed, the input text box is cleared and the command is "echoed" (printed) in the output text box. Having the output text box allows the user to keep track of past script commands.

Python's C API allows for a few different modes of interpreting. Some allow for simple 1 line command executions, others are designed to interpret single files. However, for this system it is important for the scripting environment to maintain knowledge of all the variable and function definitions throughout the TopMod session. A mode was chosen that allows this.

In this mode, if a command returns a value and that value is not assigned to a variable, then the result is printed to standard out. This means that the resulting text will show up in the UNIX/DOS terminal (only if TopMod was executed from the terminal, otherwise the output is lost). It is much more practical to see this result in the script editor's window. Most of the time the user is not executing TopMod from the terminal.

In finding a solution, I discovered that Python has the ability to redirect standard out to any class with a `write()` method. TopMod's script editor uses this to print results in the output window. I created a Python class with the sole purpose of capturing text sent to standard out. After capturing the output it is sent to the script editor. It is displayed in the output window just after the echoed input.

To visually separate the results from the command, they are commented out with a '#' symbol (the standard Python comment character). This is another idea borrowed from Maya's script editor. Users often want to copy large blocks of text from the output window and paste that code as input to be executed. Results are not meant to be executed and will cause syntax errors if they are executed. Commenting out the results will cause the interpreter to ignore them.

The visual separation is improved by adding syntax highlighting to the editor. Powerful text editors have syntax highlighters to increase the readability of the code. Keywords, comments, text strings are represented by different colors which stand out clearly against the background. This is something that Maya's script editor lacks.

The most important part of having a built-in script editor in TopMod, is receiving the graphical output from the script interactively. To achieve this the interpreter must to operate on the same set of data as the graphical user interface. When a command is executed via the script editor, the viewport updates these changes.

Python provides an excellent solution for this problem. Generally Python extension modules provide functions to be used in a Python enviroment. In this case, the module needs to provide functions to TopMod's C++ code. The objective of these functions is to tell the Python environment that it is running inside TopMod, and to pass the mesh data between the two environments.

However, these functions should not be accessible by Python's code. Python provides a mechanism to pass data from one extension module to another extension module. The mechanism is known as a C Object. Since Python extension modules are written in C, C Objects can pass data to any C code even though this is not their primary objective.

I have used C Objects to pass two different items between TopMod and the scripting enviroment.

The first is a variable that tells the scripting environment whether or not it is running within TopMod. The reason is that the behaviour of the scripting environment must be different when in the two different places. For instance, loading and saving mesh data inside TopMod should open the file browser and operate the same as if the graphical buttons were being used. In addition, some of the Python functions in the DLFL module behave a little differently in stand-alone mode. I will mention these later.

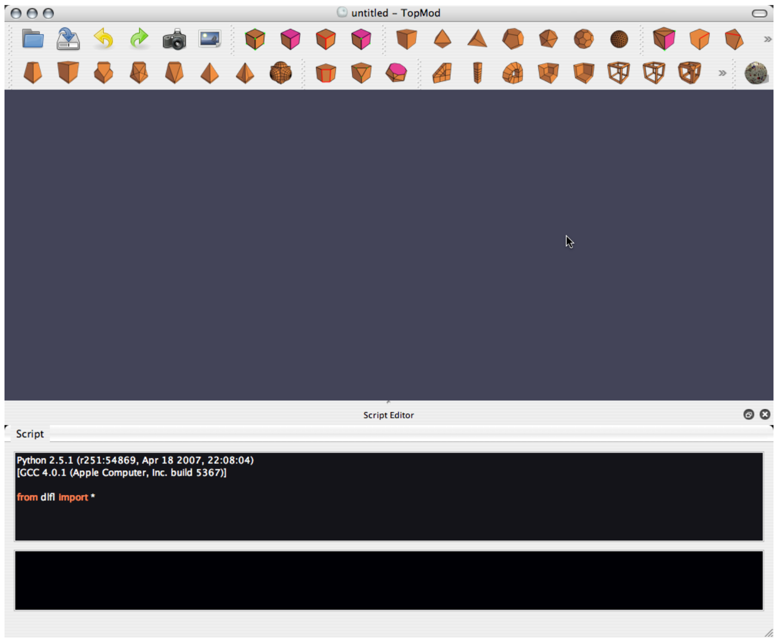
The second C Object accepts a pointer to the mesh data loaded in TopMod. Passing the pointer to the scripting environment guarantees that both systems are operating on the same data. When a command is executed in the script editor the data is changed and a signal is sent to refresh the viewport. Thus, the mesh will have changed just as if the operation has been performed via the graphical user interface.

It is also desirable for the script editor to print commands executed from the graphical user interface. This is known as “command echoing.” Users often want to construct a script, but need to know the equivalent Python command to a standard graphical user interface operation. This is especially important to new TopMod users and new TopMod scriptors. Thus, a signal system was constructed to link the graphical user interface commands with the script editor.

When an operation is executed via the graphical user interface, a signal must be sent to the script editor. The script editor must then recognize the operation. Then, the options for the operation from the graphical user interface are parsed and translated into the string. This string is printed in the output text box just as if it had been executed from the script editor. Each button in the TopMod interface has an action associated with it. Each action grabs all the necessary information to execute the command. I have extended each action to construct it’s equivalent Python command.

In addition to being useful for command echoing, command construction could be used for executing Python straight from the buttons. I have demonstrated this by making one of the operations do just that. The “Insert Edge” operation calls it’s action and constructs the Python command and then executes that command. This is closer to how Maya works with MEL. However, this may slow down more complex operations, so I have only used it here for demonstration purposes.

As I have stated, the script editor must be easily accessible. The user will often go back and forth. This action must be achieved quickly and seamlessly. While a menu button is provided, this is not as rapid as a hot-key. TopMod has hot-keys for many of its features. By pressing a simple key combination (Shift-Ctrl-E or Shift-⌘-E), the script editor will appear and disappear. These hot-keys (like all others in TopMod) can be customized by the user. If the user forgets, then he/she can use the menu button (which has the hot-key combination written next to it as a reminder).



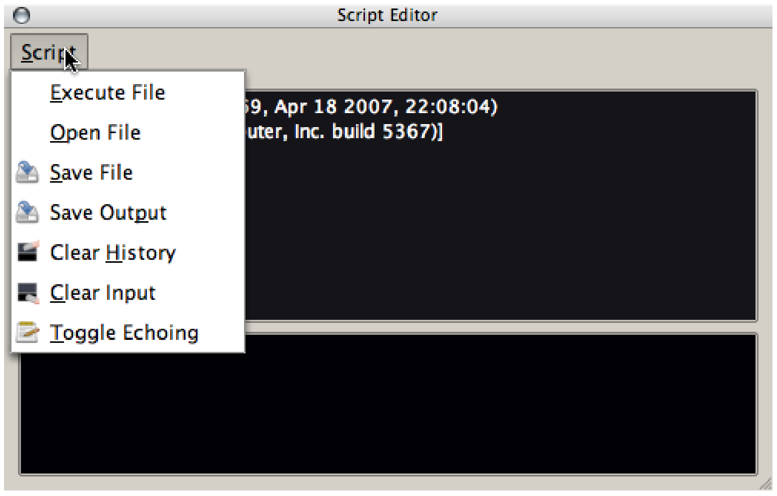
```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]

from dlfl import *
```

Fig. 17: TopMod's embedded Python script editor docked in the main window

Also contributing to the accessibility of the script editor is that it pops up in a separate window on the user's desktop. It can easily be moved around so that it does not occlude other parts of the application the user is working with. Other times, the user doesn't want the separate window. The script editor can be easily "docked" by dragging it over the main window (Figure 17).

Most all interactive scripting environments keep track of the commands executed. This is known as the command history. Scripting shells like bash, csh, zsh all store history. Python's command-line utility also uses this. A user can cycle through past commands usually with the up and down arrow keys. Maya's script editor does not have this currently. However, it is such a useful feature that I have included it in TopMod. Every execution the input text is pushed on to the history stack. Users cycle using the arrow keys as I have described. Cycling through the history places the commands back into the input text box.

A useful feature of Maya's script editor that I have incorporated into TopMod allows users to execute a portion of input text. If a user has written a sizeable script, but wants to execute only a portion of that script, he/she would just highlight that portion with the mouse and execute as normal. Only that portion is added to the history and is printed in the output text box. Additionally, the input text box does not clear. This is useful for debugging as well as designing scripts. The user can also use this feature just to prevent text from being cleared from the input text box. The user can highlight all the text and it will never disappear.



```
59, Apr 18 2007, 22:08:04)
uter, Inc. build 5367)]
```

Fig. 18: The menu for the script editor

TopMod had an undo/redo system built into it. This allows users to revert back to prior states of the model after they have done operations to it. Operations performed via scripting should be no exception. I have arranged for the script editor to signal the undo system when to push the model onto the undo stack. Thus, script operations can be undone and redone just as any other operation. However, this is not available in stand-alone scripting mode as it is not necessary. The script can be edited an re-run as long as the input mesh is not destroyed.

While I have emphasized the usefulness of the embedded script editor, it is also useful (for larger scripting projects) to be able to write scripts externally and bring them into TopMod. A menu (Figure 18) provides options to do this. Firstly, "Execute File" simply executes a Python file on disk. It brings up the file browser. The user selects the desired script. Then the command is run via the Python command `execfile("/path/to/filename")`. After executing, any global variables or functions that the script declared, still exist throughout the TopMod session. An alternative option, "Open File," also allows the user to select a script with the file browser. However, this option loads the contents of the script into the script editor's input text box. The user can than edit the script inside TopMod before execution.

