TOWARD OPTIMISTIC VERSION CONTROL IN ARCHITECTURE: DIFFING, PATCHING, AND THREE-WAY MERGING FOR OPENNURBS 3D MODELS

by

Nicholas Oren Rawlings

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Architecture
and the degree of Master of Science in
Information Technology

Charlotte

2022

Approved by:

_____
Jefferson Ellinger

_____
John Gero

_____
Jonathan Dessi-Olive

Abstract

NICHOLAS OREN RAWLINGS. Toward optimistic version control in architecture: diffing, patching, and three-way merging for openNURBS 3D models. (Under the direction of JEFFERSON ELLINGER)

The ability of architects to collaborate and work in parallel on digital assets is limited by pessimistic strategies for managing shared files. The software engineering community has worked around this problem by adopting optimistic version control techniques, which rely on the ability to diff, patch, and merge versions of the files they manage. Unfortunately, the diffing, patching, and merging algorithms in existing version control systems are designed to work with text, and not with the types of files, such as 3D models, most commonly used by architects. This thesis describes a set of command line programs capable of diffing, patching, and merging openNURBS models, an open-source 3D model format that enjoys widespread use among architects and other design professionals. Integration of these programs into an off-the-shelf version control system is demonstrated, and an abstract domain model is presented which can be used to apply their capabilities to other file formats as well.

DEDICATION

For Nanny,

who taught me that sometimes one must let go

of worry and instead laugh at the absurdity of life,

and Poppy,

whose patience, humility, and intelligence

have been an inspiration to me for as long as I can remember.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Jefferson Ellinger, and committee members John Gero and Jonathan Dessi-Olive for generously sharing their time and wisdom over the past several months. I would also like to recognize former advisor Dr. Dimitris Papanikolaou, who pushed me to greater levels of academic rigor than I had dreamed possible, as well as Eric Sauda and Dr. Mirsad Hadžikadić, who provided invaluable guidance during the earliest stages of this thesis.

Additionally, I would like to thank the following current and former members of the faculty and staff of the School of Architecture for their support and mentorship over the past eighteen years:

Mona Azarbayjani	Zhongjie Lin	Deb Ryan
Blaine Brownell	Ken Lambla	Robbie Sachs
Alex Cabral	Emily Makaš	Eric Sauda
Kelly Carlson-Reddig	Marc Manack	Michael Swisher
Rachel Dickey	Liz McCormick	Greg Synder
Phil Gaddy	Matt Parker	David Thaddeus
Lee Gray	Todd Payne	Betsy West
Chris Grech	Nicole Perri	Peter Wong
Josie Holden Bulla	Rich Preiss	Chengde Wu
Lidia Klein	James Reittinger	Catty Zhang

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
B-rep	Boundary Representation
BIM	Building Information Modeling
CAD	Computer-Aided Design
DVCS	Distributed Version Control System
GUI	Graphical User Interface
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
NURBS	Non-Uniform Rational Basis Spline
SDK	Software Development Kit
UUID	Universally Unique Identifier
VCS	Version Control System

CHAPTER 1

INTRODUCTION

In the not-so-distant past, when architectural drawings were rendered exclusively by hand, the constraints of physical space made it impractical for more than one draftsman to work on the same drawing at the same time. Today, computers have liberated architecture professionals from the physical constraints of the drafting board, yet concurrent work on a single drawing or model remains problematic, if not impossible. This persistent limitation restricts the ability of design teams to collaborate, iterate, and explore alternative solutions where digital assets are involved.

Most design software today employs some form of *resource locking*, which actively prevents a second user from modifying a digital asset as long as the first one is working in it. Often times, the digital asset is a shared file on a networked storage device. Because a single file may represent significant portions of, if not an entire design project, locking this file has the effect of shutting out any potential real-time digital collaborators. Resource locks can also occur at finer levels of granularity than an entire file: In database management systems, it is common for locks to be applied only to a single table or record at a time, and in the case of

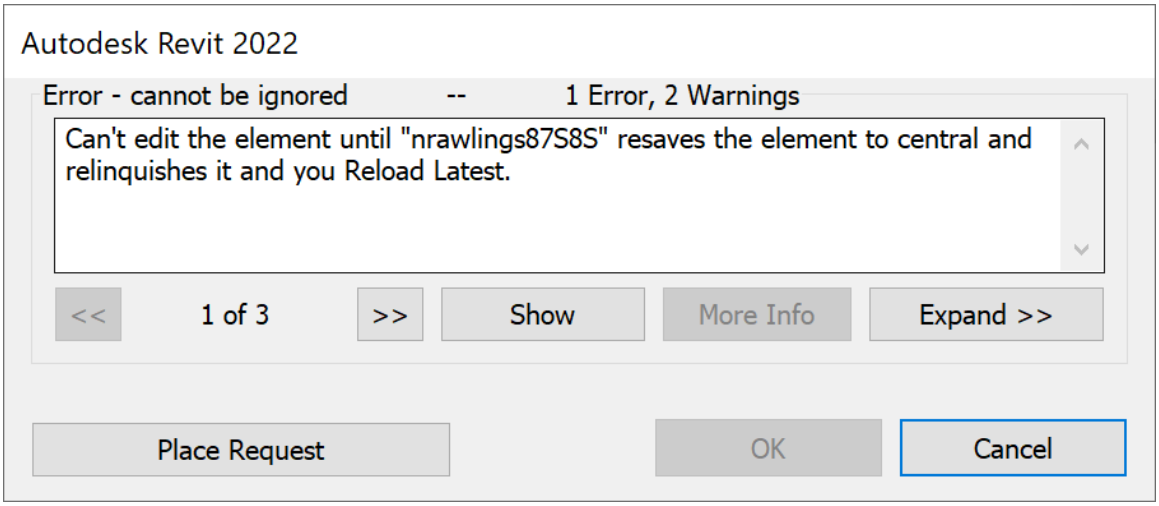


Figure 1.1. Resource locking in action. Revit displays an error message like this one when a user attempts to modify a model element that has been locked by another user. When confronted with this situation, the first user has no choice but to abort the task they were attempting to complete and wait for the second user to release the lock.

Autodesk's Revit software, elements within a workshared building information model may be locked individually (Figure 1.1). However, greater granularity does not solve the underlying problem of resource locking; it merely splinters it into scores of smaller potential conflicts.

Resource locking exists to prevent data loss and corruption due to conflicting edits; if only one user is allowed to modify a file at any given time, no conflicts can occur. Although this is a straightforward and effective strategy for handling conflicts over shared digital assets, it is fundamentally *pessimistic* because it assumes that parallel edits to a file will invariably result in data corruption. The wholesale embrace of this form of digital pessimism by contemporary design software represents a significant obstacle to parallel work within a design team (Aish 2000).

In the field of software engineering, parallel work on digital files is made possible by the use of version control systems (VCSs) such as Git, Mercurial, and Subversion. The nominal purpose of these systems is to track the changes made to a codebase over time, but they also enable developers to work independently on parallel versions of a project (known as branches) and to later integrate (or merge) their contributions with those of their teammates with little or no extra effort. Modern VCSs are said to be *optimistic* because they assume that conflicting edits to the codebase are rare and that those that do occur can usually be resolved automatically (Kung and Robinson 1981).

Optimistic version control relies on the ability to find the differences between two versions of a file (a process known as *diffing*), to apply a list of differences to a file to transform it from one version to another (*patching*), and to combine lists of differences from divergent versions of a file to arrive at a single, unified result (*merging*). Well-established and highly optimized algorithms exist for performing these tasks on *plain text* files such as those used by developers to store source code, which consist solely of sequences of characters grouped into lines. The straightforward and predictable structure of plain text files allows this relatively simple set of algorithms, which form the basis of contemporary VCSs, to provide diffing, patching, and merging functionality for a wide variety of programming languages and data formats.

Not all files, however, are composed of plain text. Images, CAD drawings, and 3D models — the types of files most frequently used by architects — consist of idiosyncratic representations of both textual and non-textual data. The diffing,

patching, and merging algorithms used by existing VCSs are unable to operate meaningfully or efficiently on these so-called *binary* files because they cannot be interpreted as a linear sequence of standard character codes. In fact, existing VCSs will not even attempt to calculate diffs or perform merge operations on binary files, and will instead fall back to older, more pessimistic methods of handling them that do not allow for parallel modes of working (Chacon and Straub 2014; Collins-Sussman, Fitzpatrick, and Pilato 2011). The difficulty in dealing with binary files is that the structure and semantics of each binary format is essentially unique. Just as each image format uses its own method of storing pixels and each type of 3D model employs a different means of representing geometry, the algorithms for diffing, patching, and merging these disparate data formats must be similarly specialized. In order to bring parallel working and other benefits of modern VCSs to the field of architecture, systems for diffing, patching, and merging each of the various file formats used by architects must be designed and implemented.

This thesis describes a set of programs that implement diffing, patching, and merging algorithms for a 3D model format commonly used by architects, namely the openNURBS format native to McNeel's Rhinoceros 3D modeling software. This format was chosen for its open-source status and its widespread use during the early stages of design when collaboration and rapid exchange of ideas is crucial. Although openNURBS is the focus of these programs, their underlying data model is designed to be abstract enough to support diffing, patching, and merging of other file formats as well.

This thesis also demonstrates how its diffing and merging programs can be integrated into an off-the-shelf VCS, thereby enabling optimistic version control for openNURBS models under its purview. In doing so, this thesis takes the first steps toward bringing the kind of collaborative and concurrent work currently enjoyed by software developers to the practice of architecture.

CHAPTER 2

BACKGROUND

2.1 Version Control Systems

A version control system (VCS) is a set of software tools that track and manage the changes made to a collection of files over time. The collection of files and their respective histories managed by a version control system is commonly known as a *repository*. Although traditionally used for organizing the source code of computer programs, version control systems are now being used by writers, lawyers, journalists, musicians, graphic designers, and others.

The first software that can be recognized as a version control system was developed at Bell Labs beginning in 1972. The Source Code Control System (SCCS) was notable for its space-efficient storage system, which accumulated all versions (both past and current) of a document into a single file (Rochkind 1975). One downside to this approach was that the system would become progressively slower with every new version stored. To work around this and other deficiencies in SCCS, the Revision Control System[1] (RCS) was developed around the concept

1. https://www.gnu.org/software/rcs/

of reverse deltas. RCS stores the most recent version of a file along with a separate reverse delta to describe each of its version transitions; it reconstructs previous states of the file by applying those reverse deltas sequentially to "rewind" to the target version. RCS is significant for introducing the concept of symbolic labels, better known today as *tags*, that could be used to reference specific versions of a file independently of its internal version number. It also included the ability to create and merge *branches* — parallel histories by which a developer could experiment in a file without interfering with the work of their teammates (Figure 2.1). However, the significance of this feature was unappreciated at the time and its use was generally avoided (Tichy 1982, 1985; Ruparelia 2010).

Both SCCS and RCS are limited to working with a single file and on a single computer at a time. The first of the so-called second-generation VCSs, known as Concurrent Versions System[2] (CVS), was originally implemented as a set of shell scripts that extended RCS to allow it to work with multiple files and over a network. CVS places the version control repository on a central server from which clients can *check out* an existing version of a file, work on a copy of that file independently of other users, and then *check in* their changes once complete. It also popularized branching and merging operations to enable the kind of parallel, collaborative workflows that are the hallmark of optimistic version control (Grune 1986; Berliner 1990; Ruparelia 2010). The successor to CVS, named Subversion[3],

2. https://www.nongnu.org/cvs/

3. https://subversion.apache.org/

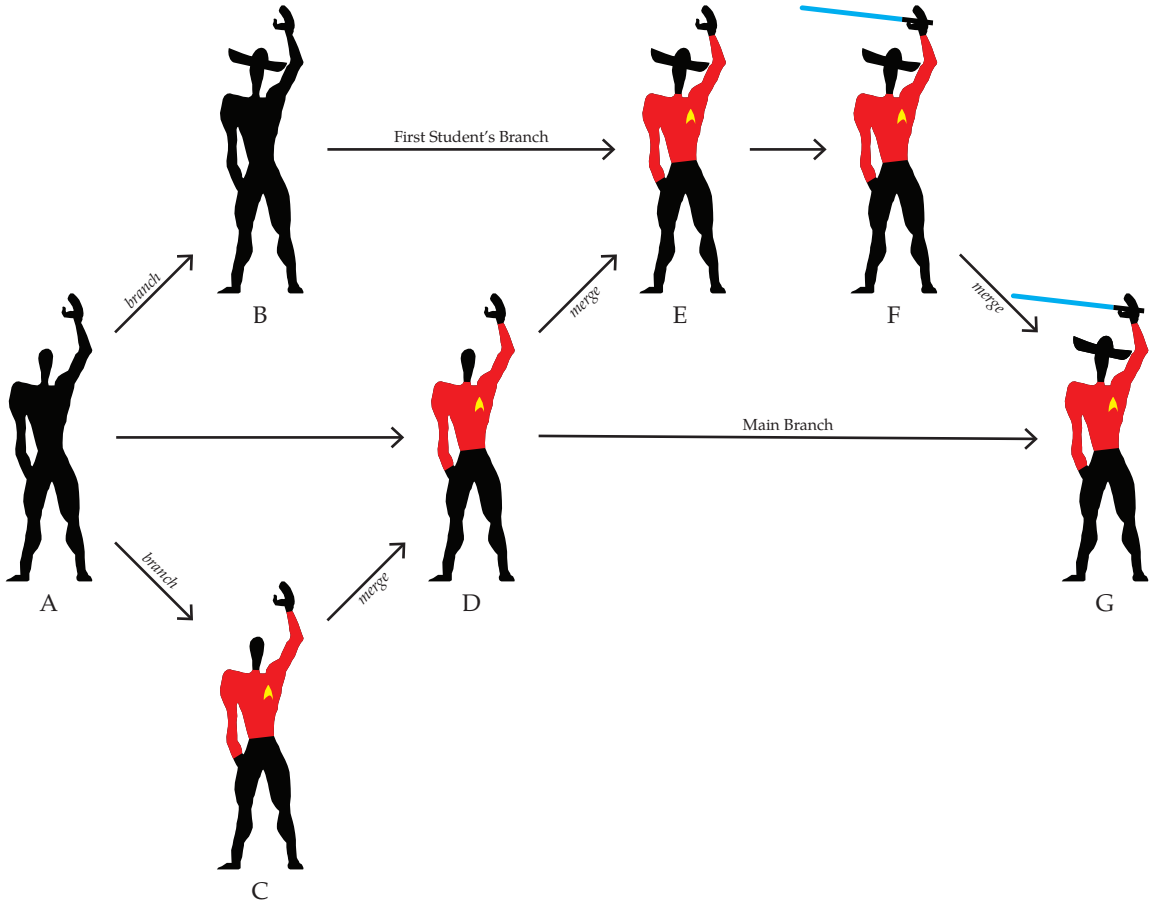


Figure 2.1. Branching and merging. Two students decide to create their own variations on Le Corbusier's iconic Modulor Man (A). The first sends him on vacation in Mexico (B) while the second assigns him to duty aboard the USS Enterprise (C). Confident of her changes, the second student merges them into the main branch (D). The first student subsequently merges those changes into his own branch (E). Woefully deficient in his knowledge of science fiction franchises, he then decides to equip the Modulor Man with a lightsaber (F). The first student then merges his changes into the main branch as well (G).

was initially released in 2000 and focused on fixing flaws in CVS rather than new innovations in version control design (Collins-Sussman, Fitzpatrick, and Pilato 2011). Its emphasis on power and usability has made it the preferred choice for centralized version control.

In contrast to their predecessors, the third generation of VCSs are inherently decentralized. These *distributed* version control systems (DVCS), which include Git[4], Mercurial[5], and Bazaar[6], maintain a full copy of the repository on each machine where it is being used to eliminate the single point of failure presented by a centralized server. Such systems are also more performant than their centralized counterparts since most operations can be performed locally without the need for network access to communicate with a central server (Chacon and Straub 2014). DVCSs, and Git in particular, manage the majority of version control repositories in use today.