The menu also has the options to "Save" the input text as a script on disk. The output text can be saved as well. Since results are commented, this should result in a valid script as well (assuming there were no errors). Other options exist to clear the history (also clearing the output text box) and clear the input text box. The option to toggle the command echoing on and off is also here. Sometimes it is undesirable to see all the commands echoed since it can clutter up the output window quickly.

Reorganizing the Core Operations

Once the script editor was working and executing Python commands, I could develop the DLFL Python module.

Before creating the Python bindings. I had to analyze the DLFL C++ API. Upon analyzation it became evident that the code would require significant reorganization. The reorganization allowed for TopMod to become modularized. This created a clear C++ API with which Python could be bound.

In addition to being a necessity for this project, the modularization of TopMod is a desirable trait for developers. Developers want to have the flexibility to make use of just the core operations and the DLFL data structure. They can use this in their own applications without requiring a large amount unneeded code.

Modularized software is more flexible for continued development. By organizing modules in a well-designed hierarchy, individual parts can be modified without affecting the others (Figure 19).

The user interface lies on top of the back-end. The advantage of this is that many user interface libraries exist and can be exchanged. Users might want to have multiple user interfaces as well (e.g., one graphical and one command-line based). The old version of TopMod used the FLTK library and the new version uses Qt. This required a major overhaul since TopMod was not modularized at the time.

I have organized the back-end into two modules. These modules are layered one on top of the other. The lowest layer is the DLFL Core. Above the Core is the DLFL Auxiliary.

TopMod's code was highly coupled. Code that belongs in the high-level module was interwoven with the Core. This needed to be addressed in order to modularize the code.

All of the core operations were embedded in the data structure representing the mesh. This means that each instance of a mesh had it's own method to insert an edge, rather than one method that performs the operation for any input mesh.

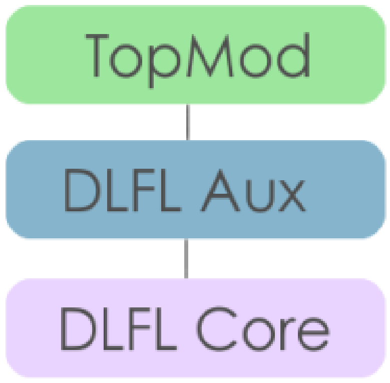


Fig. 19: TopMod's modules in hierarchical order (highest to lowest)

The reasoning behind the previous design is because TopMod only has support for working on one mesh at a time. However, in the future this will change to allow multiple meshes in one scene. Additionally, for other software to use the DLFL Core library, handling multiple meshes may be required by that software.

The scripting system I have designed does have support for multiple meshes. Thus, before I built the scripting system, the Core operations were extracted from the mesh data structure.

Performing this task of operation extraction meant redefining the functions to accept a pointer to the mesh as input. The newly extracted functions then had the problem of no longer having access to private member variables and helper functions of the mesh's data structure. Many of these functions were utilizing this private data. Some of this data did have existing public accessors. Public accessors provide a means for external functions to have access to the private data of a structure. I edited the code replacing private data calls with public accessors. For data needed by the Core operations that did not have public accessors, I created them.

Another issue with the previous code structure was that all of the rendering code is embedded into the mesh data as well. To clarify, the code to render the mesh is part of the mesh. This may seem like an adequate solution, but it is much better for the data storage and data visualization to be independant.

One reason this is desirable is that some uses of the DLFL back-end may have a completely different visualization method, or no visualization at all. This is the case with the scripting system. The system itself requires no use of graphics. When it is used inside TopMod, it is TopMod that is performing the visualization. It is wasteful to pack all the rendering code in with the scripting system when it is never used.

In addition, it is common for other rendering libraries to be used with software. TopMod uses the OpenGL library to render the mesh data. But many Windows applications use DirectX, and other software has it's own custom rendering system.

For these reasons, I have extracted the rendering code from the Core. I moved this code into a new rendering system inside TopMod. The rendering code organization was complicated with several layers of function calls. I simplified the rendering process by organizing the methods and rendering options into one place. One renderer runs per session and takes as input the mesh to render. Based on options set by the user, the renderer determines how the faces, vertices, and edges appear.

Another rendering issue was TopMod's patch rendering feature. This feature allows the user to see a smoothed version of the mesh. The mesh is rendered as smooth patches created by a subdividing the mesh.

Previously, the mesh data structure stored the information to create the patches. Each corner of the mesh had a pointer to its equivalent patch. However, the patches are purely for visualization purposes. Just like the other rendering code, the patch rendering code does not belong in the back-end. I removed them from the Core.

A "patch object" was designed to store the information to create the patches from a DLFL mesh. The patch object has the same numeric identifier as the mesh. This way it is easy to find one, given the other. The patch object now stores all of the patches in a list. The algorithm that creates the patches, finds the patches given their associated corner. However, since the patches are no longer members of corner, they cannot be found this way. To remedy this, I have created a map (using C++ Standard Template Library). This map uses the DLFL Corner pointer as the key and the patches pointer as the value. Patches can thus be easily found.

After extracting these different parts from the mesh data structure, what remains are the classes describing the mesh's faces, edges, vertices, and corners. These classes contain the necessary information to read, construct, store, and write complete 3D meshes.

In addition to the meshes data structure, the Core provides the operations (Insert-Edge, Delete-Edge, Create-Vertex and Remove-Vertex). These operations form the minimal set required to construct 3D manifold meshes. The operations Subdivide-Edge and Collapse-Edge are also be provided. Although these operations can be performed with the previous four, they are provided with more efficient implementations, since they are very common operations.

All of this now forms the self-contained library known as the DLFL Core.

Creating the Core Scripting Bindings

Once I had the Core designed in C++, I had to provide a means for users to access the Core library in Python. I developed a Python extension module. Python extension modules are built in C (or C++, as in this case). The modules provide access to the library with function bindings. Bindings interpret the Python code input by the user into C code. The code is then evaluated in C and the results are converted back into Python.

As with the creation TopMod's Python script editor, I used Python's C API to implement the Python module. The script editor makes use of the API's ability to embed Python in C. However, the module makes use of the API's ability to extend Python.

Binding a function with the Python C API works as follows: create a C function which accepts a Python object as input and returns a Python object as output. In Python, everything that stores data is an object [Chun 2000]. This includes general objects such as numbers, strings, lists, files, and classes.

The input Python object represents a tuple containing all of the arguments to the Python function. This tuple must be parsed to convert each of the arguments into a C data type. Python's C API provides a simple function to do this. The programmer determines the format of the input arguments and constructs a string representing the types of the arguments. For example, the character 'i' represents an integer, and parentheses designate a tuple. The format also allows you to specify optional arguments with a '—' character

Once the data types are converted, the C/C++ function evaluates the data and constructs the result. The resulting C data types must then be converted back into Python data types to be returned. Again, the API provides a simple function to construct a resulting value. Just as before the return format is determined by a string.

Once all of the functions are created, the module is initialized. This provides the proper function definitions to Python. This is passed with a large array with each index containing a string representing the name of the function in Python, and the name of the function in C.

Users first need the ability to create and design meshes with the scripting system. Bindings to the minimal operators provide this ability. However, the operators by themselves are not very useful. Users need to direct the creation and deletion of specific mesh elements. For example, the user must identify which edge to delete. This is not something a user can do without helper functions to identify and track mesh elements. Thus helper functions are provided for this purpose.

In order for a user to choose a mesh element to manipulate, he or she needs to identify it. I have provided access to mesh data (vertices, edges, faces, corners) via identification numbers (represented by integers). The DLFL C++ code was already storing these IDs for the mesh data. However, the Core operations recognized mesh data by pointers to an address in memory. Thus, a conversion scheme between numeric identifiers and memory addresses is neccessary.

Given an address, finding the ID is trivial, since the data structure stores is. However, it is also necessary to find the address given the ID. The preliminary approach was to loop through all elements until it is found. When the mesh only has a few elements, finding these IDs is not too costly, however as the mesh's detail increases, the search becomes slower.

Most of the time, TopMod users develop meshes of high density data (meaning lots of faces, vertices, etc.). Therefore, I wanted to provide a more optimal means of element identification. With the linked lists being used, the address lookup is an O(n) operation. Alternatively, storing vertices in a hash map allows pointers to be found in O(1) time in most cases. To implement this I used the hash map provided by C++ Standard Template Library. This allowed me to use the ID as the key for each entry and the pointers address as the value. When the element is created it is given and ID and an address. The address is a number allowing it to be typecast into an integer. Both of these are then placed into the map. Then, at any future point a value (the element address) can be looked up by its key (the numeric ID).

Many of the operations take corners as input. For example, the `insertEdge` operation takes two corners and inserts an edge between them. Currently, I have implemented corner IDs as a 2-tuple: `(faceid,vertexid)`. This representation is easy for a user to understand and work quickly. The idea of a corner is a bit more abstract. Additionally, there exist many more corners than faces or vertices. Displaying all the corners on the screen can clutter it even more. Despite the benefits of representing corners this way, there is a possible issue over this which has caused some controversy. I will address this in the future work.

In order for the Python module to be of use in a stand-alone environment, users need to be able to move data to and from disk. The DLFL Core module provides operations for opening and saving DLFL mesh data. I provided Python bindings to these functions. The functions also determine the file format (Wavefront OBJ or DLFL) by the filename's extension (respectively, .obj or .dlfl).