The decentralized nature of DVCSs does not preclude the use of a central server; in fact, most projects using a DVCS rely on a central repository to facilitate coordination among team members and to act as a single source of truth for build processes and public releases. This central repository often resides on a repository

4. https://git-scm.com/

5. https://www.mercurial-scm.org/

6. https://bazaar.canonical.com/

hosting service such as GitHub[7], GitLab[8], or Bitbucket[9], which provide a web-based graphical user interface (GUI) to the VCS and integration with tools such as bug trackers, build systems, and wikis. These services have contributed to the popularity of DVCSs, as well as to the adoption of VCSs in general outside the software development community.

2.2 Diffing, Patching, and Merging

Diffing is the process of finding the differences between two versions of a file. In the same way that coordinate geometry allows us to subtract two points to produce a vector that translates between them, diffing can be thought of as the subtraction of two files to produce a directed delta that contains the information necessary to transform one of those files into the other. *Patching* is the process of applying that delta to one version of a file to transform it into another version. If diffing is comparable to subtracting two points in coordinate geometry, then patching is analogous to adding a vector and a point to arrive at a new point (Figure 2.2).

Algorithms for diffing and patching are among the most fundamental components of any VCS. The term *diff* itself originated from a program developed by James Hunt, Thomas Szymanski, and Douglas McIlroy at Bell Labs in the early

7. https://github.com/

8. https://about.gitlab.com/

9. https://bitbucket.org/

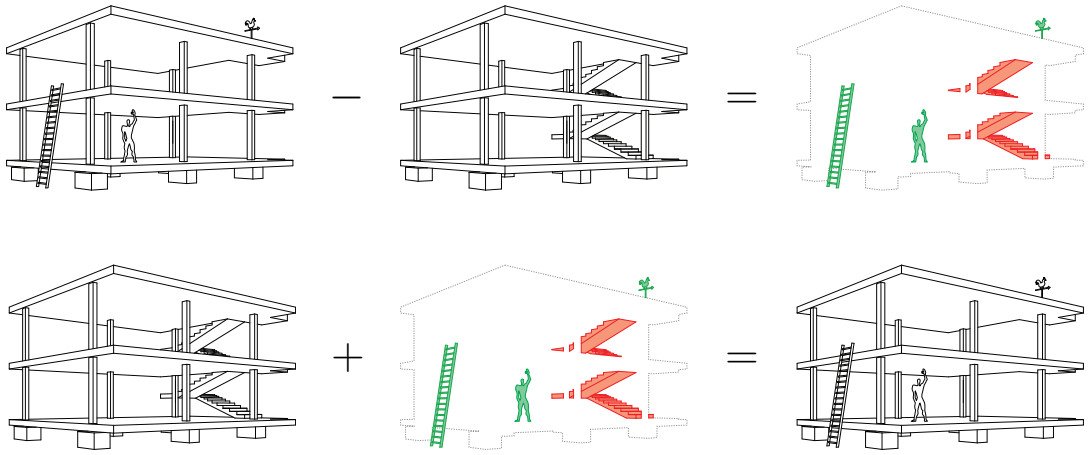


Figure 2.2. Diffing and patching. Some misguided soul has "improved" upon Le Corbusier's Maison Dom-ino by removing the staircase and adding a ladder, a weather vane, and a human figure. Diffing the two versions of this iconic image is analogous to subtracting the original version from the new one (top row). Patching is analogous to adding that difference to the original version to recreate the new one (bottom row).

1970s. It relies upon a novel solution to the longest common subsequence (LCS) problem to detect which groups of lines are shared between two text files, and thus indirectly indicates which lines have changed (Hunt and McIlroy 1976). A more efficient algorithm for solving the LCS problem discovered in the 1980s forms the basis of all diffing operations carried out by modern version control systems (Myers 1986).

Merging is the process of combining the sets of changes made in two divergent versions of a file to arrive at a unified result (Figure 2.3). Users of VCSs employ merging to reconcile the changes made by others with their own working copy of a project, thereby bringing their copy up to date with the rest of the team. Because of its role in eliminating the need for resource locks, it is not an

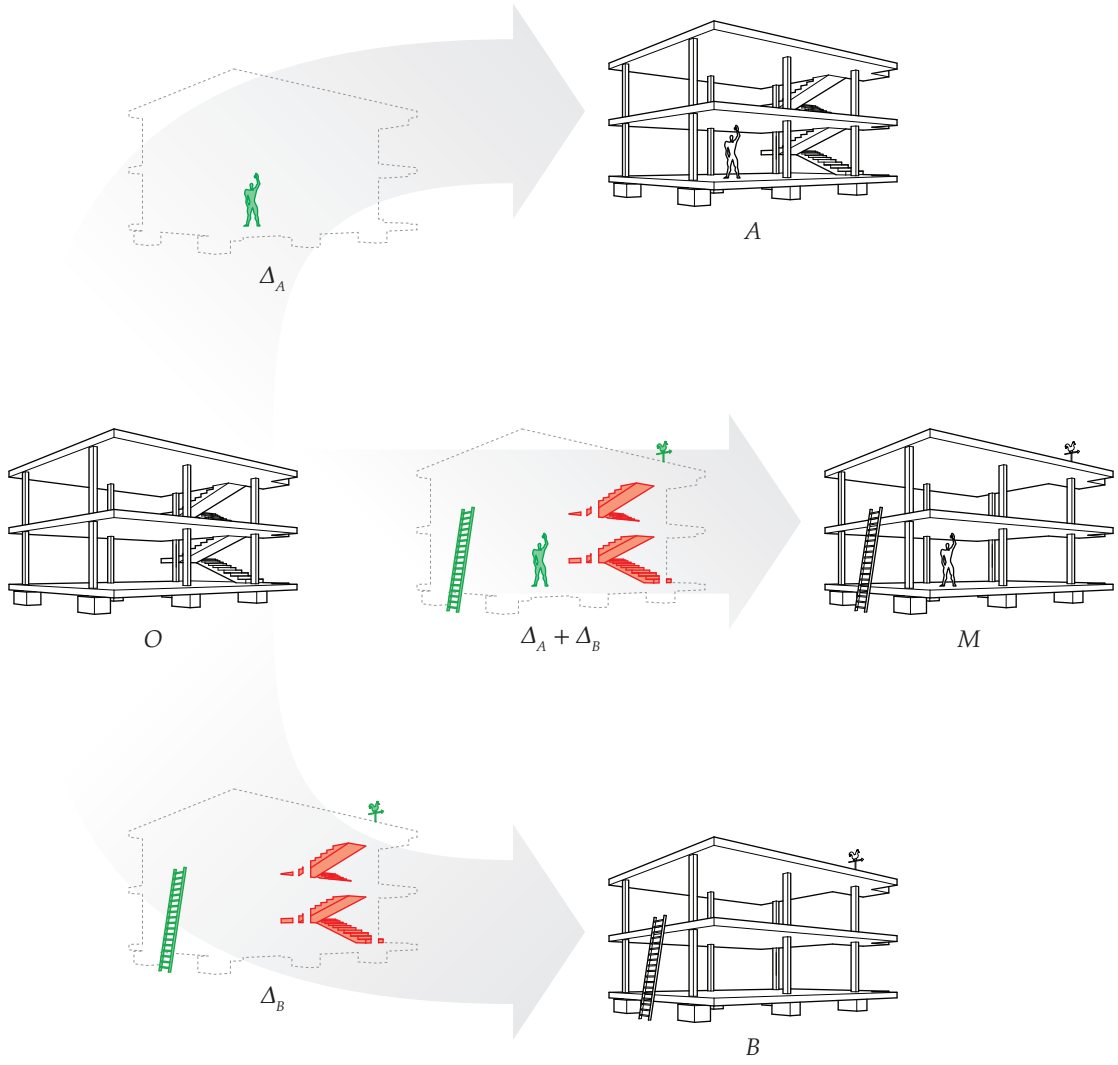


Figure 2.3. Three-way merging. A three-way merge requires that the two files being merged (A and B) originate from a common ancestor (O). The operation compares each of the two files with the common ancestor (Δ_A and Δ_B) to arrive at a more accurate result (M) than was possible with older and less reliable two-way merging algorithms.

exaggeration to state that merging is what makes optimistic version control possible (Mens 2002).

The merging algorithms used by nearly all contemporary VCSs are designed to work with plain text, a data format consisting purely of sequentially-arranged, human-readable characters (The Unicode Consortium 2021). Regardless of their content, all plain text files adhere to a common structure consisting of sequences of characters grouped into lines. The power and popularity of these algorithms arise from this consistency; because they are able to ignore the syntactic and semantic content of the files they are given, these algorithms can operate equally well on C source code as with the LaTeX files used to typeset this thesis. Neither obscure nor yet-to-be-invented file formats pose any challenges so long as their contents are textually encoded.

Not all files, however, are composed of plain text. Binary files such as images, videos, and 3D models consist of domain-specific and application-dependent encodings of both textual and non-textual data. Because these files do not adhere to a consistent structure of characters and lines, the merging algorithms embedded in most contemporary VCSs are unable to operate effectively on them (MacKenzie, Eggert, and Stallman 2021a). The idiosyncratic structure of binary files makes it likely that any attempt to merge them using textual merging algorithms would result in syntactically invalid and possibly semantically nonsensical output — images that won't load, videos that won't play, and 3D models that crash their modeling programs.

Contemporary VCSs handle any modification to binary files, no matter how small, by storing an entirely new version of the file in the repository. This strategy precludes the benefits of change tracking and asynchronous, parallel collaboration for such resources, making it the single largest technical hurdle to the availability of optimistic VCSs for the architecture profession. A number of structurally- and semantically-aware algorithms that could enable merging of binary data have been proposed (Rönnau, Scheffczyk, and Borghoff 2005; Chen, Wei, and Chang 2011), but none have gained widespread acceptance in version control applications.

2.3 Benefits of Optimistic Version Control in Architecture

The vast majority of design professionals today handle the storage of alternative and past versions of a project by manually copying and renaming files. This highly idiosyncratic process is prone to errors and data loss and imposes an unnecessary cognitive burden on the computer user (Ashtari 2018; Cristie and Joyce 2021). VCSs offer a standardized, automated, and reliable method of managing alternative and past versions of a design that scales to accommodate design teams of varying sizes. They enhance the collaborative potential of design teams by enabling their members to work in parallel. Beyond these more prosaic benefits, adoption of version control techniques may actually help to augment an architect's creative potential as well.

Modern VCSs enable a non-linear view of history that encompasses not only the depth of time but also the breadth of alternatives inherent in any design project. The processes of differentiating and synthesizing alternatives are known respectively as branching and merging within the parlance of version control systems, and offer compelling parallels to the patterns of divergence and convergence seen in the design processes (Cross 2006). Adopting version control practices could heighten architects' awareness of their place in the design process, and thereby increase their capacity for reflective practice (Schön 1984; Cristie, Ibrahim, and Joyce 2021).

Placing a project under version control creates a digitized memory of every step (and misstep) of that project's development. This memory can augment the designer's own thought processes and facilitate idea navigation during the design process, and can afterward act as an archive not only of the final outcome of the design process but also of the designer's ideas and intent as well (Cristie, Ibrahim, and Joyce 2021). Such an archive could be consulted by the designer or his associates in the months and years after a project has concluded, or by architectural historians decades later. It can also provide novice designers valuable insight into the more decisive decision-making process of their expert colleagues (Cross 2004).

Within the scope of architectural education, version control could also facilitate a digital reincarnation of the open design atelier, which has suffered in recent years due to the increasing prevalence of digital-only workflows as well as physical distancing driven by the recent pandemic (Meagher et al. 2015).

Recognizing the need for tooling specifically designed to handle multiple versions of a design, some architecture firms have begun to take advantage of cloud-based document management systems such as Autodesk's BIM 360. While these products represent a huge step forward in terms of collaboration and data integrity, they still suffer from the constraints of resource locking. Furthermore, they are limited to a linear model of the project's history, making them poorly suited to handling multiple design alternatives in the early stages of a design project. Only optimistic version control can enable the kind of unrestricted parallel work currently enjoyed by software developers in the field of architecture (Schneider 2011; Firmenich et al. 2005).

2.4 Research into Version Control for Architecture and Related Disciplines

The need for version control tools in architectural practice has been recognized since at least 2000, when Robert Aish reported on the implementation of "long transaction" and "change merge" features (which would be recognized today as branching and merging, respectively) in Bentley's since-discontinued ProjectBank product (Aish 2000). However, the most promising work in the development of diffing and merging algorithms for visual designers has instead dealt with file formats used by digital animators and illustrators for 3D modeling. Doboš and Steed (2012) proposed the first system for diffing and merging these 3D scene graphs. Their node-based method relies on matching the universally unique identifiers (UUIDs) assigned to model components to determine which

ones have been added to and removed from the model. However, their approach to change detection is insufficiently granular to support collaboration.

3D scene graphs typically use *meshes*, networks of planar, polygonal surfaces, to approximate complex forms rather than attempting to model their curvature directly. MeshGit is an algorithm for determining the edit distance between two meshes as a means of implementing diffing and merging for that style of geometry (Denning and Pellacini 2013). Its original implementation suffered from a number of performance issues and was subsequently improved in MeshHisto to be precise and scalable enough for real-time collaborative editing (Salvati et al. 2015).

SceneGit likewise improves on MeshGit through the addition of performance-enhancing heuristics. It defines a comprehensive set of data structures and algorithms for diffing and merging not only the geometry of a 3D scene graph, but also its materials, lighting, and camera configuration. Because it focuses on interchange formats (namely OBJ and gITF) in which UUIDs are frequently missing or unreliable, SceneGit also implements a novel method of deriving unique but stable object identifiers across file versions. SceneGit represents the state of the art in diffing and merging 3D models at this time (Carra and Pellacini 2019).

Doboš et al. (2018) proposed an alternative method of detecting differences between different versions of a 3D scene by comparing the pixels of coordinated rendered views. The downside of this approach is that it is unable to detect differences that are not visible in the rendered view, such as small changes

in the geometry of a large-scale model or modifications to non-visible attributes, making it unsuitable for use in a VCS.

Daum and Borrmann (2016) present a system for diffing and merging architectural models that employ the Industry Foundation Classes (IFC) schema, an open standard for storing building information model (BIM) data. GeomDiff includes a similar set of algorithms for geospatial data, notably describing changes in geometry in terms of the movement of points rather than the substitution of new values for old ones (Sveen 2020).

GHShot is a web-based VCS for Grasshopper scripts, a visual programming language used by architects to develop parametric models in Rhinoceros. In this system, a custom Grasshopper component communicates with a server to record the editing history of a Grasshopper script. Unfortunately, this approach lacks the necessary level of separation between the VCS and the content under version control: In order for a Grasshopper script to be managed by GHShot, it must itself contain the GHShot software in the form of the custom component (Cristie and Joyce 2017; Cristie, Ibrahim, and Joyce 2021).

Sakai and Tsunoda (2015) created LMNArchitecture, a web application that presents the evolution of simple architectural models as an interactive tree diagram. The models are stored in a text-based format so that off-the-shelf diffing and merging programs can be used.

Most existing VCSs take a state-based approach to merging that considers only the differences between two versions of a file and not how those differences arose. In contrast, Koch and Firmenich (2011) propose an operation-based

approach to collaboration within an architectural model whereby the actions taken by one user are transmitted to another user's computer and applied to that user's copy of the same model, in much the same way that online collaborative editors such as Google Docs synchronize changes among users. Operation-based merging tends to benefit from better conflict detection and resolution than state-based approaches, but also requires deep integration with the software used to edit the files in question.