When a mesh is loaded it is given a numeric identifier. This is necessary for the multiple mesh support that I have provided in the scripting system. The ID can be used to switch between meshes. The current mesh is the one that is being operated on. The mesh ID can also be used to "kill" the mesh (unloading it from memory).

To demonstrate this, I have re-written the Doo-Sabin subdivision scheme in Python using only those functions provided by the DLFL Core. The Doo-Sabin subdivision scheme was already written in C++ and is provided in the DLFL Auxiliary Python Library. However, I additionally wrote this version in Python to prove it could be done, and also to find out if any additional helper functions were necessary.

The result of the Python version of Doo-Sabin is identical to the C++ version. The method works by looping through the existing geometry and computing the new vertex coordinates. Once the new vertex coordinates are created the old geometry is deleted and new geometry is created from the calculated vertices. This subdivision scheme was tested on a variety of primitives. To ensure its correctness, a cube with an overlapping edge was tested. In TopMod, an edge can be inserted on top of an existing edge. This produces a hole in the mesh when it is subdivided. A comparison of the Python and C++ versions can be seen in the results section.

Scripting with primitive data accessed via IDs creates a problem for the scripting interface: How can one choose the appropriate ID to manipulate the desired data? When using the graphical user interface, users have the option to toggle on/off the display of IDs tagging faces, edges, and vertices. This places a rectangular box next to each element with its ID integer value. Each type of element is represented with a different color so they can be quickly visually identified by the user. Each tag is also given a opacity value based on its distance from the viewport's camera. The distances of all the displayed tags are compared to find the minimum and maximum distance. Then the distances are normalized to fit in the range of valid opacity values (0 to 1). The further the distance the more transparent the tag.

Drawing all of the tags can cause the viewport to become cluttered for complex meshes. With a high density mesh, the tags can sometimes cover the entire object. Thus, the option to draw labels for only those items selected by the user is also available. The user can toggle this option on and off. When the option is on for faces and no faces are selected, no face tags are drawn. When the user selects a face, that face tag is displayed.

Users might also select a set of data which would be placed into an array. MEL accomplishes this with `ls -selection`. The `ls` stands for list, and the `-selection` flag says to list only those items selected. The results of this command can be stored as an array of strings.

A similar method is used in this system. Functions exist to return the faces, edges, vertices, and corners as a list. A boolean flag tells the function whether to return all of the items or only those selected by the user in the viewport. When scripting stand-alone, the user cannot select items. Therefore, it always returns all of the items. Using this command creates a very simple way to loop through the faces, edges, vertices, or corners (Figure 20).

```
for f in faces():
    print f
```

Fig. 20: Sample Python code to loop through faces and print their IDs

Another procedure to access elements by ID is `Walk`. The procedure takes a face ID as input and returns a list of vertices and a list of edges that are visited by "walking" the faces boundary. The procedure is derived from a previously existing procedure inside the DLFL Core, called Boundary-Walk. This prints a list of vertex IDs given a face ID [Akleman and Chen 2000]. An additional version of `Walk`, (`cornerWalk` is also provided for convenience. Often time the results of the walk are to be used in with operations taking corners as input. Rather than the user prepare the face, vertex tuples, the walk function prepares them.

Another useful feature of the scripting interface is the set of element information functions. Each of the classes representing vertices, edges, faces, and corners contain several attributes which are useful for scripting. None of these attributes requires any intense calculation. Rather than provide separate functions for each attribute, one function will return all the information for each element type. `vertexInfo`, `edgeInfo`, `faceInfo`, and `cornerInfo` provide this useful data for their respective element type. The data is returned as a Python dictionary.

A Python dictionary has a key and a value. Values can be looked up given a key. For example, `vertexInfo(vertexid)['valence']` outputs the valence of the vertice passed as input.

- `vertexInfo` returns: `'id','type', 'coords', 'valence'`
- `edgeInfo` returns `'id', 'type', 'midpoint', 'normal', 'cornerA', 'cornerB', 'cornerC', 'cornerD','length'`
- `faceInfo` returns `'id', 'type', 'centroid', 'normal', 'size'`
- `cornerInfo` returns `'faceid', 'vertexid', 'edgeid, 'type'`

One issue which I came across in development is the effect of `insertEdge` on the face IDs of the object. The `insertEdge` operation takes as input two corners – each corner is represented by a `(faceid,vertexid)` tuple. Often the face ID for at least one of the input corners changes. However, since the Python does not allow the values of its inputs to be changed by the function, this causes problems for future use of the variables. To address this issue, I have constructed the `insertEdge` operation to return the new edge ID and all four corners associated by the new edge. These four corners will always be returned in the same order. These corners can also be found at any later time using `edgeInfo`.

To test this new scripting system, I began by executing each operation individually. I validated it by comparing its visual result to that created via TopMod's graphical user interface. Any operations that weren't satisfactory were debugged and corrected.

After testing simple one line scripts, I began experimenting with more complex scripts. I incorporated loops and conditionals and combined operations. I will discuss these in the results.

The Python module is provided as `dlfl` and can be imported like any other Python module via either of these two lines:

```
import dlfl
```

```
from dlfl import *
```

The difference in these two methods of importing is in the way module methods are called. In the prior method, a `dlfl.` must be prepended to each call (e.g., `dlfl.load("cube.obj")`). The latter method does not require the module name prefix.

The script editor is setup to automatically import the module and will print "`from dlfl import *`" in the script editor to let the user know the module loaded correctly. Thus, in TopMod's script editor the `dlfl.` is not needed. It the module is not loaded because it is missing or not in the right place then an error is printed to notify the user. The script editor is still useable, but it will not interact with TopMod.

Reorganizing the Auxiliary Operations

Above the DLFL Core is the DLFL Auxiliary (Aux). These are higher-level operations which provide TopMod with a strong set of useful operations. These are the operations most commonly performed by users.

The Aux library contains many useful functions that operate using the Core operations and DLFL data structure. These operations include many different face extrusion methods, over two dozen different subdivision schemes, and many high-genus operations.

TopMod would not be as useful a program if the user could only work with the minimal set of operations provided by the Core. This would mean a lot of work just to perform seemingly simple operations on the mesh. Just to perform a simple extrusion would require several vertices to be created and several edges to be inserted connecting the new vertices. This is why a set of Auxiliary operations is provided as a library to be used by TopMod, the scripting system, and any other application that desires.

As with the Core operations, all of the Auxiliary operations were members of the mesh's data structure. In this case it is even more important to extract the operations from the data structure.

The structure is provided with the Core library. Since the Auxiliary operations do not belong in the Core, it was adement that I extract them.

Each operations was extracted from the mesh's data structure and placed into an appropriate source file. Each file represents a different type of Auxiliary operation. All of the extrusion operations and helper functions were grouped together in a file. The subdivision operations were grouped together in a file.

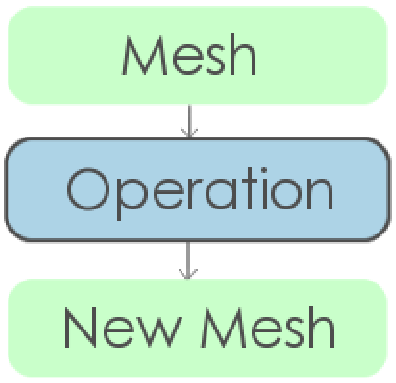


Fig. 21: This illustrates the design theory that a mesh is input to an operation which results in a new modified mesh

The newly extracted function definitions for the operations were rearranged to take a mesh as input to operate on. Again, this allows support for multiple meshes in a scene. It also follows with the design theory that the mesh is input to an operation and is output as a modified mesh (Figure 21). This is a more logical approach than having an operation such as EXTRUDE-FACE belong to a mesh.

Many of Auxiliary operations were accessing private member data just as the Core operations had been. I solved this in a similar manner to the earlier solution by creating the necessary public accessors, and changing the code in the Auxiliary operations to use them.

All of this forms the DLFL Auxiliary. This library requires the presence of the DLFL Core library. For TopMod, I have left the linking stage up to TopMod. This means changes to the Core library does not require the Auxiliary library to be recompiled as long as the Core API does not change.

Creating the Auxiliary Scripting Bindings

After setting up the Auxiliary library in C++, I added the bindings to the Python extension module that I created for the Core. Since these operations are such a pertinent part of TopMod, it follows that they are important to a scripting system used with TopMod.

As I mentioned, the Auxiliary functions could all potentially be rewritten in Python. However, because they already existed in C++, I thought it better for them to remain as they were. There is a lot of complex code to perform these operations. Translating the code to Python would not be productive, since they are perfectly capable of being used in Python as bindings.

The process of creating the auxiliary scripting bindings was the same as the core scripting bindings. The Python input arguments are read in and parsed into C++ data types. Then, the results of the C++ code are used to build any Python return value required.

There are several different extrusion types and two dozen different subdivision schemes. In the C++ source code, each of these is its own function. I wanted a much cleaner interface for the user. I found a solution to reduce the number of functions. I chose to consolidate these different types into 1 extrusion function and 1 subdivision function. I borrowed a method used in the RenderMan Shading Language (Figure 22). This method uses string identifiers to represent options for a function. A string is a better representation of these options than a numeric identifier. The string describes the choice. Users can remember names, but would have trouble remembering which number corresponds to which option.