CHAPTER 3

METHODS

3.1 OpenNURBS

Optimistic version control has the potential to confer a multitude of benefits to the architecture community, including enhanced collaboration and parallel working, but its adoption is currently stymied by a lack of three-way merging algorithms that work with the binary file formats commonly used by architects. This thesis demonstrates diffing, patching, and three-way merging algorithms for one such file format, *openNURBS*, which was developed by Robert McNeel & Associates for its Rhinoceros[1] 3D modeling software. Typically associated with the `.3dm` file extension, openNURBS excels at representing complex curves and surfaces using non-uniform rational basis splines (NURBS) but also supports meshes, subdivision surfaces, Bézier splines, and elementary geometries such as points, lines, and planes. McNeel has released openNURBS, as well as a software development kit (SDK) for reading and writing files in that format, as open source under an MIT-like license. This license allows anyone to integrate

1. https://www.rhino3d.com/

openNURBS into their own software, or to publish enhancements to openNURBS itself, increasing the likelihood that openNURBS will become a widely-accepted data exchange format in the future (Robert McNeel & Associates, n.d.).

The openNURBS SDK provides an object-oriented application programming interface (API) for examining and manipulating 3D models in the openNURBS format. Object-oriented programming (OOP) rose to prominence in the 1980s and 90s with the emergence of graphical user interfaces (GUIs) and languages such as C++ and Java and has remained the dominant programming paradigm ever since. OOP centers on the concept of the *object*, "an individual, identifiable item, either real or abstract, which contains data about itself and descriptions of its manipulations of the data" (Armstrong 2006). The data storage mechanisms of an object are variously referred to as its *attributes*, *fields*, or *properties*, and the procedures it can perform are known as its *methods*. Objects are typically derived from a *class*, which provides a template for the organization and behavior of a set of related objects (known as *instances* of the class) and typically represents some real-world concept or entity. Sets of classes are used in OOP to model (in the conceptual sense) the problem domain of a software system.

OpenNURBS presents a 3D model as an instance of the `ONX_Model` class, which provides access to the model's components, settings, and metadata as well as methods for reading and writing `.3dm` files. A *component* of a model is any of its constituent parts, such as a layer, line type, or geometric entity. All components are instances of the `ON_ModelComponent` class, and as such, share certain common attributes and behaviors. Each is identified by a UUID, which is unique across all

possible openNURBS models, and an index, which defines its position in the `.3dm` file relative to other components of the same type. Each component also has a name attribute, by which it can be assigned a human-readable label, and a status to indicate whether it is locked or hidden.

In OOP, *inheritance* refers to the ability of one class to incorporate and extend the properties and behaviors of another. The relationship between a child class and its parent is analogous to the relationship between a square and a quadrilateral: The square inherits the characteristics of the quadrilateral (such as having four sides) and builds upon them by defining new characteristics (such as those four sides having equal length) of its own. OpenNURBS defines sixteen child classes of `ON_ModelComponent` that play specific roles within a model while inheriting the common characteristics described in the previous paragraph. They are:

`ON_Bitmap` An embedded raster image.

`ON_DimStyle` A collection of settings, including arrowheads and number formats, that can be applied to annotation objects such as dimension lines and text notes.

`ON_Group` A collection of related geometric objects.

`ON_HatchPattern` A pattern of lines used to fill a hatched region.

`ON_HistoryRecord` A connection between a complex geometric object and the simpler ones used to create it that permits modifications to the simpler objects to be passed on to the more complex one.

`ON_InstanceDefinition` A reusable collection of geometric objects. Unlike an `ON_Group`, multiple references to an instance definition can be placed in a model and will remain linked so that modifications to one are propagated to the others.

`ON_Layer` A category into which geometric objects can be sorted.

ON_Linetype A pattern of dashes and spaces that can be applied to a line or other curve.

ON_Material A set of physical and optical properties that affect the appearance of a rendered object.

ON_ModelGeometryComponent A geometric object such as a point, curve, surface, or solid.

ON_TextStyle A collection of type-related settings, including font name and weight, that can be applied to textual objects.

ON_TextureMapping A set of parameters that control how a texture is projected onto a non-planar surface during rendering.

Most of these classes expose a fairly straightforward, if somewhat lengthy, list of properties and methods pertaining to the aspects of the model they represent. For example, the `ON_Material` class defines a `m_shine` property to describe the shininess of the material it represents. The `ON_Linetype` class exposes an `AppendSegment` method to add a new dash or space to a line pattern and a `DeleteSegment` method to remove an existing one.

Instances of `ON_ModelGeometryComponent`, which comprise the bulk of a typical openNURBS model, are organized somewhat differently in that their characteristics are stored in a pair of objects nested within the geometry component. One of these nested objects is an instance of `ON_3dmObjectAttributes` and is responsible for holding non-geometric information such as the component's color, layer, and material. The other stores the geometry itself and is an instance of one of the dozens of child classes of `ON_Geometry`. Each child class represents a different type of geometrical object with a distinct set of properties: An `ON_Point` represents a point and contains a

single coordinate triple, and an `ON_LineCurve` represents a line defined by a start point and an end point. Appendix B provides a complete listing of the types of geometry supported by openNURBS.

3.2 Programming Languages

OpenNURBS is written in C++[2], a statically-typed compiled language designed by Bjarne Stroustrup and first released to the public in 1985. C++ is known for its speed and efficiency and is most commonly used today in resource-constrained environments, such as microcontrollers, and computationally intensive applications such as graphics. However, C++ also has a reputation for being difficult to write, debug and comprehend, and lacks features such as automatic memory management that are expected of a modern programming language.

Bindings for openNURBS are also available in three other programming languages through a collection of libraries known as rhino3dm[3]. The first of these languages, Python[4], is a dynamically-typed interpreted language created by Guido van Rossum and first released in 1991. Its design philosophy, which emphasizes simplicity, readability, and unambiguousness, has led Python to

2. https://isocpp.org/

3. https://github.com/mcneel/rhino3dm

4. https://www.python.org/

become one of the most widely-used programming languages in contemporary software development (Peters 2004; TIOBE 2022).

Rhino3dm also offers a library for JavaScript, a dynamically-typed interpreted language most commonly used in web applications. JavaScript was created by Brendan Eich at Netscape and was first introduced in that company's Navigator web browser in 1995. The language was subsequently adopted by Microsoft for the Internet Explorer browser and later standardized as ECMAScript[5] in 1997.

C#[6] is a statically-typed compiled language developed by Anders Hejlsberg at Microsoft and first published in 2000. Although the C# language was standardized by ECMA in 2002[7] and ISO/IEC in 2003[8], the Microsoft-produced compiler and associated .NET libraries remained proprietary until the mid-2010s. Even today, C# is typically associated with applications developed for Microsoft's Windows operating system. McNeel's Rhinoceros 3D modeling software, which originated on Windows, is written in C#, and many programs that read and write openNURBS models rely on its C# bindings.

Of the four programming languages described above, only the original C++ implementation of openNURBS offers complete access to all facets of an openNURBS model. The C# library contains the vast majority of the functionality

---

5. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

6. https://learn.microsoft.com/en-us/dotnet/csharp/

7. https://www.ecma-international.org/publications-and-standards/standards/ecma-334/

8. https://www.iso.org/standard/75178.html

present in the C++ SDK, and the Python and JavaScript libraries are each subsets of what is available through C#. Initial explorations for this thesis were conducted in C++ in anticipation of a program that would eventually need access to virtually all parts of an openNURBS model. However, it became apparent that the idiosyncrasies of the C++ language posed an unacceptable impediment to rapid prototype development. Therefore, development was switched to Python, a language with broad support from the open source community and which would support rapid iteration of prototype applications for diffing, patching, and merging openNURBS models.

During the development of this thesis, the limitations of the Python bindings to openNURBS necessitated contributions to the open-source rhino3dm project. These contributions are detailed in Appendix C.

3.3 The Unix Philosophy

Previous investigations into the intersection of version control and design have tended to suffer from an overly broad focus which has led to the development of expansive platforms rather than purpose-specific tools. Several have implemented entirely new geometry- or design-oriented version control systems instead of developing discrete tools for diffing or the visualization of diffs that could expand on the strengths of existing VCSs. Some have even implemented their own 3D modeling environments (Sakai and Tsunoda 2015) or added such extraneous features as real-time chat (Zhang 2021), essentially

"reinventing the wheel" by creating new implementations of already well-defined applications.

In contrast, this thesis seeks to develop a set of programs that are intentionally narrow in scope and are designed to work as components of a larger system rather than as systems unto themselves. The three programs will focus on the problems of diffing, patching, and three-way merging of openNURBS models, respectively, and will ignore "superfluous" issues such as the visualization of diffs or the creation of a graphical user interface (GUI) for merging openNURBS models. Although such features may be desirable in the broader scope of optimistic version control for architecture, they are not immediately relevant to solving the problems of diffing, patching, and three-way merging of openNURBS models.

In maintaining a narrow scope for its programmatic output, this thesis abides by the *Unix philosophy* which, in the words of Douglas McIlroy, states that developers should:

> Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface. (Salus 1994, 52)

The Unix philosophy encourages the development of modular and reusable tools from which solutions to larger problems can be composed. As an example, let us consider the traditional Unix commands `ls` and `wc`. The `ls` command lists the names of files and folders in a directory, printing each name on a separate line of the output. The `wc` command, when used with the `-l` option,

counts the number of lines in a file. By piping the output of ls into `wc -l`, we create a new command that counts the number of files and folders in a directory. This combined command, `ls | wc -l`, accomplishes a more complex task that neither of its simpler constituent commands can do alone.

The programs developed as part of this thesis are meant to be used in combination with other programs as well. By combining them with an existing VCS, end users can enjoy the full breadth of features provided by that VCS and have those features extended to openNURBS models. This thesis will demonstrate the integration of its programs into a Git repository so that common Git operations such as commits and merges can be performed on openNURBS models transparently and in a fully optimistic manner.

3.4 Delta Format

The Unix philosophy also encourages the use of plain text as a data exchange format between programs. Text is a "universal interface" not only in the sense that it can be read by multiple programs, but also because it can be interpreted by humans as well. Plain text has the capacity to be *self-describing* by including not only data, but also the context necessary to understand those data. Storing information in self-describing plain text files greatly simplifies testing of software that uses that information and allows for further processing and analysis of that information by other applications (Hunt and Thomas 2000).

The output format of the diffing program described in this thesis (and, consequentially, the input format of its patching program) is inspired by the plain text "unified" format supported by the standard `diff` and `patch` commands and favored by most modern VCSs (MacKenzie, Eggert, and Stallman 2021b). It is hoped that by mimicking the unified diff format, the openNURBS delta format described here may retain at least some compatibility with existing diff analysis tools.

As in a unified diff, an openNURBS diff begins with a two-line preamble that lists the names of the two files that were compared and the times at which they were last modified. For example, the preamble

```
--- olderModel.3dm 2022-10-02 13:47:06.959975 -0400
+++ newerModel.3dm 2022-10-03 15:23:11.582114 -0400
```

corresponds to a diff between two files, `olderModel.3dm` and `newerModel.3dm`, that were last modified roughly 25 hours and 36 minutes apart at the beginning of October 2022.

Following the preamble, both a unified diff and an openNURBS delta consist of a series of *hunks*, or groups of related changes. In the unified format, a hunk describes a set of changes that occur within a few lines of one another. Because openNURBS is a binary format that does not have lines, the hunks in an openNURBS delta correspond to the sets of changes made within individual model components.

Each hunk starts with a header line that indicates the type of component, its UUID, and the operation performed on it and begins and ends with a pair of `@`

symbols. The operation is encoded in the first non-whitespace character after the initial `@` symbols: a plus sign indicates that the component was added to the model, a minus sign indicates that the component was removed from the model, and a tilde indicates that the component was modified. The component type, which immediately follows the operation symbol, corresponds to the list of component types presented in Section 3.1 unless the component is an instance of `ON_ModelGeometryComponent`, in which case the name of its geometry type is used instead. For example, the line

```
@@ ~Layer 6f9cb61e-f062-4751-aca1-5cc587ccd0ae @@
```

introduces the hunk for a modified layer identified by a UUID beginning with `6f9c`, and the line

```
@@ +LineCurve f497799f-c237-4219-b456-80250c0ea593 @@
```

introduces the hunk for a newly created geometric component that contains a `LineCurve` and is identified by a UUID beginning with `f497`.

The openNURBS delta format simplifies the openNURBS data model by depicting components as having a flat set of properties that can be expressed as a list of key-value pairs. Whereas the openNURBS SDK places the `CastsShadows` property on an `ON_ObjectRenderingAttributes` object which is nested inside an `ON_3dmObjectAttributes` object which is linked to the component, the openNURBS delta format presents that property as belonging directly to the component itself. In doing so, it presents a readily comprehensible data model to

end users who wish to inspect the output of a diff and streamlines the reconciliation process during merges.

A list of colon-separated key-value pairs follows the header of each hunk. For components that were added to the model, this list includes all of the properties of the component whose values differ from their defaults, which provides sufficient information to recreate the component in a future patch or merge operation. For example, the hunk for the `LineCurve` mentioned above might include the lines

```
StartPoint: (1, 0, 0)
EndPoint: (9, 0, 0)
Layer: 6f9cb61e-f062-4751-aca1-5cc587ccd0ae
```

which would indicate that the newly created line runs horizontally for eight units and was placed on the layer identified by a UUID beginning with `6f9c`. Note that a `LineCurve` has many properties besides `StartPoint`, `EndPoint`, and `Layer`, but they are not included in the hunk because they still have their default values. A full listing of the properties supported by each component type is given in Appendix D.

The names and values of the non-default properties of deleted components are similarly enumerated in their respective hunks. Their inclusion makes the delta bidirectional so that a patch operation can be applied in reverse to transform a newer version of a model back into an older version.

For components that were modified, the list of key-value pairs enumerates the properties whose values have changed along with a representation of how

their values have changed. Most changes will be represented as substitutions of one value for another. For example, the hunk for the aforementioned modified layer might include the lines

```
Name: "Layer 01" -> "Lines"
Color: (200, 0, 0, 255) -> (0, 255, 255, 255)
```

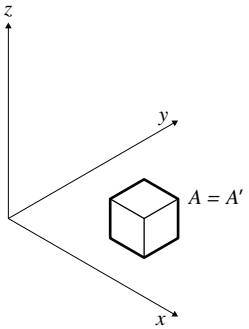
which would indicate that the layer was renamed from “Layer 01” to “Lines” and that its color was changed from red to cyan.

Changes to certain geometric properties will be expressed as 4×4 matrices, which are capable of representing a multitude of geometric transformations, including translation, rotation, scaling, and shearing, in a singular and consistent data structure (Figure 3.1). These matrices, which are common in 3D computer graphics, can be used to transform any type of geometry in an openNURBS model, and can be inverted, composed with other matrices, and decomposed into sets of simpler transformations. In the openNURBS delta format, these matrices will be expressed in row-major order, which lists the sixteen values in the matrix from left to right beginning with the top row and proceeding to to the bottom. The matrix shown in Figure 3.1b, for example, would be written as

```
transform(1 0 0 2; 0 1 0 3; 0 0 1 5; 0 0 0 1)
```

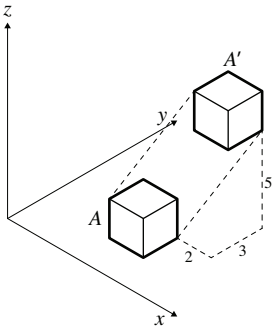
3.5 System Architecture

The goal of this thesis is to develop a suite of three command-line programs that implement diffing, patching, and three-way merging for



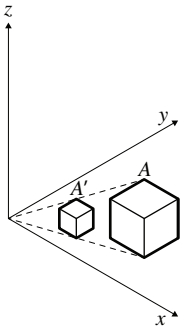
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) The identity matrix represents no change to the geometry.



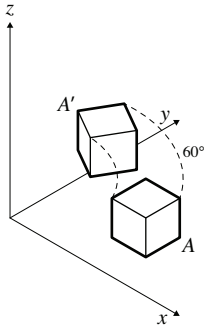
$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) A translation 2 units in the x direction, 3 units in the y direction, and 5 units in the z direction.



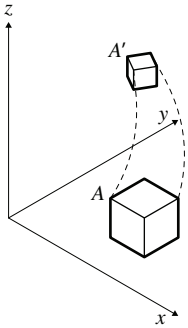
$$\begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c) Uniform scaling by one-half centered on the origin.



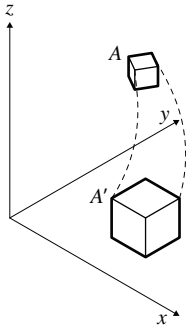
$$\begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} & 0 & 0 \\ \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(d) A 60 degree rotation about the z axis.



$$\begin{bmatrix} \frac{1}{4} & -\frac{\sqrt{3}}{4} & 0 & 2 \\ \frac{\sqrt{3}}{4} & \frac{1}{4} & 0 & 3 \\ 0 & 0 & \frac{1}{2} & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(e) Transformations **b**, **c**, and **d** combined into a single matrix.



$$\begin{bmatrix} 1 & \sqrt{3} & 0 & -2-3\sqrt{3} \\ -\sqrt{3} & 1 & 0 & -3+2\sqrt{3} \\ 0 & 0 & 2 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(f) The inverse of transformation **e**.

Figure 3.1. Example transformation matrices

openNURBS models and to demonstrate their integration into a Git repository so that common operations such as commits and merges can be performed on openNURBS models transparently and in a fully optimistic manner. The programs will be named `3dmdiff`, `3dmpatch`, and `3dmdiff3`, respectively, after the standard programs used to perform these operations on plain text files and the `.3dm` file extension conventionally used to identify openNURBS models.

A significant amount of code will be shared among these three programs. Key concepts such as components and properties remain the same regardless of whether one is diffing, patching, or merging; they are simply used in different ways depending on which operation is being performed. This situation lends itself to a *layered architecture*, a software design pattern in which the application's responsibilities are divided among multiple levels, each of which relies on the functionality of the one below it. Layered architectures are generally organized such that the lowest level defines fundamental concepts (using classes) in as generalized a manner as possible, and each level above it employs those concepts in an increasingly specialized fashion. This arrangement promotes code reuse by allowing new specializations to be constructed on top of existing layers as well as by permitting one implementation of the set of concepts expressed by a layer to be exchanged for another. A layered architecture can also advance the testability of an application, help developers more easily comprehend its code, and simplify dependency management (Buschmann et al. 1996; Fowler 2002).

The base of the layered architecture of the `3dmdiff` suite (Figure 3.2) will consist of an abstract model that defines the elemental concepts, such as

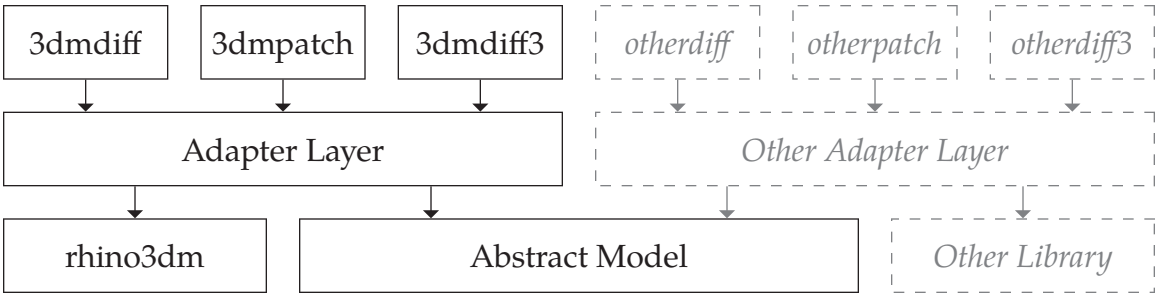


Figure 3.2. The layered architecture of the 3dmdiff suite

components, properties, values, and deltas, around which the programs are based. The fundamental algorithms involved in diffing, patching, and three-way merging will also be defined at this level of the application in a way that is agnostic of openNURBS or the details of the rhino3dm library. This separation will encourage a clearer expression of the ideas underpinning the application and will allow the software to be more easily modified to use a different openNURBS library, or even to work with a different file format entirely, at some point in the future.

The next layer will consist of a collection of adapters that connect the generalized classes of the abstract model to the particulars of the openNURBS format as implemented in the rhino3dm library. Specific component types and properties will be defined at this level, and peculiarities of the openNURBS format, such as components being addressed both by UUID and by index, will be dealt with.

The `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs themselves form the third and uppermost layer of the application. They will be responsible for collecting

user input, reading and writing files, and employing the functionality provided by the adapter layer to accomplish their respective tasks.

CHAPTER 4

IMPLEMENTATION

4.1 The Abstract Model

The abstract model defines the fundamental concepts that underlie the `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs in terms of a suite of classes and interfaces. In object-oriented programming, an *interface* describes a set of behaviors (in the form of methods) without specifying how those behaviors should be carried out. Classes that implement an interface are free to choose the most appropriate means of carrying out the actions it specifies. This ability of different classes to respond to the same method call, or message, with their own set of instructions is known as *polymorphism* and is one of the pillars of object-oriented programming. (Armstrong 2006).

Polymorphism can also be achieved through *abstract classes*, which provide implementations for some, but not all, of the methods they specify. Abstract classes are generally used to define common routines that can be included in multiple other classes through inheritance. However, unlike a normal parent class, an abstract class forces its subclasses to provide implementations of any methods for which it does not.

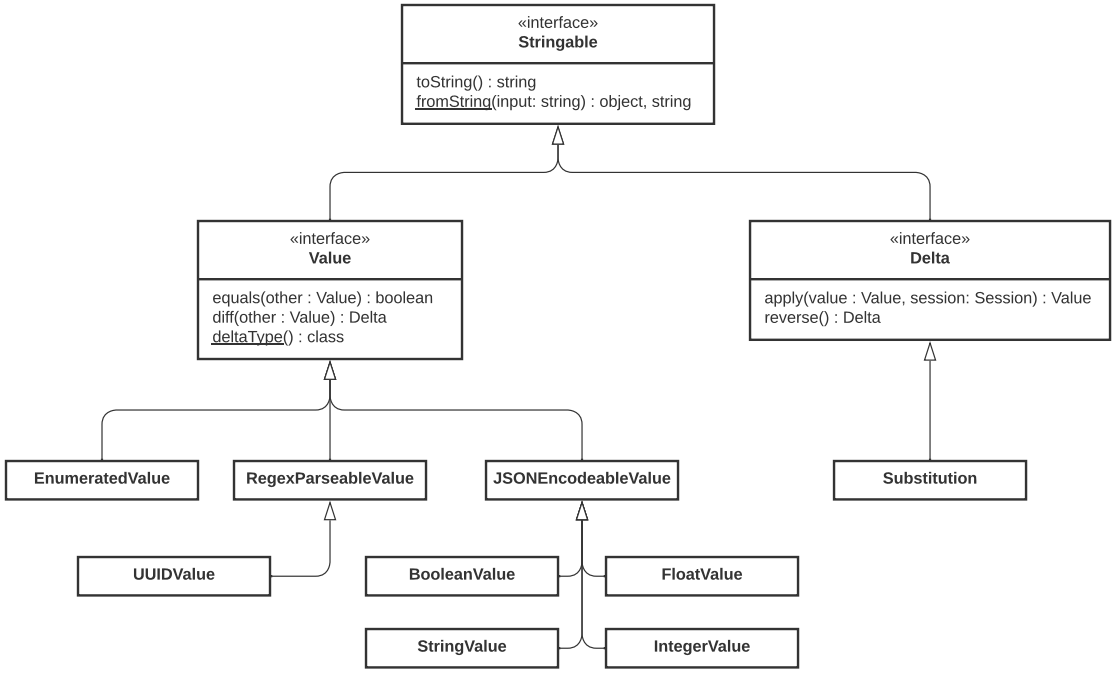


Figure 4.1. A UML diagram depicting the relationships between the `Stringable`, `Value`, and `Delta` interfaces and their concrete implementations in the abstract model.

The abstract model described here includes both interfaces and abstract classes. Because the Python programming language does not support interfaces, abstract classes are used in place of interfaces in the code for the `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs.

4.1.1 Stringables

In computer science, a *string* is a sequence of textual characters. Because `3dmdiff` outputs its deltas in a plain text format, and because `3dmpatch` must be able to read that plain text data, the ability to create, manipulate, and parse strings is a crucial part of their operation.

In the abstract model, a `Stringable` object is one that can be converted to and from a string. The `Stringable` interface is comprised of two methods:

`toString` () : *string*
: produces a textual representation of the object it is called on. In the Python code, this method is named `__str__`, which allows it to be automatically called when a string representation of the object is required.

`fromString` (`input` : *string*) : *object, string*
: is a class method, which means that it is invoked on a class rather than on any particular instance of that class. It accepts a string and attempts to parse a value from the beginning of that string. If successful, it returns an instance of the class it was called on as well as the unparsed remainder of the input string.

4.1.2 Values

A `Value` holds a piece of information retrieved from or that can be assigned to a component property and wraps it in a consistent interface that provides methods for common operations such as checking equality and computing deltas. This consistent interface allows other objects to operate on `Value` instances without needing to make special allowances for the varying types of their enclosed values.

A `Value` is a `Stringable` object, which means that possesses `toString` and `fromString` methods along with the following:

`equals` (`other` : *Value*) : *boolean*
: returns true if the `Value` passed to it is equal to the one it was called on.

`diff` (`other` : *Value*) : *Delta*
: returns an object that describes how the `Value` it was called on can be changed into the one that was passed to it. The returned object is an instance of the `Delta` class, which is described in Section 4.1.3.

`deltaType` () : *class*
is a class method that identifies which child class of `Delta` is returned by its class's `diff` method. It is used when parsing openNURBS deltas.

The abstract model defines a number of implementations of the `Value` interface. A `JSONEncodeableValue` is one that uses Python's built-in `json` package to convert values to and from their textual representations. `BooleanValue`, `FloatValue`, `IntegerValue`, and `StringValue` are subclasses of `JSONEncodeableValue` that are used for booleans, floating-point numbers, integers, and strings, respectively.

A `RegexParseableValue` is one that uses a regular expression to parse the textual representation of its value. `UUIDValue` is a subclass of `RegexParseableValue` meant for storing UUIDs.

Finally, the `EnumeratedValue` class serves as a base for value types that have only a limited number of acceptable values. It stores those acceptable values in a lookup table along with their textual representations for encoding and decoding.

4.1.3 Deltas

A `Delta` represents a change to the value of a component property. Like a `Value`, a `Delta` encapsulates the details of that change so that other objects can work with its instances without needing specific knowledge of the type of value that was changed or how the change takes place. A `Delta` is a `Stringable` object, which means that possesses `toString` and `fromString` methods along with the following:

apply (`value` : *Value*, `session` : *Session*) : *Value*
produces a new `Value` by executing the change described in the object it was invoked on with the value that was passed to it.

reverse () : *Delta*
returns a new `Delta` that performs the opposite change from the one it was called on. This method is used during reverse patch operations.

The abstract model defines one implementation of the `Delta` interface, named `Substitution`, which simply replaces an older value with a newer one.

4.1.4 Accesssors

An `Accessor` specifies the means by which values are retrieved from and assigned to component properties. It encapsulates knowledge about how to navigate specific parts of the rhino3dm API in a way that can be leveraged by the rest of the abstract model. The `Accessor` interface exposes two methods:

get (`component` : *object*) : *any*
returns the value of the given component's property.

set (`component` : *object*, `value` : *any*) : *void*
assigns the provided value to the given component's property.

The abstract model defines two main implementations of the `Accessor` interface. A `FunctionalAccessor` is constructed from two functions, one of which retrieves values from a component property and the other of which assigns values to a component property. A `PathAccessor` leverages Python's introspective capabilities, namely the `getattr` and `setattr` functions, to get and set properties based on their names in the rhino3dm API.

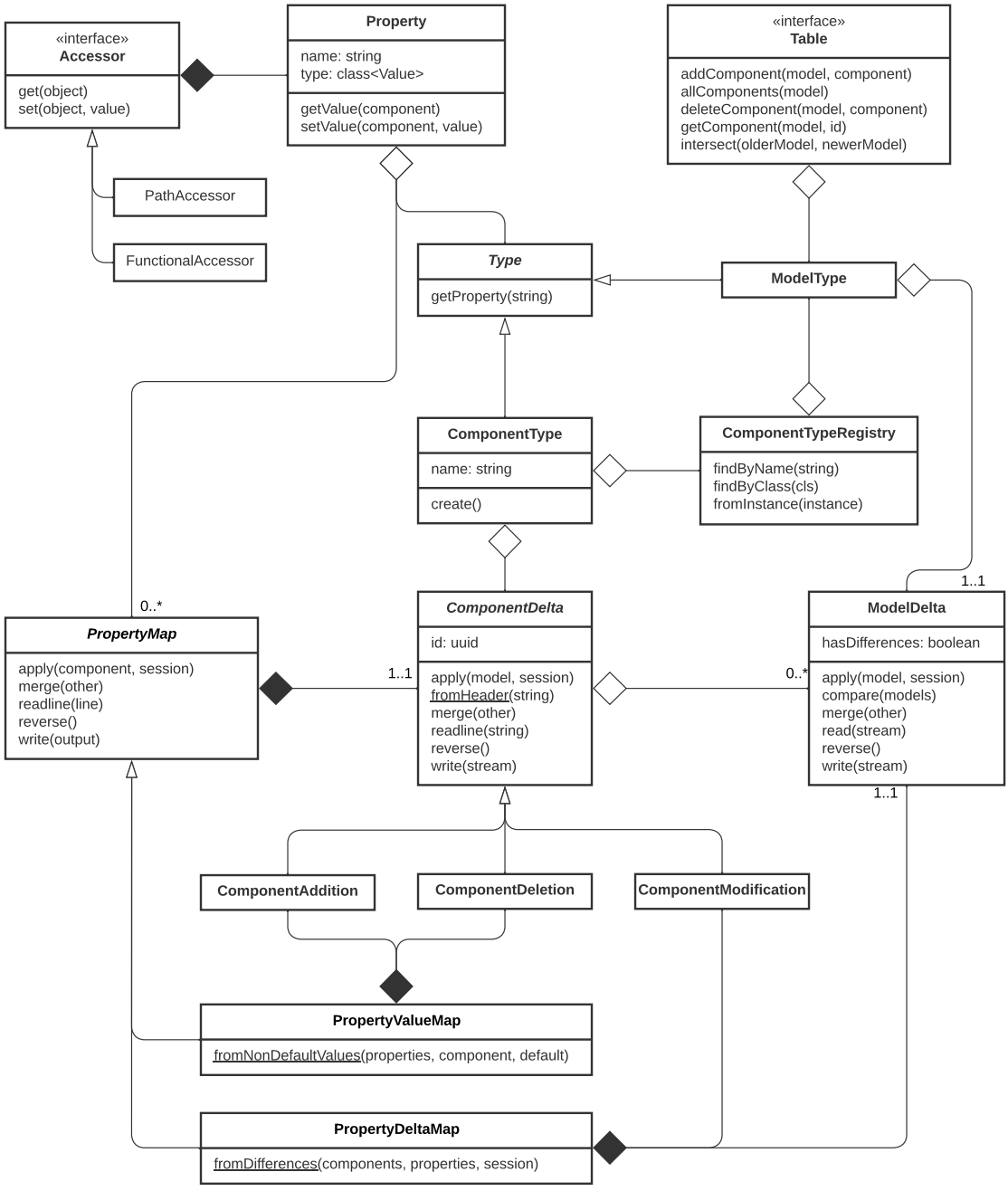


Figure 4.2. A UML diagram depicting part of the abstract model.