In this system, both of these functions (`extrude` and `subdivide`) accepts a string to designate the type (Figure 23). Each extrusion algorithm has different arguments and each subdivision

```
    filterstep(x, 0, 1, "filter", "catmull-rom");
```

Fig. 22: Example of string identifiers used in RenderMan Shading Language

algorithm has different arguments. However, the number of arguments and type of arguments across the different types is mostly the same. Thus, the use of arguments can be determined based on the string identifier passed as the first argument. This is a little tricky in Python since all of the arguments must be parsed in one statement.

```
    extrude("doo-sabin",faceID)
    subdivide("simplest")
```

Fig. 23: Examples for using strings identifiers for extrusion type and subdivision scheme

The solution was to make certain arguments optional. Thus, if an option is not necessary for a certain type of extrusion it is not used (even if the user provides it). The same goes for subdivision schemes. This was much trickier since there are a lot more subdivision schemes then there are extrusion types, and the argument types are more variant. Still, I was able to work out a solution with optional arguments. I also used to my advantage, the fact that a boolean is really just an integer. For those schemes that required booleans, I just typecast the argument to a boolean.

Again, I tested this new scripting system. This time testing the new operations that are part of the DLFL Aux. I began by executing each operation individually. I validated it by comparing its visual result to that created via TopMod's graphical user interface. Any operations that weren't satisfactory were debugged and corrected.

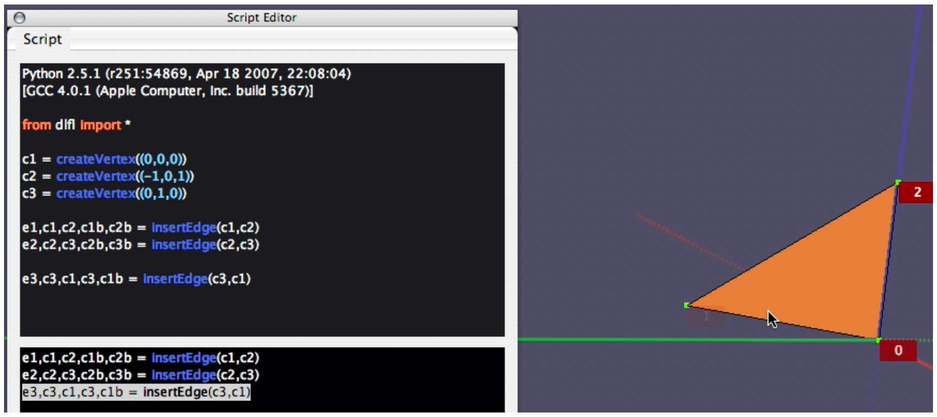
CHAPTER IV

RESULTS

Scripting with the Core Bindings

I first began by testing only the DLFL Core. At this point, I hadn't written the DLFL Aux. I wanted to test the Core before moving on so I could determine any potential issues that might carry over, and resolve those issues.

An initial test I did was to build a triangle from scratch with scripting. This makes use of two operations: `createVertex` and `insertEdge`. In TopMod's graphical user interface, `createVertex` is not available. It is a low level operation, that is only provided through the scripting interface. The consequence of this is that modeling without using scripting requires the user to begin with some primitive shape. The user cannot create any geometry in empty space at any point during the modeling process without the scripting interface.



```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]

from dlfl import *

c1 = createVertex((0,0,0))
c2 = createVertex((-1,0,1))
c3 = createVertex((0,1,0))

e1,c1,c2,c1b,c2b = insertEdge(c1,c2)
e2,c2,c3,c2b,c3b = insertEdge(c2,c3)

e3,c3,c1,c3,c1b = insertEdge(c3,c1)
```

```
e1,c1,c2,c1b,c2b = insertEdge(c1,c2)
e2,c2,c3,c2b,c3b = insertEdge(c2,c3)
e3,c3,c1,c3,c1b = insertEdge(c3,c1)
```

Fig. 24: Triangle using only createVertex and insertEdge

The script works by first creating three vertices. The placement is up to the user. Now edges must be created between the vertices. The `insertEdge` operator does this. This operation takes two 2-tuples representing the corners to connect. Before inserting one edge between the first two vertices, each vertice has a belongs to a different face. However, after the operation, the two vertices belong to the same face. This is the reason it is crucial to have the resulting corners returned. These new values are needed in order to insert the next edge correctly. The result can be seen in Figure 24.

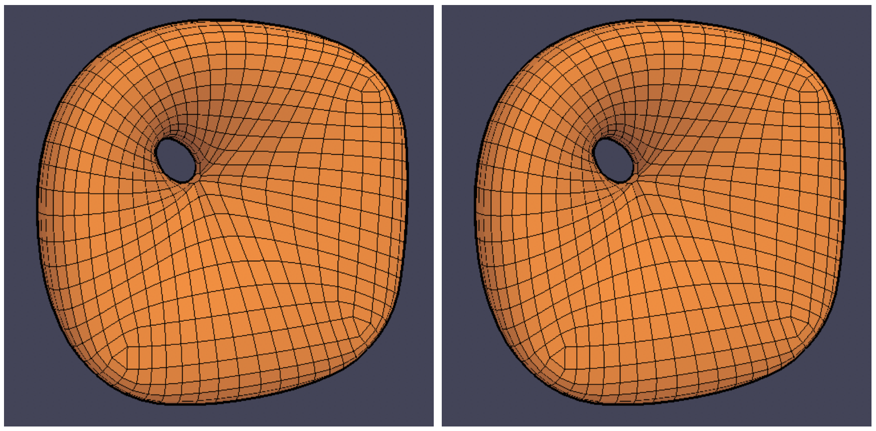


Fig. 25: Doo-Sabin subdivision scheme comparison on a cube (with an extra edge). This version of the subdivision scheme was completely rewritten in Python using only the DLFL Core, but produces identical results to the C++ version in the DLFL Aux

I also did a much more complex test of the scripting system. In theory, all of the Auxiliary operations can be performed using only the Core operations. To test this, I chose an Auxiliary operation to create entirely in Python. I chose the Doo-Sabin subdivision scheme. The operation works by looping through the current faces and its vertices. Based on its vertices' coordinates, it calculates new vertex coordinates. In Python, I store these as a list. Then I use the `createFace` operation with the new vertex coordinates for input. This operation saves some code for scriptors who would normally have to use `createVertex` with `insertEdge` several times. All of the old geometry is deleted. The resulting mesh has the same number of faces before, but now they are all detached from each other. Finally, these faces are connected with an edge connect routine. This routine exists in the DLFL Aux, but was rewritten in Python for this exercise. A comparison of the Doo-Sabin subdivision in Python and C++ can be seen in Figure 25.