4.1.5 **Properties**

A `Property` represents a property of a component. It associates an `Accessor` instance and an implementation of `Value` with a textual label that is unique within the scope of the property's component. Each `Property` instance has the following attributes and methods:

`name` : *string*
the property's textual label.

`type` : *class*
the subclass of `Value` that is produced by the property's `getValue` method and expected by its `setValue` method.

`getValue` (`component` : *object*, `model` : *object*) : *void*
retrieves the value of the given component's property.

`setValue` (`component` : *object*, `value` : *Value*, `model` : *object*) : *void*
assigns the provided value to the given component's property.

Properties are hashable so that they can be used as keys in a Python dictionary, such as in the `PropertyMap` classes described in Section 4.1.6. Instances of `Property` are also considered equal to a string containing their name in order to ease lookups.

4.1.6 **Property Maps**

A `PropertyMap` is a mapping (or dictionary, in Python terms) that correlates one or more `Property` instances to an associated `Value` or `Delta`. A `PropertyMap` has the following methods:

`apply` (`component` : *Component*, `model` : *Model*, `session` : *Session*) : *void*
applies the values or deltas listed in the `PropertyMap` to the given component.

merge (`other` : *PropertyMap*, `session` : *Session*) : *void*
: returns a new `PropertyMap` that combines the properties listed in `other` with those listed in the one it was called on. The merge will not succeed if the two `PropertyMaps` have different values or deltas assigned to the same property.

readline (`line` : *string*, `componentType` : *ComponentType*) : *void*
: parses a property and value or delta from the given string and adds them to the `PropertyMap`.

reverse (`other` : *PropertyMap*) : *PropertyMap*
: returns a new `PropertyMap` that has the opposite meaning of the one it was called on.

write (`output` : *Stream*) : *void*
: writes a textual representation of the `PropertyMap` to the given output stream.

The abstract model defines two concrete implementations of `PropertyMap`. A `PropertyValueMap` maps properties to `Value` instances and is used to describe components that have been added to or deleted from a model. A `PropertyDeltaMap` maps properties to `Delta` instances and is used to describe components that have been modified, as well as certain modifications to the model itself.

4.1.7 Tables

A `Table` specifies how to retrieve components from, add them to, and delete them from a model. Its interface specifies the following methods:

allComponents (`model` : *Model*) : *Iterable<Component>*
: retrieves the complete list of components in the table.

getComponent (`model` : *Model*, `id` : *uuid*) : *Component*
: retrieves the component with the given ID.

addComponent (`component` : *Component*) : *void*
: adds a component to the table.

deleteComponent (`component` : *Component*) : *void*
removes a component from the table.

intersect (`older` : *Model*, `newer` : *Model*) : *Intersection*
determines which components have been removed from the given `older` model, which ones have been added to the given `newer` model, and which ones appear in both.

4.1.8 Types

A `Type` enumerates the properties supported by a kind of object.

properties : *Iterable<Property>*
the properties supported by objects of this type.

getProperty (`name` : *string*) : *Property*
returns the property with the given name. Property names are case-insensitive, so Color, `color`, `COLOR`, and `CoLOr` all resolve to the same property.

The abstract model defines two concrete classes that inherit from `Type`. A `ComponentType` describes a type of component by associating a table and a list of properties with a unique name. It exposes the following properties and methods in addition to the ones described above:

name : *string*
the name of the type.

create () : *object*
creates a new component of this type.

The set of component types supported by a model format are gathered in a `ComponentTypeRegistry` which exposes the following methods:

findByName (`name` : *string*) : *Property*
returns the component type with the given name. Like properties, component type names are case-insensitive.

findByClass (`cls` : *class*) : *Property*
returns the component type that corresponds to the given class.

fromInstance (`instance` : *object*) : *Property*
returns the component type that corresponds to the class of the given object.

A `ModelType` also inherits from `Type`, and lists the tables, component types, and properties associated with a model format. It exposes the following properties in addition to those belonging to its parent class:

componentTypes : *ComponentTypeRegistry*
the set of component types supported by the model format.

tables : *Iterable<Table>*
the set of tables that store components in this model format.

4.1.9 Component Deltas

A `ComponentDelta` describes changes to a single model component, and corresponds to a single hunk in the output of a diff operation. It provides the following properties and methods:

id : *uuid*
the UUID of the component.

type : *ComponentType*
the type of the component.

properties : *PropertyMap*
the non-default properties of the component if was added or deleted, or if the component was modified, its altered properties.

apply (`model` : *Model*, `session` : *Session*) : *void*
applies the changes described in the `ComponentDelta` to the given model.

fromHeader (`header` : *string*, `types` : *ComponentTypeRegistry*) : *ComponentDelta*
creates an empty `ComponentDelta` from a header line. The class of the returned object depends on the first character of the header.

readline (`line` : *string*) : *void*
parses a property and value or delta from the given string and adds them to the object's `PropertyMap`.

reverse (`other` : *ComponentDelta*) : *ComponentDelta*
returns a new `ComponentDelta` that has the opposite meaning of the one it was called on.

merge (`other` : *ComponentDelta*, `session` : *Session*) : *void*
returns a new `ComponentDelta` that includes both the changes listed in `other` as well as those listed in the one it was called on. The merge will not succeed if the two `ComponentDelta`s have different values or deltas assigned to the same property.

write (`output` : *Stream*) : *void*
writes a textual representation of the `ComponentDelta` to the given output stream.

The abstract model defines three concrete implementations of `ComponentDelta`. The `ComponentAddition` and `ComponentDeletion` classes represent components that have been added to and deleted from the model, respectively. Each contains a reference to a `PropertyValueMap` that describes how the component that was added or deleted differs from the default state of a component of its type. The `ComponentModification` class represents a component that has been modified between older and newer versions of a model. It contains a reference to a `PropertyDeltaMap` that describes how the component was changed.

4.1.10 Model Deltas

A ModelDelta describes the result of diffing two models. It contains a collection of `ComponentDelta` instances as well as a list of properties that have changed on the model itself.

components : *Iterable<ComponentDelta>*
the list of model components that were added, removed, or modified.

hasDifferences : *boolean*
true when the ModelDelta describes at least one change.

properties : *PropertyDeltaMap*
the properties of the model that were changed.

apply (`model` : *Model*, `session` : *Session*) : *void*
applies the changes described in the ModelDelta to the given model.

compare (`models` : *tuple<Model>*, `session` : *Session*) : *void*
looks for and records differences between the given models.

merge (`other` : *ModelDelta*) : *ModelDelta*
returns a new `ModelDelta` that includes both the changes listed in `other` as well as those listed in the one it was called on. The merge will not succeed if the two `ModelDelta`s have different deltas assigned to the same property or if any of the component deltas in the models are incompatible.

read (`input` : *Stream*) : *void*
reads a delta from the given input stream.

reverse () : *ModelDelta*
returns a new `ModelDelta` that has the opposite meaning of the one it was called on.

write (`output` : *Stream*) : *void*
writes a textual representation of the `ModelDelta` to the given output stream.

4.1.11 Sessions

A Session encapsulates procedures for communicating abnormal or unexpected situations to the user in a way that does not require the abstract model to be tied to a specific user interface paradigm. Its interface specifies the following methods:

warn (`message` : *string*) : *void*
displays a warning message.

fatal (`message` : *string*) : *void*
displays an error message and exits the program.

setContext (`componentType` : *string*, `componentID` : *string*, `property` : *string*) : *void*
sets contextual information about the component and property currently

being processed that can be displayed in conjunction with an error or warning message.

4.2 The Adapter Layer

Between the abstract model and the `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs lies an adapter layer that connects the fundamental concepts and algorithms laid out in the abstract model with the openNURBS component and geometry classes provided by the rhino3dm library. Much of this layer consists of relatively straightforward definitions of component types, value types, and properties in terms of the classes and interfaces described in section 4.1. For example, the `Color` property of a geometric object is defined by the expression

```
Property("Color", value_types.Color, "Attributes.ObjectColor")
```

where `Property` is the class described in section 4.1.5, `"Color"` is the name of the property that will appear in deltas, `values_types.Color` is an implementation of the `Value` interface (Section 4.1.2) that stores the red, green, blue, and alpha values that comprise a color. The string `"Attributes.ObjectColor"` is converted by the `Property` constructor into an instance of `PathAccessor` (Section 4.1.4) that links the `Property` instance to the `ObjectColor` property of a component's `Attributes` object.

The adapter layer defines a number of `Value` implementations in addition to `Color`, including `Point3d` for storing points in three-dimensional space, `Vector3d` for storing three-dimensional vectors, and `Interval` for storing ranges

of numbers. It also defines an implementation of `Delta` called `Transformation` to store and apply the 4×4 matrices described in Section 3.4.

The adapter layer also attends to a number of non-trivial challenges involved in marrying the abstract model to the rhino3dm library. One such challenge involves the dual identification scheme present in openNURBS models in which components are identified both by a UUID and by an index. A component's UUID is stable, but its index may change if other components of the same type are added to or removed from the model. The `3dmdiff` and `3dmdiff3` programs therefore rely on component UUIDs alone in order to prevent changing indexes from increasing the complexity deltas more than necessary. However, certain properties that reference other components, such as the linetype of a layer or the material of a geometric object, do so by storing the referenced component's index. The adapter layer works around these situations by providing a special implementation of `Accessor`, called an `IndexReferenceAccessor`, that translates between indexes and UUIDs. The `Linetype` property of a layer is therefore defined by the expression

```
Property("Linetype", UUIDValue, IndexReferenceAccessor("
   LinetypeIndex", tables.LINETYPE_TABLE))
```

where `"LinetypeIndex"` is the name of the attribute on the rhino3dm `Layer` object that contains the index of the layer's linetype and `tables.LINETYPE_TABLE` is the table in which that index should be looked up to convert it to a UUID.

Another difficulty the adapter layer contends with involves differences in how Python and C++ (the language in which openNURBS is written) handle the

movement of data into and out of functions. In C++, data can be passed *by value*, which means it is copied from one scope (such as the code that calls a function) to another (the code inside the function). It can also be passed *by pointer* or *by reference*, in which case the location of the data in memory is shared so that changes to that data that occur in one scope are automatically propagated to the other. Python, on the other hand, uses a simpler system in which all values are passed by reference. The `Line` property of the `LineCurve` class is one of a few places in the rhino3dm API where these two systems collide, as seen in the following example:

```
1 from rhino3d import LineCurve, Point3d
2
3 p0 = Point3d(3, 0, 0)
4 p1 = Point3d(0, 4, 0)
5 curve = LineCurve(p0, p1)
6 print(curve.Line.From) # outputs "3.0,0.0,0.0"
7
8 curve.Line.From = Point3d(10, 10, 0)
9 print(curve.Line.From) # still outputs "3.0,0.0,0.0"
```

Under Python's standard pass-by-reference semantics, one would expect the assignment on line 8 to have changed the start point of the `LineCurve` so that `print` statement on line 9 would output `10.0,10.0,0.0`. However, the underlying C++ function that serves as the getter for `LineCurve.Line` returns its value *by reference*. Line 8 therefore assigns a new start point to a *copy* of the line stored within the `LineCurve` component rather than the original line. After line 8 has finished executing, the copy is destroyed along with its new start point.

To change the start point of the line contained within the `LineCurve` component, one must instead store a copy of that line in a variable, change the start point of the copy, and then assign the copy back to the `LineCurve`'s `Line` attribute as shown below.

```
10 line = curve.Line
11 line.From = Point3d(10, 10, 0)
12 curve.Line = line
```

The adapter layer provides an implementation of `Accessor`, called `ValueObjectAccessor`, which performs this action automatically. The `StartPoint` property of a `LineCurve` is therefore defined using the expression

```
Property("StartPoint", value_types.Point3d, ValueObjectAccessor
   ("Geometry.Line", "From"))
```

where the string `"Geometry.Line"` specifies the object that must be copied and reassigned and `"From"` indicates the attribute of that property that contains the property's value.

4.3 Command Line Interface

One of the major contributions of this thesis is the development of three programs, named `3dmdiff`, `3dmpatch`, and `3dmdiff3`, that implement diffing, patching, and three-way merging for openNURBS models. The command-line interfaces of these programs are modeled after those provided by the GNU

Diffutils package[1], which furnishes industry-standard open-source tools for comparing and merging plain text files. It is hoped that the similarity of `3dmdiff`, `3dmpatch`, and `3dmdiff3` to the widely-used `diff`, `patch`, and `diff3` programs will foster a sense of familiarity among prospective users and allow the openNURBS-specific tools described in this thesis to more easily integrate into existing workflows.

Like their plain text counterparts, `3dmdiff` and `3dmpatch` make extensive use of the *standard streams*, a set of three communications channels that provide a way for computer programs to exchange information with their environments. Standard input, also known as *stdin*, is used to pass information into a program, and in a command line environment, is typically connected to the keyboard. Standard output, abbreviated as *stdout*, is used to convey the results of a program's execution to the outside world. Data that is written to stdout is typically printed in the command line console. Finally, the standard error (*stderr*) stream is used for error messages and debugging information that is not part of a program's normal output.

The standard streams can be *redirected* so that they connect to a file or another program instead of the keyboard or screen. One form of redirection, called a pipe, has already been demonstrated in Section 3.3 with the command

```
ls | wc -l
```

1. https://www.gnu.org/software/diffutils/

In that example, the | character plugs the standard output stream of `ls` (which contains a listing of the files and folders in the current directory) into the standard input stream of `wc` (which counts the number of lines in a file) to count the number of files and folders in the current directory.

The standard output stream can be redirected to a file using the > operator. Rather than being piped directly into `wc` as in the previous example, the output of `ls` could have been written to a file called `file_list.txt` using the command

```
ls > file_list.txt
```

Similarly, the < operator connects a file to a program's standard input stream. The command

```
wc -l < file_list.txt
```

allows `wc` to read the contents `file_list.txt` as if they had been typed by the user. Several other forms of stream redirection exist, but the basic patterns shown here are sufficient to understand the operation of the `3dmdiff` and `3dmpatch` programs.

4.3.1 **3dmdiff**

The `3dmdiff` program finds differences between two openNURBS models. and prints an account of those differences to standard output using the delta format described in Section 3.4. Like GNU `diff`, it accepts two file system paths as arguments. However, both arguments to `3dmdiff` must reference normal files as the program does not support diffing of directory structures. It is invoked as:

```
3dmdiff [options] fromfile tofile
```

where `fromfile` is a path to the openNURBS model that will serve as the origin of the resulting delta and `tofile` is a path to the openNURBS model that will serve as its destination. The following options are supported:

`-q, --brief` Causes `3dmdiff` to report only whether the models differ, and not what those differences are

`-s, --report-identical-files` Causes `3dmdiff` to print a brief statement when the two models given to it are identical. Normally, it would output nothing in that case.

`--label=LABEL` Instructs `3dmdiff` to use `LABEL` in place of `fromfile` in the delta's preamble. This option may be repeated to make a similar substitution for `tofile`.

`--git` Tells `3dmdiff` to expect a different set of arguments for easier integration with Git. See Section 4.4 for more information.

`-v, --version` Instructs 3dmdiff to print its version information and then exit.

`-h, --help` Instructs 3dmdiff to print a description of its usage and then exit.

Users of `3dmdiff` will frequently want to capture the outputted delta in a file to use later with `3dmpatch`. This can be accomplished by redirecting standard output to a file as in the command

```
3dmdiff version01.3dm version02.3dm > my_delta.txt
```

which finds the differences between `version01.3dm` and `version02.3dm` and saves the result in a file called `my_delta.txt`.