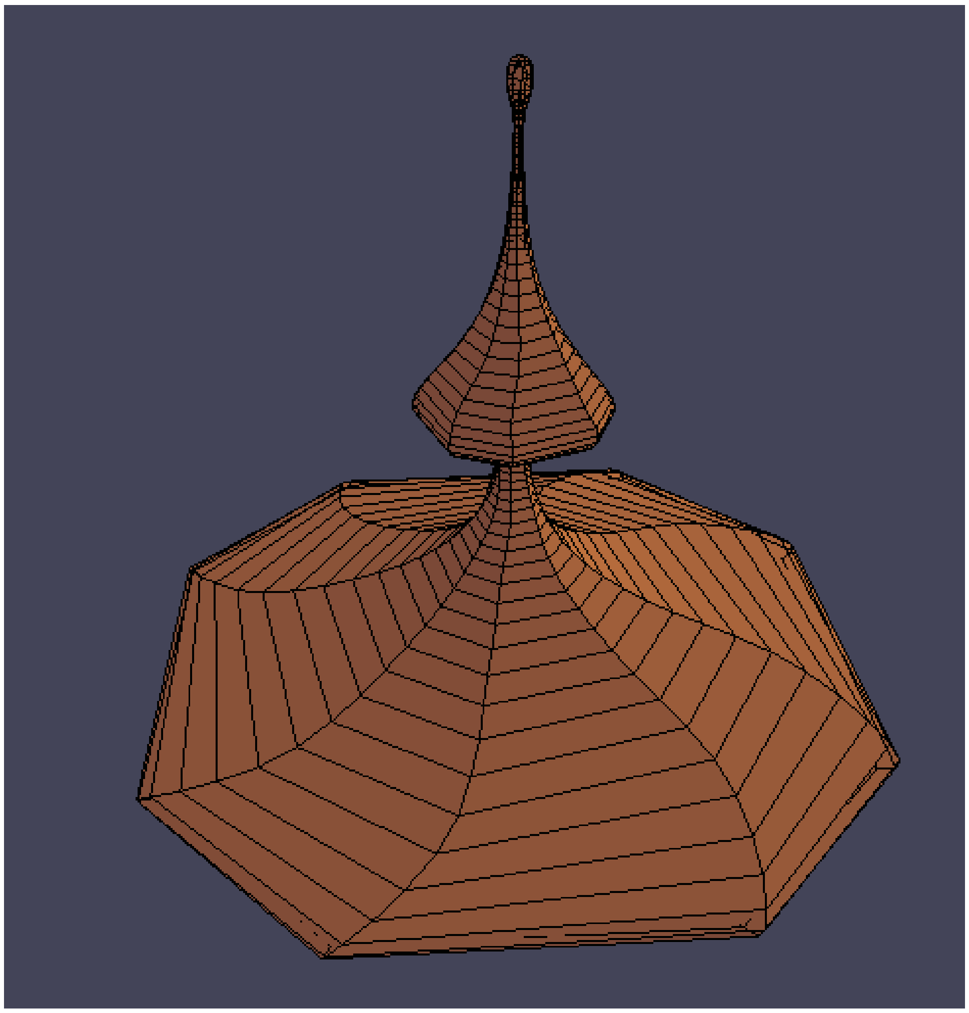


Fig. 26: Example of controlling parameters with a sine function

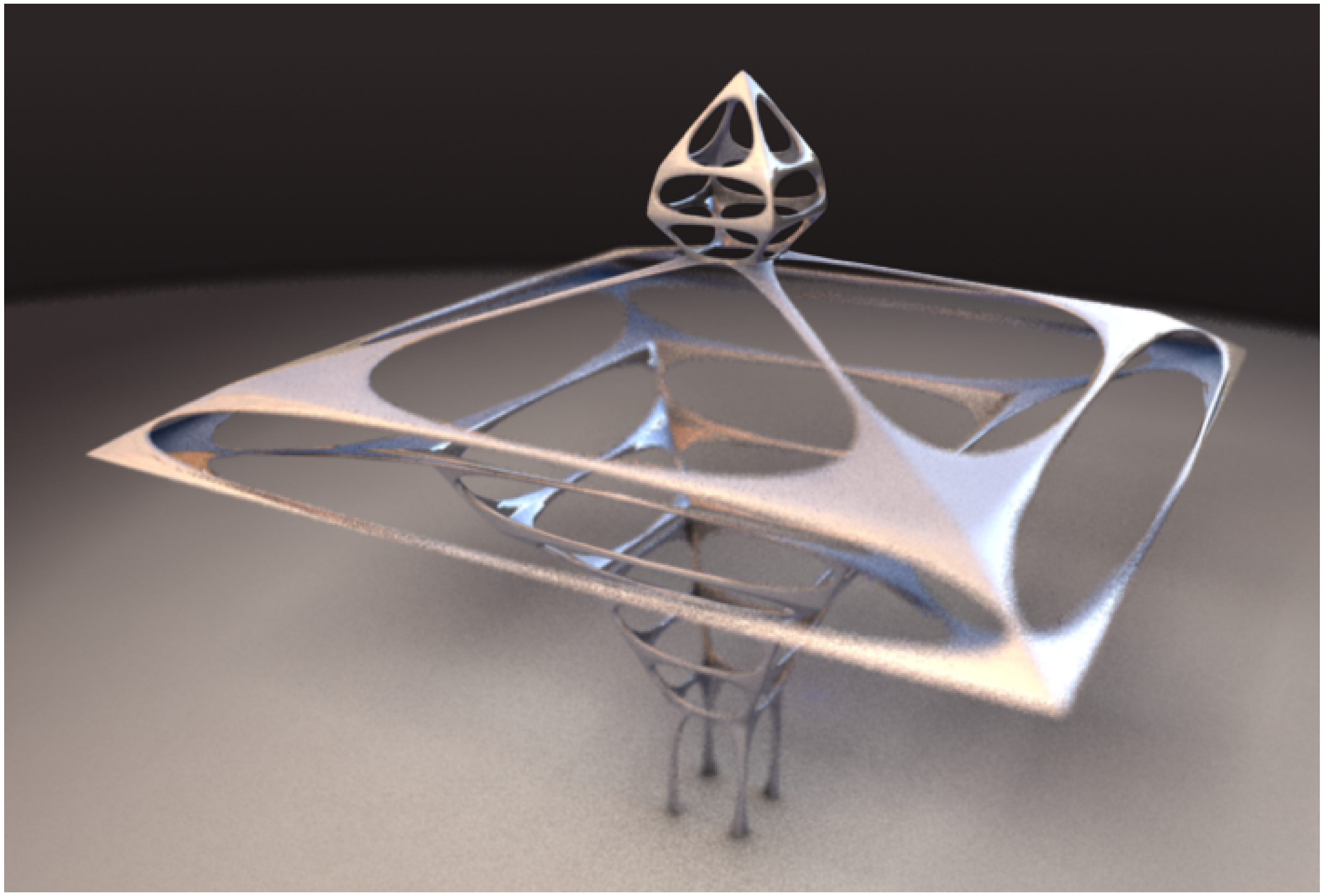


Fig. 27: Tower created by controlling the distance and scale parameters of an extrude with the sine of the current time

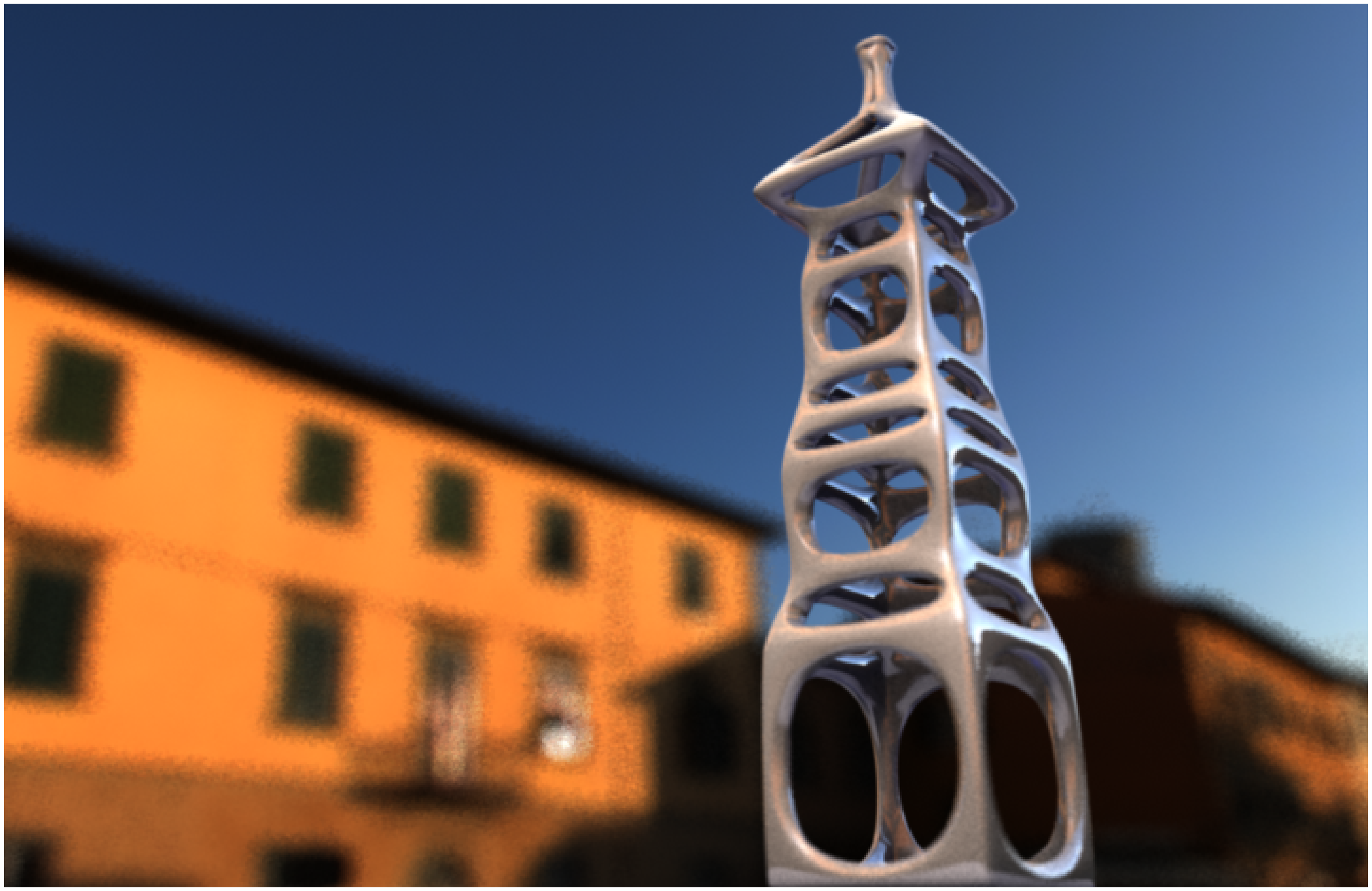


Fig. 28: Another tower created by controlling the distance and scale parameters of an extrude with the sine of the current time. The code for this tower is the same as the previous, but running it at a different time yielded different results