4.3.2 **3dmpatch**

The `3dmpatch` program applies a delta to an openNURBS model. Typically, `3dmpatch` reads the delta from standard input as follows:

```
3dmpatch [options] < patchfile
```

When invoked in this manner, `3dmpatch` attempts to apply the changes in the delta to the first file listed in the delta's preamble. Alternatively, the model to which the delta should be applied can be specified as an argument:

```
3dmpatch [options] originalfile < patchfile
```

The delta may also be specified as an argument instead of being read from standard input:

```
3dmpatch [options] originalfile patchfile
```

All three of these invocation patterns are consistent with the way GNU `patch` operates. Additionally, the following options are supported:

`-o PATH, --output=PATH` Instructs `3dmpatch` to write the result of the patch operation to `PATH`. If this option is not specified, the file is patched in place.

`-R, --reverse` Causes `3dmpatch` to apply the delta in reverse.

`-v, --version` Instructs 3dmpatch to print its version information and then exit.

`-h, --help` Instructs 3dmpatch to print a description of its usage and then exit.

4.3.3 3dmdiff3

The `3dmdiff3` program performs a three-way diff between two openNURBS models that share a common ancestor and optionally applies the resulting delta to the common ancestor to produce a merged model. Unlike GNU `diff3`, `3dmdiff3` produces deltas that are meant to be applied to the common ancestor rather than the first argument. It is invoked as

```
3dmdiff3 [options] myfile oldfile yourfile
```

where `myfile` and `yourfile` are the paths to the openNURBS models to be merged and `oldfile` is the path to their common ancestor. The following options are supported:

`-m, --merge` Instructs `3dmdiff3` to produce a merged model. Without this option, `3dmdiff3` outputs a diff

`-o PATH, --output=PATH` Instructs `3dmdiff3` to write the result of a merge operation to `PATH`. If this option is not specified, `oldfile` is patched in place.

`-v, --version` Instructs 3dmdiff3 to print its version information and then exit.

`-h, --help` Instructs 3dmdiff3 to print a description of its usage and then exit.

4.4 Git Integration

Integrating the `3dmdiff` and `3dmdiff3` programs into a Git repository involves defining custom diff and merge drivers and then associating those drivers

with the `.3dm` file extension (*Gitattributes Documentation*, n.d.). Git does not presently support integration with third-party patch programs such as `3dmpatch`.

The custom diff and merge drivers may be defined in one of several configuration files depending on their desired scope. If the drivers are needed for only one repository, they may be defined in that repository's `.git/config` file. If they are required in all of a user's repositories, they may be defined in a file called `.gitconfig` in that user's home folder. Finally, if the drivers are required in all repositories belonging to all users of a computer, they may be defined in `$(prefix)/etc/gitconfig`, where `$(prefix)` is the directory into which Git was installed (*Git-Config Documentation*, n.d.).

Git configuration files follow a syntax similar to that of INI files, and custom diff and merge drivers are defined as sections within that syntactical framework. The section for a diff driver begins with a header that contains the word *diff* followed by an arbitrary quoted name. It contains a single key, command, that specifies the program to be used for the diff operation. This program receives seven command-line arguments:

1. The path of the file being diffed within the repository.
2. A path from which the contents of the older file can be read.
3. The 40-hexdigit hash of the older file.
4. The octal representation of the older file's mode.
5. A path from which the contents of the newer file can be read.
6. The 40-hexdigit hash of the newer file.
7. The octal representation of the newer file's mode.

The `--git` option to `3dmdiff` tells the program to expect those seven arguments in place of the two arguments described in Section 4.3.1. A diff driver for openNURBS models may therefore be defined as:

```
[diff "opennurbs-diff-driver"]
command = 3dmdiff --git
```

The section for a merge driver begins with a header that contains the word *merge* followed by an arbitrary quoted name. It contains up two keys: `name` defines a human-readable label for the merge driver and `driver` specifies the command to be used for the merge operation. A third key, `recursive`, is also supported but is not required for merging openNURBS models.

Git constructs the merge command by replacing special tokens in the text of the `driver` setting with information relevant to the merge operation. The token `%A` is replaced with a path from which the merge program can read the current branch's version of the file being merged, and `%B` is replaced with a path from which it can read the other branch's version. The `%O` token is replaced with a path to the common ancestor and `%P` is replaced with the path to which the merged file should be saved. Therefore, a merge driver for openNURBS models may be defined as:

```
[merge "opennurbs-merge-driver"]
name = Merge driver for openNURBS models
driver = 3dmdiff -m -o %P %A %O %B
```

The final step of integrating `3dmdiff` and `3dmdiff3` with Git is to associate the diff and merge drivers defined above with the `.3dm` file extension by adding a

line similar to the one below to one of Git's attribute files. Note that the names `opennurbs-diff-driver` and `opennurbs-merge-driver` correspond to the names assigned to the diff and merge drivers in their respective headers.

```
*.3dm diff=opennurbs-diff-driver merge=opennurbs-merge-driver
```

Like Git's configuration files, there are several attributes files that may be used depending on the desired scope of the directives shown above. If the openNURBS diff and merge drivers are needed for only one repository, the line may be added to a file named `.gitattributes` in the root of that repository or in its `.git/info/attributes` file. If they are required in all repositories on a machine, the line may added to `$(prefix)/etc/gitattributes`, where `$(prefix)` is the directory into which Git was installed.

CHAPTER 5

DEMONSTRATION

5.1 Diffing and Patching

The operation of the `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs, along with their ability to work within the context of a Git repository, was demonstrated during the defense of this thesis on December 8, 2022. This demonstration began with an openNURBS model containing the tracing of Le Corbusier's Maison Dom-ino that was used to produce several illustrations in Chapter 2 (Figure 5.1). A copy of that file was created and subsequently modified so that two of the columns on the upper level were joined together to form a wall (Figure 5.2). The `3dmdiff` utility was then used to reveal the differences between the two files with the command

```
3dmdiff domino.3dm domino_copy.3dm
```

which produced the following output:

```
--- domino.3dm 2022-12-08 11:29:15.385805 -0500
+++ domino_copy.3dm 2022-12-08 11:40:49.943148 -0500
@@ ~LineCurve 18ff88dd-01a4-47ff-96d6-0eaf02ef9484 @@
    Domain: [8.272019639391859, 8.606367521859378] ->
        ↪ [-2.6645352591003757e-15, 8.606367521859378]
```

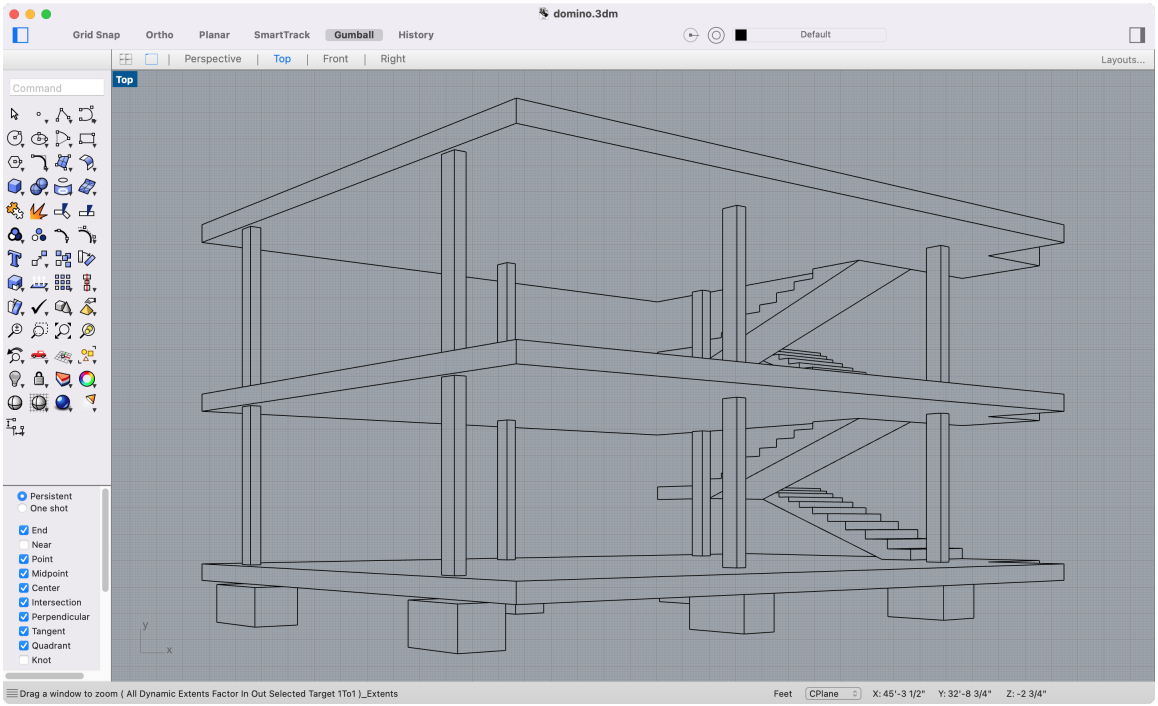


Figure 5.1. The original tracing of Le Corbusier's Maison Dom-ino.

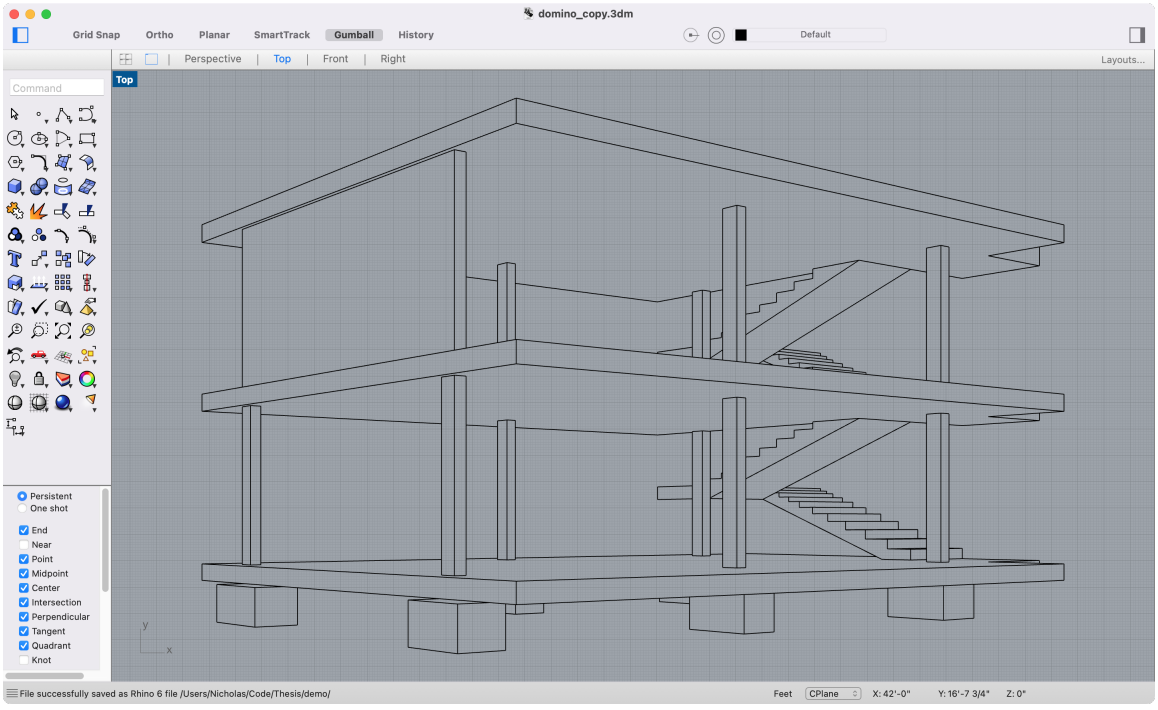


Figure 5.2. The modified copy of the Dom-ino tracing.

```
5       Geometry: transform(25.74075698852539, 0.0, 0.0,
           ↪ -229.1952667236328, 0.0, 25.74075698852539, 0.0,
           ↪ -632.5133056640625, 0.0, 0.0, 25.74075698852539, 0.0,
           ↪  0.0, 0.0, 0.0, 1.0)
6       StartPoint: (17.320573165521864, 28.592050881888326, 0.0) ->
           ↪  (17.32057316552188, 28.59205088188827, 0.0)
7       EndPoint: (9.263874233629338, 25.565639766035588, 0.0) ->
           ↪ (9.26387423362934, 25.56563976603544, 0.0)
8   @@ -LineCurve 5ebba82c-b909-4313-8247-6fdc5c634031 @@
9       Domain: [2.2551487856584056, 9.17534041976403]
10      StartPoint: (9.966767623614944, 24.781772236940483, 0.0)
11      EndPoint: (16.828909351701345, 23.887309741135965, 0.0)
12  @@ -LineCurve b07d85f3-efc1-4c07-be07-06211237b92e @@
13      Domain: [0.0, 0.39375629476038715]
14      StartPoint: (9.576868097848447, 25.68321249872336, 0.0)
15      EndPoint: (9.966767623614944, 25.628236326880554, 0.0)
16  @@ -LineCurve c4427803-1bc0-4067-9240-464e2f3525b7 @@
17      Domain: [2.7712785312946515, 10.277364432022416]
18      StartPoint: (16.82890935170135, 28.407362730576203, 0.0)
19      EndPoint: (16.82890935170135, 20.90127682984844, 0.0)
20  @@ -LineCurve 3b960de7-8cd5-4e90-a263-2be74deb736e @@
21      Domain: [0.0, 5.920871468485612]
22      StartPoint: (9.966767623614944, 25.628236326880554, 0.0)
23      EndPoint: (9.966767623614944, 19.707364858394943, 0.0)
24  @@ -LineCurve 3c8bf58c-820a-4989-854a-656862c9f028 @@
25      Domain: [4.7333498517256665, 10.777034288483303]
26      StartPoint: (9.576868097848447, 25.68321249872336, 0.0)
27      EndPoint: (9.576868097848447, 19.639528061965724, 0.0)
28  @@ -LineCurve 6d03ffd7-30e4-437d-9003-a608867ba401 @@
29      Domain: [0.0, 0.5252075961518584]
30      StartPoint: (17.32057316552188, 28.59205088188826, 0.0)
31      EndPoint: (16.82890935170135, 28.407362730576203, 0.0)
```

The hunk beginning on the third line of the output describes the line that was extended to form the top of the wall. Its transformation matrix indicates that the line in Figure 5.2 is roughly 25.74 times longer than its counterpart in Figure 5.1. The `StartPoint` and `EndPoint` properties on lines 6 and 7 were not expected to be included in the delta since the transformation matrix fully describes the

changes that were made to the line's geometry. Their appearance is due to floating-point rounding errors, which will be discussed in Section 6.2.

The other hunks in the delta correspond to lines that were removed from the model. Their domains and start and end points are listed in the delta so that the lines can be recreated in the event of a reverse patch.

The demonstration continued by re-running the `3dmdiff` command, this time saving the output to a file:

```
3dmdiff domino.3dm domino_copy.3dm > domino_delta
```

Next, the modified copy of the Dom-ino tracing was deleted. It was then re-created by applying the saved delta to the original file using `3dmpatch`:

```
3dmpatch -o domino_patched.3dm domino.3dm domino_delta
```

It would also have been possible to delete and re-create the original file using a reverse patch:

```
3dmpatch -R -o domino_revpatched.3dm domino_copy.3dm domino_delta
```

However, this functionality was not demonstrated during the defense.