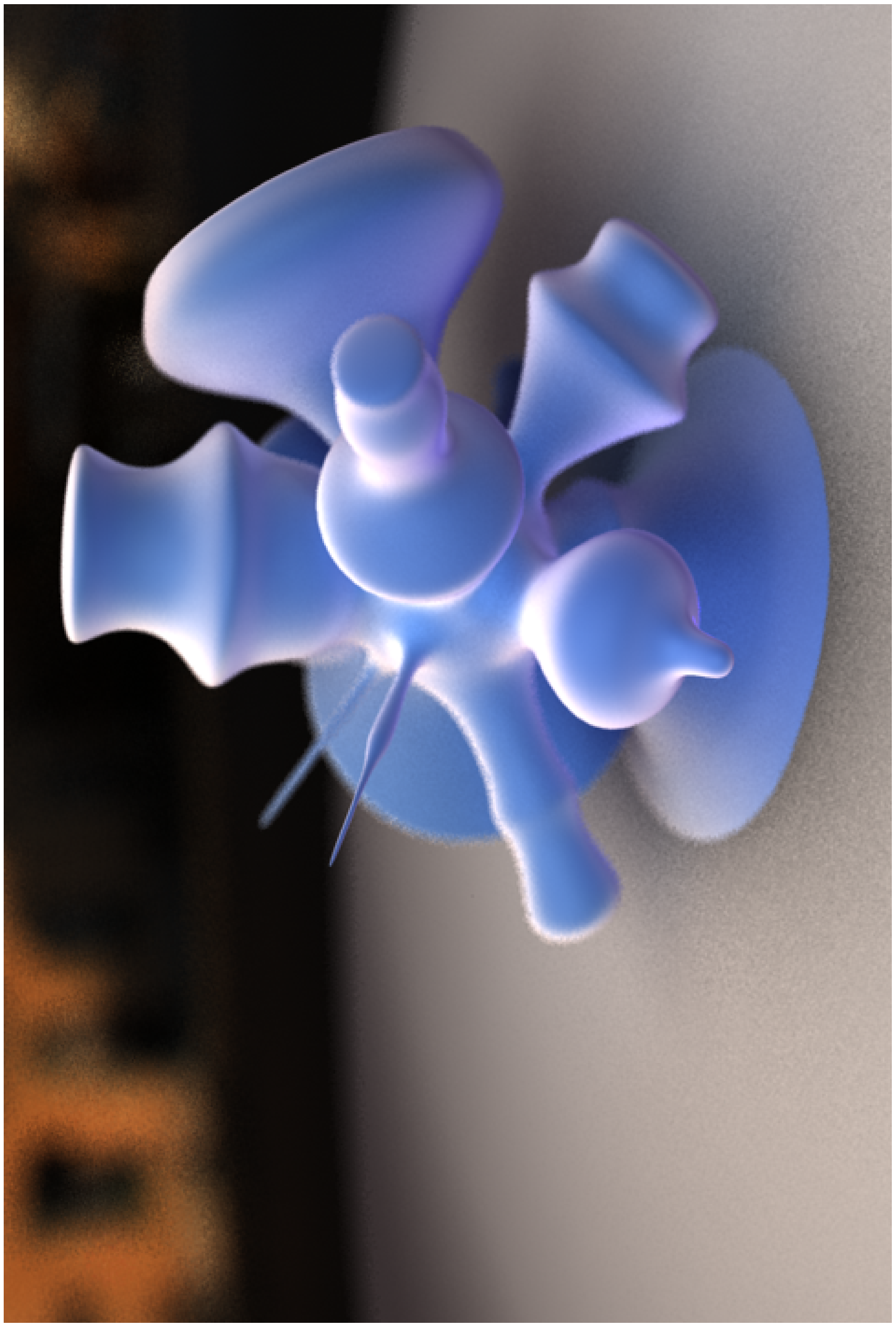
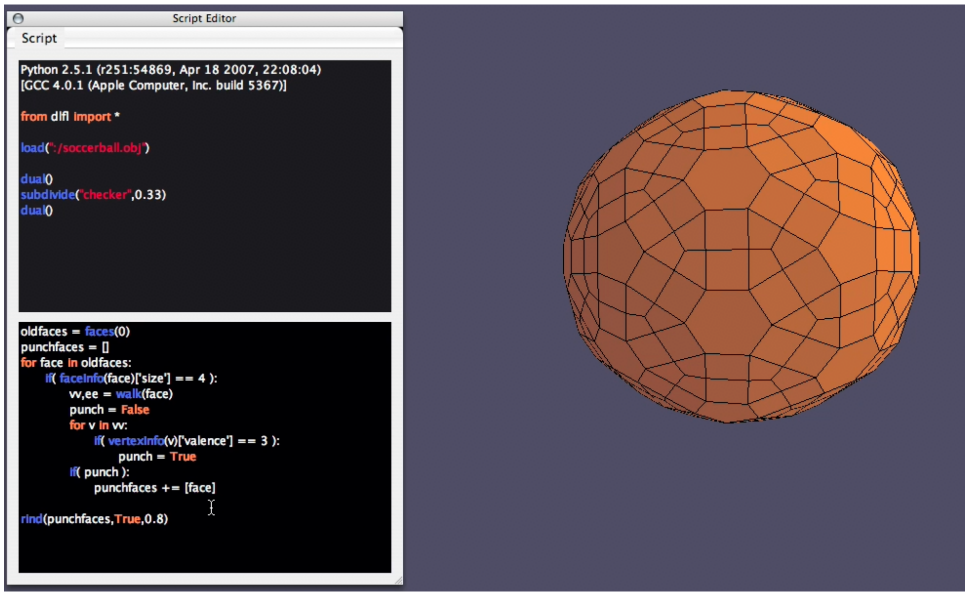
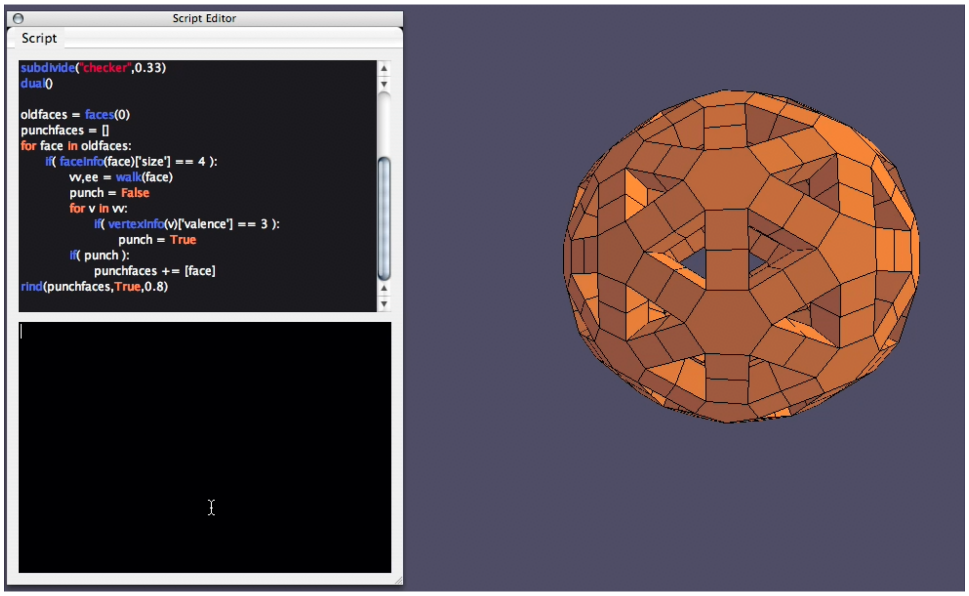


Fig. 29: A dodecahedron which used similar code as in the time towers in Figures 27 and 28



```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]

from dlfl import *

load(":/soccerball.obj")

dual()
subdivide("checker",0.33)
dual()
```

```
oldfaces = faces(0)
punchfaces = []
for face in oldfaces:
    if( faceInfo(face)['size'] == 4 ):
        vv,ee = walk(face)
        punch = False
        for v in vv:
            if( vertexInfo(v)['valence'] == 3 ):
                punch = True
        if( punch ):
            punchfaces += [face]

rind(punchfaces,True,0.8)
```



```
subdivide("checker",0.33)
dual()

oldfaces = faces(0)
punchfaces = []
for face in oldfaces:
    if( faceInfo(face)['size'] == 4 ):
        vv,ee = walk(face)
        punch = False
        for v in vv:
            if( vertexInfo(v)['valence'] == 3 ):
                punch = True
        if( punch ):
            punchfaces += [face]
rind(punchfaces,True,0.8)
```

Fig. 30: An example of rind modeling. I have chosen to punch all the four sided faces where at least 1 vertex is 3 valence