5.2 Merging via the Command Line

After diffing and patching had been demonstrated, another copy of file containing the Maison Dom-ino tracing was made. In this copy, the color of the lines that formed one of the columns was changed (Figure 5.3). A three-way diff

between the two copies and the original file was then performed using the command

```
3dmdiff3 domino_copy.3dm domino.3dm domino_copy2.3dm
```

which produced the following delta combined the contents of the diff from Section 5.1 with the changes (lines 8–22 in the output below) necessary to describe the recolored column seen in Figure 5.3:

```
1  --- domino.3dm 2022-12-08 11:29:15.385805 -0500
2  +++ domino_copy.3dm 2022-12-08 11:40:49.943148 -0500
3  @@ ~LineCurve 18ff88dd-01a4-47ff-96d6-0eaf02ef9484 @@
4      Domain: [8.272019639391859, 8.606367521859378] ->
           ↪ [-2.6645352591003757e-15, 8.606367521859378]
5      Geometry: transform(25.74075698852539, 0.0, 0.0,
           ↪ -229.1952667236328, 0.0, 25.74075698852539, 0.0,
           ↪ -632.5133056640625, 0.0, 0.0, 25.74075698852539, 0.0,
           ↪  0.0, 0.0, 0.0, 1.0)
6      StartPoint: (17.320573165521864, 28.592050881888326, 0.0) ->
           ↪  (17.32057316552188, 28.59205088188827, 0.0)
7      EndPoint: (9.263874233629338, 25.565639766035588, 0.0) ->
           ↪ (9.26387423362934, 25.56563976603544, 0.0)
8  @@ ~LineCurve 18f77ee6-89ec-4ac1-a11f-7281c559f9c5 @@
9      Color: (0, 0, 0, 255) -> (255, 0, 0, 255)
10     ColorSource: layer -> object
11 @@ ~LineCurve f097dad3-3fd6-4cdd-8209-4ac81a89b923 @@
12     Color: (0, 0, 0, 255) -> (255, 0, 0, 255)
13     ColorSource: layer -> object
14 @@ ~LineCurve b67d7103-9ebc-4ab5-8fae-d7909747dfd8 @@
15     Color: (0, 0, 0, 255) -> (255, 0, 0, 255)
16     ColorSource: layer -> object
17 @@ ~LineCurve 22fb014a-85bd-4a56-a0cd-07321aadd52e @@
18     Color: (0, 0, 0, 255) -> (255, 0, 0, 255)
19     ColorSource: layer -> object
20 @@ ~LineCurve 4fdbb6d3-2513-45dc-94a9-c9875077de85 @@
21     Color: (0, 0, 0, 255) -> (255, 0, 0, 255)
22     ColorSource: layer -> object
23 @@ -LineCurve 5ebba82c-b909-4313-8247-6fdc5c634031 @@
24     Domain: [2.2551487856584056, 9.17534041976403]
25     StartPoint: (9.966767623614944, 24.781772236940483, 0.0)
26     EndPoint: (16.828909351701345, 23.887309741135965, 0.0)
```

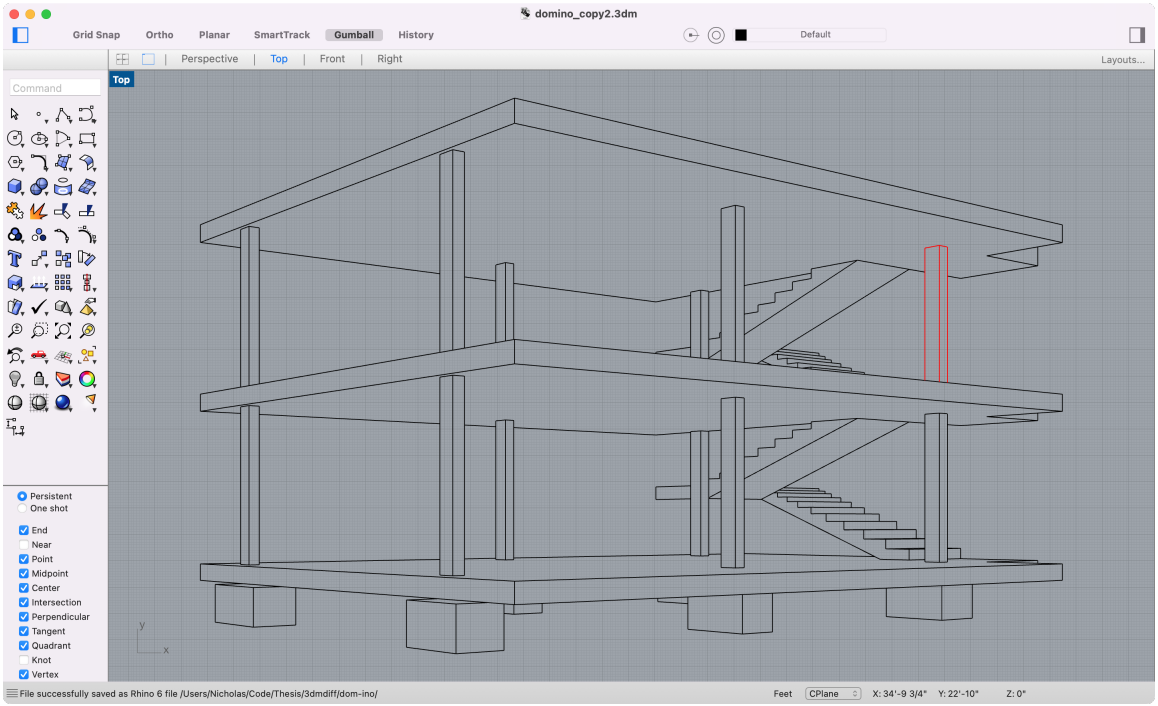


Figure 5.3. The second modified copy of the Dom-ino tracing.

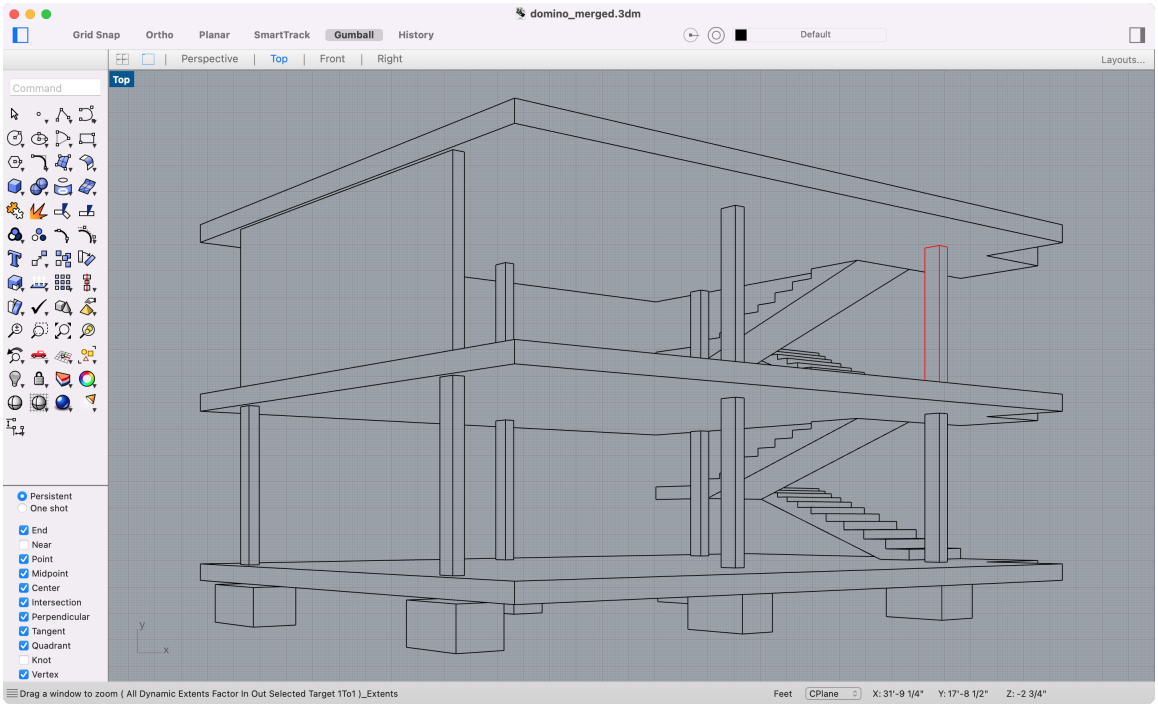


Figure 5.4. The merged model containing the changes from Figures 5.2 and 5.3.

```
@@ -LineCurve b07d85f3-efc1-4c07-be07-06211237b92e @@
    Domain: [0.0, 0.39375629476038715]
    StartPoint: (9.576868097848447, 25.68321249872336, 0.0)
    EndPoint: (9.966767623614944, 25.628236326880554, 0.0)
@@ -LineCurve c4427803-1bc0-4067-9240-464e2f3525b7 @@
    Domain: [2.7712785312946515, 10.277364432022416]
    StartPoint: (16.82890935170135, 28.407362730576203, 0.0)
    EndPoint: (16.82890935170135, 20.90127682984844, 0.0)
@@ -LineCurve 3b960de7-8cd5-4e90-a263-2be74deb736e @@
    Domain: [0.0, 5.920871468485612]
    StartPoint: (9.966767623614944, 25.628236326880554, 0.0)
    EndPoint: (9.966767623614944, 19.707364858394943, 0.0)
@@ -LineCurve 3c8bf58c-820a-4989-854a-656862c9f028 @@
    Domain: [4.7333498517256665, 10.777034288483303]
    StartPoint: (9.576868097848447, 25.68321249872336, 0.0)
    EndPoint: (9.576868097848447, 19.639528061965724, 0.0)
@@ -LineCurve 6d03ffd7-30e4-437d-9003-a608867ba401 @@
    Domain: [0.0, 0.5252075961518584]
    StartPoint: (17.32057316552188, 28.59205088188826, 0.0)
    EndPoint: (16.82890935170135, 28.407362730576203, 0.0)
```

Next, a merged openNURBS model (Figure 5.4) was produced by re-running the `3dmdiff` command with the `-m` option:

```
3dmdiff3 -m -o delta_merged.3dm domino_copy.3dm domino.3dm
   ↪ domino_copy2.3dm
```

5.3 Merging via Sourcetree

Most computer users, and perhaps especially those accustomed to working visually like architects, are unlikely to be willing to contend with the tedium of a command line interface. Therefore, the final portion of the December 8 demonstration illustrated not only the integration of the openNURBS diff utilities with a Git repository but also how branching and merging of openNURBS models

can be accomplished through a graphical user interface such as Atlassian's Sourcetree[1].

First, the original tracing of the Maison Dom-ino was copied into a repository that had already been set up according to the procedures described in Section 4.4. That file was then committed to the repository (Figure 5.5). Next, a new branch named `alternate` was created (Figure 5.6) and the version of the model on that branch was modified in the same way as Figure 5.2 (Figure 5.7). The repository was then switched back to the `master` branch, which restored the model to its original state. The model was again modified to resemble Figure 5.3 and those changes were committed directly to the `master` branch (Figure 5.8). Finally, the `alternate` branch was merged into the `master` branch to produce a model similar to Figure 5.4 (Figure 5.9).

1. https://www.sourcetreeapp.com/

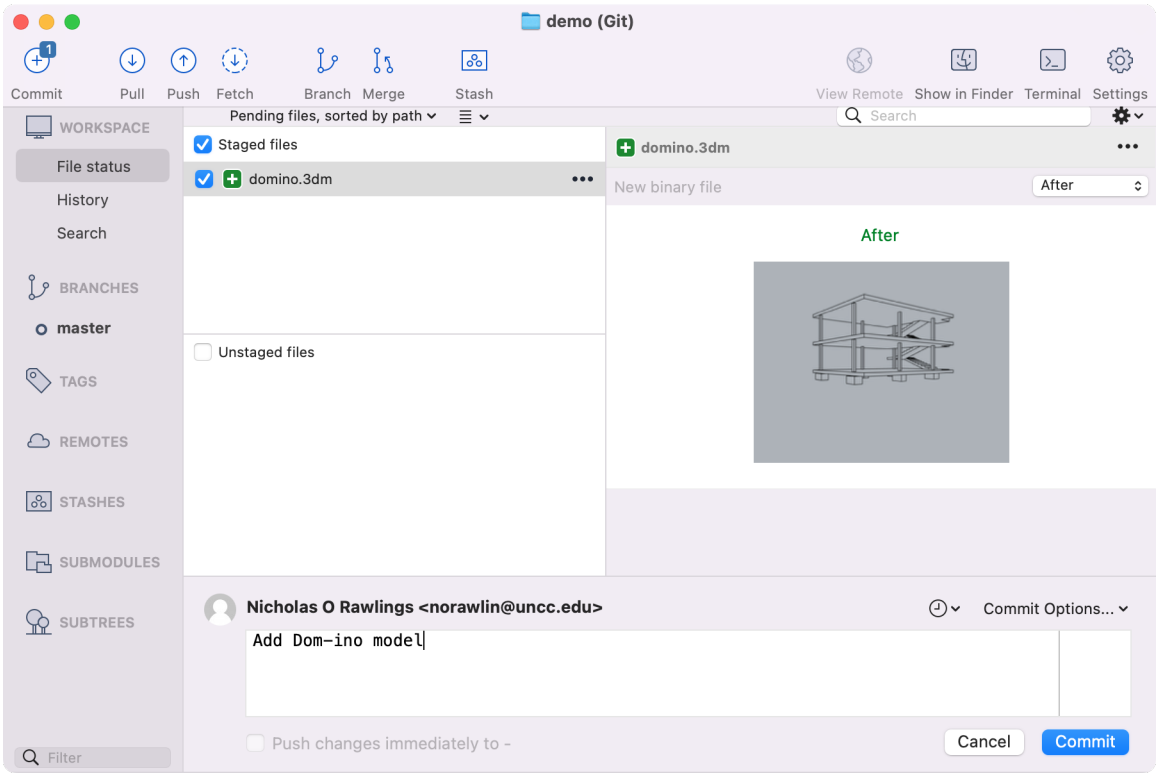


Figure 5.5. Committing the Maison Dom-ino model to a Git repository in Sourcetree.

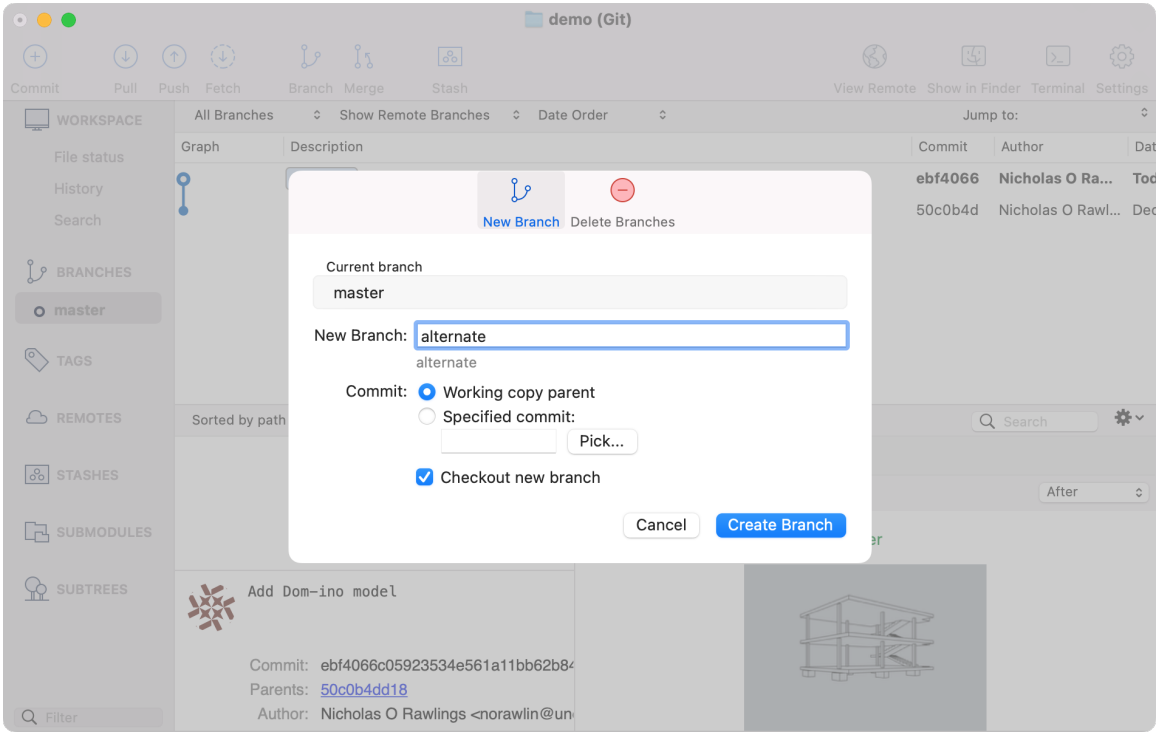


Figure 5.6. Creating a branch in Sourcetree.