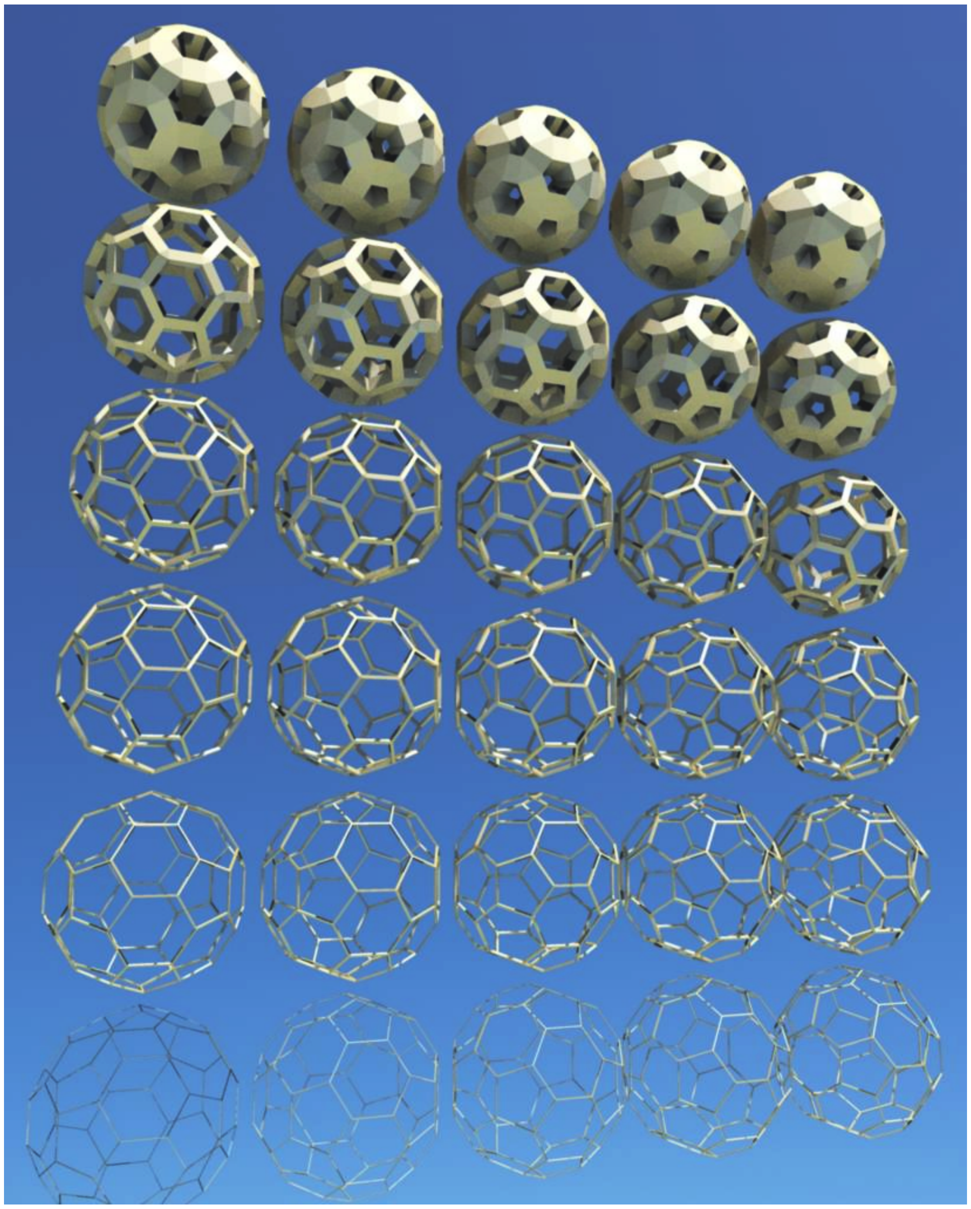


Fig. 31: 30 versions of a wireframe soccerball. Each has a different thickness. Then a script was written to organize them into this grid arrangement

Scripting with the Auxiliary Bindings

Scripting with the Auxiliary operations is very useful. Most of these operations contain several parameters and require several clicks to perform.

One of the most interested operations to script is extrusion. All of the extrusions have parameters that affect the distance, the rotation, and the scale. I played with these by creating a loop to extrude several times. In Figure 26, I began with a cube (seen at the very tip of the mesh) and controlled its distance and scale by the sine value starting at $0°$ to $360°$. To smooth out the model, I took the dual twice and used Doo-Sabin subdivision once.

Similarly, Figures 27 and 28 were based on sine. However, these extrusions were modulated by the sine of the current system time. The current time is returned by Python as a floating-point number in seconds since the Epoch. This was put in a loop and used Python's sleep command to create a significant difference in the time for each extrusion. Again, I used this to control distance and scale. To make the sculpture more interesting, the script also performed the wireframe option on the mesh. The two sculptures are completely different even though they were run with identical code.

Figure 29 also used this concept of taking the sine of the time. This time it started out as a dodecahedron. The script looped through each face and extruded outward several times, turning it into a very organic shape. Each face extruded differently.

Another test was rind modeling. To reiterate, rind modeling is creates a crust from the mesh with a given thickness. Then the user punches faces to create holes as desired. Previously, the user had to hand pick each face to punch. However, with scripting conditions can be used to determine which faces to punch.

In the example that resulted in what is shown in Figure 30, I started with a soccer ball and did dual checkerboard subdivision. The result was the top image in the figure. I decided as a user that I wanted to punch all of the faces that formed the triangle shapes. I noticed a pattern. Each of these faces had four sides. However, there were other faces with four sides that I did not want to punch. I looked for another condition. Of all the faces with four sides, these are the only ones that have a three-valence vertex. I wrote a script to loop through all these faces and put into a list only those that met my conditions. Then the list was fed to the rind command and it worked perfectly.

An interesting example I made in Figure 31, further illustrates the power of the scripting system. I started with a simple soccer ball mesh. I performed the `wireframe` operation on the mesh. This operation is provided by the DLFL Auxiliary library. The operation has a parameter to control the thickness of the solid wireframe it creates. I ran a loop to perform this several times starting very thin and increasing the thickness. The script then moved the soccer ball to a position on a grid and save each soccer ball out. I then loaded the grid into Maya with a MEL script and rendered it.

CHAPTER V

CONCLUSION AND FUTURE WORK

With this new scripting interface, TopMod users can write their own code, perform useful low-level operations, efficiently perform repetitive tasks, and control parameters with functions.

The scripting system provides a module with access to the basic operations to build meshes represented by the DLFL data structure. This includes operations like `createVertex` which are not accessible in TopMod's user interface.

Since the scripting system is designed with Python scripting language, users can utilize loops, conditional statements, and other logic to build procedural meshes.

Python provides users with access to other utilities, such as mathematical functions, that can be used to manipulate operations' parameters.

TopMods source code is now modularized. The lowest level, the DLFL Core is useful on its own. Other software can use it without the rest of the TopMod software. The DLFL Auxiliary provides another layer of very useful operations. It is dependent on the DLFL Core, and together they make up TopMod's back-end. TopMod's front-end now is separable from the back-end. The front-end contains all of the code for visualization of the DLFL meshes.

The future work with this system remains largely in the hands of the users. They can extend the functionality of the system by building on top of each others scripts. The more users share their scripts and collaborate, the more robust the scripting system will become. It is in the users' creativity that the full potential of the scripting system can be realized. Because Python is a widely used language with a plethora of existing features, the types of scripts sthat can be written are endless. Like MEL, Python can create graphical user interfaces. But MEL is limited to creating widgets in just one toolkit. Python has PyQt which binds to Qt. Python also has bindings to Tk, GTK, and others. Users can create custom dialogs with their preferred toolkit.

Python can also be used with a database, like SQL. Databases can be very useful with 3D software. For example, databases are often used to store variations on a 3D mesh. The meshes can be organized by certain features. The features can then be queried for matching results. This could be useful for storing a database of architectural models.

And there are modules to create networking tools. This could allow mesh data to be shared across a network. These leads to the possibility of interactively sharing mesh data.

Future work can also be done in terms of the scripting engine itself. A useful feature would be a “Play through Script” option inside TopMod. This would run through a script pausing between steps defined in the script. The would be a great learning tool for others to see how others created their models.

Since Python has a `sleep` command (which I have verified works in TopMod’s script editor), this could be achieved. However, the scripting system needs to be modified to run as a separate thread for this to work properly. Currently, when the `sleep` command is running, it causes the whole TopMod application to sleep.

One issue with the scripting system that still needs to be resolved is the representation of corners. Currently, the corners are accessed by a 2-tuple of face ID and vertex ID. Developers may want to use the Python scripting system in a way that causes the face,vertex tuple to no longer be unique. For general users, however, it is much easier to access with a face-vertex tuple.

REFERENCES

Akleman, E., and Chen, J. 2000. Guaranteeing 2-manifold property for meshes by using doubly linked face list. *International Journal of Shape Modeling 5*, 2, 149–177.

Akleman, E., and Srinivasan, V. 2002. Honeycomb subdivision. *Proceedings of ISCIS'02 17th International Symposium on Computer and Information Sciences*, 137–141.

Akleman, E., Srinivasan, V., and Chen, J. 2002. Interactive construction of multi-segment curved handles. *Proceedings of Pacific Graphics*, 429–430.

Akleman, E., Chen, J., and Srinivasan, V. 2003. A minimal and complete set of operators for the development of robust manifold mesh modelers. *Graphical Models 65*, 286–304.

Akleman, E., Srinivasan, V., and Chen, J. 2003. Interactive rind modeling. *SMI '03: Proceedings of the Shape Modeling International 2003*, 23.

Akleman, E., Edmundson, P., and Ozener, O. 2004. A vertex truncation subdivision scheme to create intriguing polyhedra. *Bridges: Mathematical Connections in Art, Music, and Science 2004*, 117–124.

Akleman, E., Srinivasan, V., Melek, Z., and Edmundson, P. 2004. Semiregular pentagonal subdivisions. *SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04)*, 110–118.

Chun, W. 2000. *Core Python Programming.* Prentice Hall, Upper Saddle River, NJ.

Fitzmaurice, G. W., and Buxton, B. 1998. Compatibility and interaction style in computer graphics. *Computer Graphics 32*, 4, 64–68.

Gould, D. A. D. 2003. *Complete Maya Programming: An extensive guide to MEL and the C++ API*, vol. 1. Morgan Kaufmann Publisher, San Francisco.

Gould, D. A. D. 2005. *Complete Maya Programming. Volume II: An in-depth guide to 3D fundamentals, geometry, and modeling*, vol. 2. Elsevier/Morgan Kaufmann, Boston.

Landreneau, E., Akleman, E., and Srinivasan, V. 2005. Local mesh operators: Extrusions revisited. *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI' 05)*, 351–356.

Landreneau, E., Akleman, E., and Keyser, J. 2006. Interactive face replacements for modeling detailed shapes. *Proceedings of Geometry, Modeling and Processing*, 602–608.

Lutz, M. 1996. *Programming Python.* O'Reilly & Associates, Inc., Sebastopol, CA.

Mandal, E., Akleman, E., and Srinivasan, V. 2003. Wire modeling. *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, 1.

Srinivasan, V., and Akleman, E. 2004. Connected & manifold sierpinsky polyhedra. *SM '04: Proceedings of the ninth ACM symposium on Solid modeling and applications*, 261–266.

Srinivasan, V., Mandal, E., and Akleman, E. 2005. Solidifying frames. *Bridges: Mathematical Connections in Art, Music, and Science*, 203–210.

APPENDIX

Python Core Commands

Core Functions

createVertex((x,y,z)). *Creates an isolated vertex with it's point-sphere at a position specified by a 3-tuple. No edge is created. An object can be created this way. This is one of the minimal operators.* ***Result:*** *(faceid,vertexid),objectid*

removeVertex(vertexid). *Removes/Deletes an isolated vertex and it's point-sphere. This is one of the minimal operators.* ***Result:*** *None*

insertEdge((faceid,vertexid),(faceid,vertexid)). *Inserts an edge connecting two corners. Each corner is represented by a tuple: (faceid,vertexid). When the new edge is created the input vertices become members of the same face. Since integers are immutable in Python, we must return these also.* ***Result:*** *edgeid,(faceid,vertexid),(faceid,vertexid),(faceid,vertexid),(faceid,vertexid)*

deleteEdge(edgeid). *Deletes the edge represented by edgeid. When the edge is deleted then one face remains. If the deleted edge's sides belonged to two separate faces then the second one is deleted. A list (of size 1) with remaining face's ID is returned. Otherwise, if the deleted edge's sides belonged to the same face, then one new face is created. A list (of size 2) with the 2 face IDs is returned.* ***Result:****[faceid,...]*

subdivideEdge([divisions,]edgeid). *Subdivides the edge represented by edgeid. By default it just subdivides it in half, but when given the optional argument divisions it subdivides the edge by that number. Then it returns a list of all the newly created vertices.* ***Result:****[vertexid,...]*

collapseEdge(edgeid). *Collapses the edge represented by edgeid.* ***Result:*** *vertexid*

Finding IDs

faces([selected]). *Returns a tuple containing the ids of the faces. If selected is True, then only those faces selected are returned (this is the default in the TopMod interface), otherwise all the faces. Since selection is impossible in stand-alone, the default is all the faces.* ***Result:*** *(faceid,...)*

edges([selected]). *Returns a tuple containing the ids of the edges. If selected is True, then only those edges selected are returned (this is the default in the TopMod interface), otherwise all the edges. Since selection is impossible in stand-alone, the default is all the edges.* ***Result:*** *(edgeid,...)*

verts([selected]). *Returns a tuple containing the ids of the vertices. If selected is True, then only those vertices selected are returned (this is the default in the TopMod interface), otherwise all the vertices. Since selection is impossible in stand-alone, the default is all the vertices.* ***Result:*** *(vertexid,...)*

corners([selected]). *Returns a tuple containing 2-tuples with the ids of the faces and the ids of the vertices. If selected is True, then only those corners selected are returned (this is the default in the TopMod interface), otherwise all the corners. Since selection is impossible in stand-alone, the default is all the corners.* ***Result:*** *((faceid,vertexid),...)*

walk(faceid). *This does a boundary walk around a face. The format looks like this:* `DLFLFace faceid (numsides) : vertexid[backface]--(edgeid)-->....` *It returns a list of vertices and a list of edges. It continues until it walks through each side of the face.* ***Result:*** *vertexlist,edgelist*

cornerWalk(faceid). *Performs the boundary walk as specified by the walk above, but instead of grabbing the vertices and edges walked, it returns the corners walked. This means the result will contain the tuples with the input faceid and each vertex.* ***Result:*** *[(faceid,vertexid),...]*

saveCorner((faceid,vertexid). *Save a face-vertex corner and grab a unique ID from the address in memory.*

restoreCorner(cornerid). *Get the new face-vertex tuple of the corner represented by the unique ID returned by saveCorner.*

Object Management

load(filename). *Loads a file of either Wavefront OBJ (.obj) or DLFL (.dlfl) into the scene. The file data becomes the current object. The filetype is recognized by the extension. Filename is given as a string (e.g., "myfile.obj"). In the TopMod interface this replaces any current object loaded. In stand-alone, this pushes any current object onto the stack.* ***Result:*** *objectid*

save(filename). *Saves a file of either Wavefront OBJ (.obj) or DLFL (.dlfl) from the current object. The filetype is recognized by the extension. Filename is given as a string (e.g., "myfile.obj").* ***Result:*** *wasSuccess*

kill(objectid). *Kills or destroys the object represented by objectid. This is disabled when using the TopMod interface! It serves only when working in python standalone.* ***Result:*** *None*

switch(objectid). *Switches the current object to the object represented by objectid. This is disabled when using the TopMod interface!. Since the interface currently only allows one object to be open at a time.* ***Result:*** *objectid*

Object Information

printObject(). *Print out object information to standard out. For example, a cube:*

```
Number of vertices : 8

Number of faces : 6

Number of edges : 12

Number of materials : 1

Genus : 0
```

vertexInfo(vertexid). *Store information about the vertex in a dictionary. The options can be accessed by there keys like this vertexInfo(vertexid)['coords']. Valid keys are: 'id','type','coords','valence'.* ***Result:*** *'key': value*

edgeInfo(edgeid). *Store information about the edge in a dictionary. The options can be accessed by there keys like this edgeInfo(edgeid)['midpoint']. Valid keys are: 'id', 'type', 'midpoint', 'normal',* *‘cornerA’, ‘cornerB’, ‘length’.* ***Result:*** *‘key’: value*

faceInfo(faceid). *Store information about the face in a dictionary. The options can be accessed by there keys like this faceInfo(faceid)[‘centroid’]. Valid keys are: ‘id’, ‘type’, ‘centroid’, ‘normal’, ‘size’.* ***Result:*** *‘key’: value*

cornerInfo((faceid,vertexid)). *Store information about the corner in a dictionary. The options can be accessed by there keys like this cornerInfo((faceid,vertexid))[‘type’]. Valid keys are: ‘face’, ‘vertex’, ‘edge’, ‘type’.* ***Result:*** *‘key’: value*

centroid(vertexids). *Find the centroid of a given set of vertices.* ***Result:*** *centroid*

Transformations

translate(x,y,z[,relative]). *Translate an object in world space.*

scale((x,y,z)). *Scale an object in world space.*

move(vertexids,(x,y,z)[,relative]). *Move vertices in the list.*

Python Auxiliary Commands

extrude(type,faceid[,...]). *Extrudes a face with the type of extrusions specified by a string. Valid types include:*

- *“doo-sabin” options: [distance, segments, rotation, scale]*
- *“dodeca” options: [distance, segments, twist, scale, hexagonalize]*
- *“icosa” options: [distance, segments, rotation, scale]*
- *“octa” options: [distance, rotation, scale]*
- *“stellate” options: [distance]*
- *“double-stellate” options: [distance]*
- *“cubical” options: [distance, segments, rotation, scale]*

Result: *faceid*

subdivide(scheme[,...]). *Subdivides the current object with the specified scheme. Each scheme has a different set of optional arguments. Valid scheme names are:*

- *“loop”*
- *“checker” options: [thickness=0.33]*
- *“simplest”*
- *“vertex-cut” options: [offset=0.25]*
- *“pentagon” options: [offset=0.0]*
- *“dual-pentagon” options: [scale=0.75]*
- *“honeycomb”*
- *“doo-sabin” options: [check=true]*
- *“doo-sabin-bc” options: [check=true]*
- *“doo-sabin-bc-new” options: [length]*

- *“corner-cut”*
- *“modified-corner-cut” options: [thickness]*
- *“root4” options: [a,twist]*
- *“catmull-clark”*
- *“star” options: [offset=0.0]*
- *“sqrt3”*
- *“fractal” options: [offset=1.0]*
- *“stellate”*
- *“double-stellate” options: [offset,curve]*
- *“dome” options: [length,scale]*
- *“dual-12.6.4” options: [scale]*
- *“loop-style” options: [length]*
- *“linear-vertex” options: [usequads=true]*

Result: *None*

subdivideFace(faceid[,usequads]). *Subdivide a face into n faces (where n is the number of edges of the face). By default the new faces are quadralaterals, but if specified with False, then the new faces will be triangular.* ***Result:*** *None*

subdivideFaces(faceidList[,usequads]). *Subdivide faces in the list into n faces (where n is the number of edges of the face). By default the new faces are quadralaterals, but if specified with False, then the new faces will be triangular. If you want to do all faces you can also Use subdivide(“linear-vertex”).* ***Result:****None*

dual(). *Takes the dual of the current object.* ***Result:*** *None*

connectEdges((edgeid,faceid),(edgeid,faceid)[,loopCheck]). *Connect two half-edges with a face. If loopCheck is True then only connect if the edges are not adjacent to their corresponding* *faces.* ***Result:*** *None*

connectCorners((faceid,vertexid),(faceid,vertexid)[,numsegs,maxconn,'dual']). *Connect two faces given a corner from each face. Uses repeated insertEdge operations.* ***Result:****None*

connectFaces(faceid,faceid[,numsegs,maxconn]). *Connect two faces with multiple segments. Intermediate points are calculated by linear interpolation based on number of segments. Maximum connections should be set to -1 when connecting all is desired.* ***Result:*** *None*

addHandle(interp,(faceid,vertexid),(faceid,vertexid),numsegs,...). *Create an interpolated handle. The option interp should be set to either "hermite" or "bezier" depending on the desired interpolation method. Setting the number of segments and playing with the weights and number of twists will yield different results.* ***Result:*** *None*

rind(facelist,useScaling,percent[,uniform]). *Create a rind model - create a crust then punch the specified faces. The percentage is a thickness parameter when useScaling is false, otherwise it is a scaling parameter.* ***Result:*** *None*

wireframe([thicknes,split]). *Creates a solid wireframe from the object.* ***Result:*** *None*

column(thickness,segments). *Creates a solid column wireframe from the object.* ***Result:*** *None*

VITA

Stuart Tosten Tett

PDI/DreamWorks

1800 Seaport Blvd.

Redwood City, CA 94063

stuart@tett.net

M.S. in Visualization Sciences, Texas A&M University, December 2007

B.S. in Computer Science and Art & Design, Iowa State University, May 2005