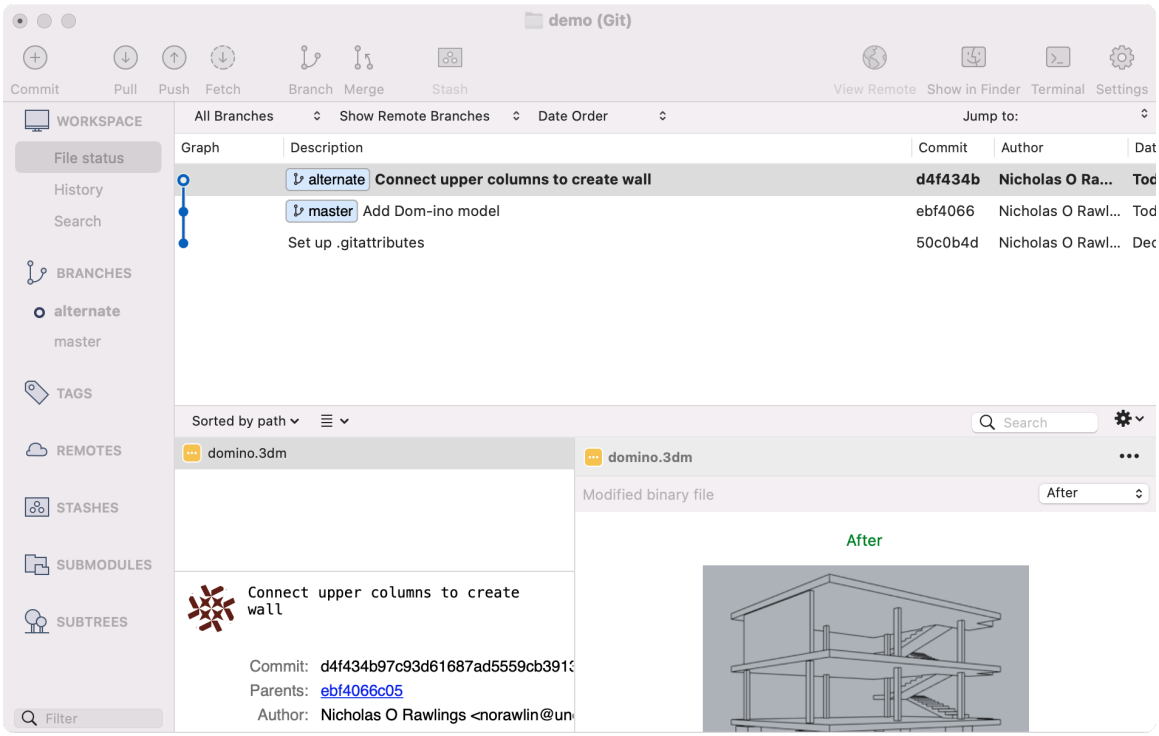


Figure 5.7. The state of the repository after committing the first set of changes on the `alternate` branch.

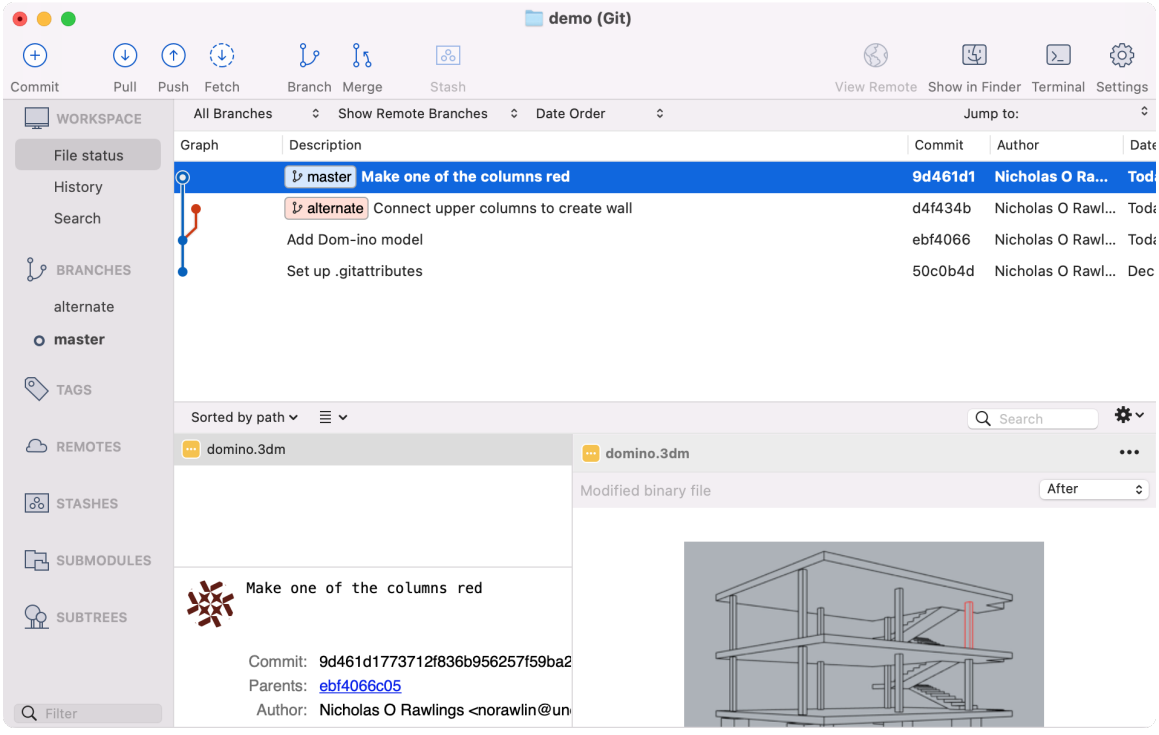


Figure 5.8. The state of the repository after committing the second set of changes on the `master` branch.

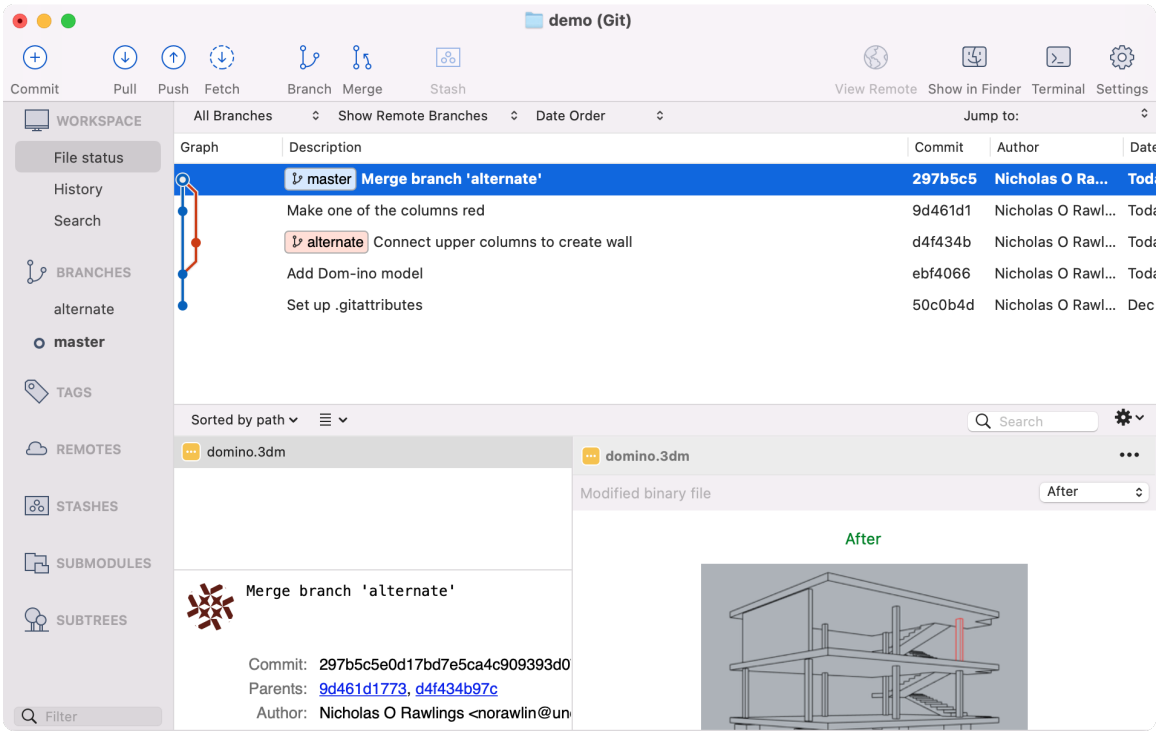


Figure 5.9. The state of the repository after merging the `alternate` branch into the `master` branch.

CHAPTER 6

DISCUSSION

6.1 A Proof of Concept

The `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs developed for this thesis and described in Chapter 4 have been successfully shown to accomplish each of their respective intended tasks. Furthermore, the procedure described in Section 4.4 has been used to integrate these programs into a Git repository, and two branches in that repository which contained separately modified versions of the same openNURBS model were able to be merged. This thesis has therefore successfully demonstrated the possibility of enabling optimistic version control for openNURBS models and, by extension, other binary file formats used in the architecture profession using an off-the-shelf version control system.

However, the `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs as they exist today are far too limited in their support for openNURBS component and geometry types to be of practical use to architects. As of this writing, only `ON_Layer`, `ON_Linetype`, `ON_Group`, and `ON_Material` components are able to be diffed, patched, and merged, along with `ON_ModelGeometryComponent` instances containing `ON_Point`, `ON_ArcCurve`, `ON_LineCurve`, and `ON_TextDot` geometries.

Even among these eight component types, not all properties are fully supported. The `3dmpatch` program, for example, is not yet able to change the stacking order of layers, and `3dmdiff` is unable to examine the textures assigned to a material.

In order to progress beyond the proof-of-concept stage, a significantly broader set of component types and properties must be supported. This will require developing algorithms to handle complex geometric shapes such as polylines, Bézier splines, and NURBS curves and surfaces which are defined by a variable number of control points. These algorithms must be able to detect the addition and deletion of control points between versions of such geometric components, and to do so, they must determine which control points in one version of a shape correspond to the control points in another version of the same shape. Unlike the components in an openNURBS model, control points do not come with a unique ID that can be used to identify them across model versions. They cannot be reliably identified by their position within the shape's overall list of control points because items may have been added to or removed from that list, nor can they be compared by value because the shape itself may have been moved, rotated, or transformed in some other way.

The most promising solution to this dilemma is to employ a *registration* algorithm such as those used for aligning point cloud data collected by 3D scanners as well as in computer vision applications. A point registration algorithm would not only be able to identify correspondences between two sets of control points, but would also produce a best-fit transformation matrix for converting one version of a complex geometric shape into another. One potential

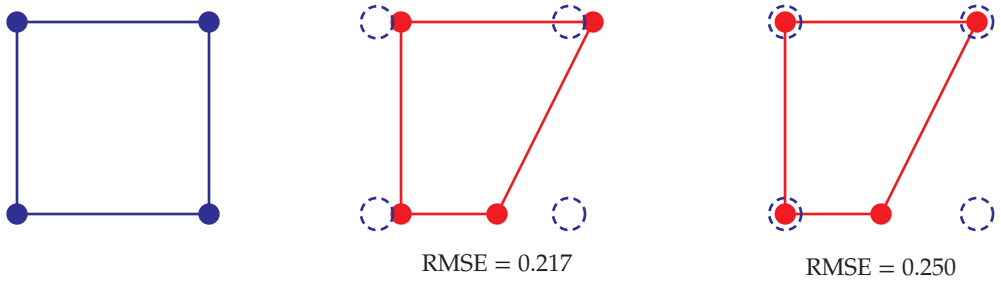


Figure 6.1. Two approaches to point registration. A square (left, in blue) is transformed into a trapezoid (red) by moving its lower right corner inward. A typical point registration algorithm might attempt to minimize the error rate between the old and new sets of control points by producing a result similar to the center diagram. A better alternative for diffing applications would match as many of the points exactly as possible.

difficulty is that these algorithms minimize the average error rate across all the points in the data set, often resulting in a matrix that fails to transform any points in the source data set to the exact location of its counterpart in the target data set. A more desirable outcome for the purposes of diffing would be to arrive at a matrix that produces an exact solution for as many points as possible and distributes any error among the remaining points (Figure 6.1). Additionally, existing registration algorithms are designed to work with data sets consisting of millions of points; it remains to be seen how well they would perform in situations where the number of points often does not exceed a few dozen.

In addition to singular geometric types such as points, curves, and surfaces, openNURBS also supports composite geometries such as `ON_PolyCurve` and `ON_Brep` which are made up of simpler shapes linked together to form a complex whole. The naïve approach to comparing these geometric constructs would be to examine each of their constituent parts independently of the parent geometry.

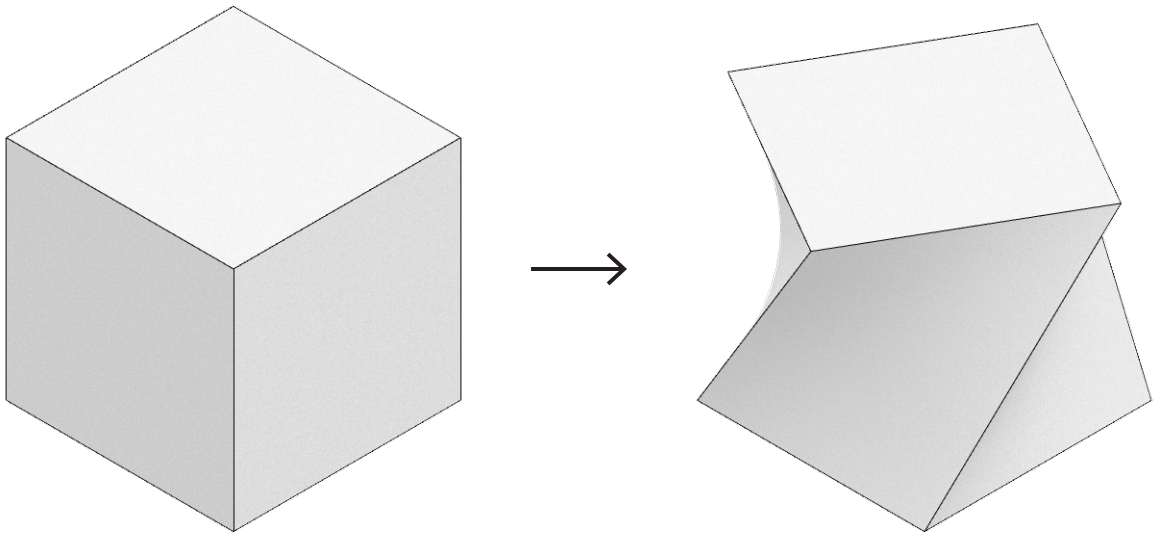


Figure 6.2. A twisted cube. Should a delta describing the differences between these two shapes record only the rotation of the top face, or should it also explicitly include the deformation of the faces that adjoin it?

However, this strategy ignores the efficiencies that could be gained by recognizing that those parts share vertices and edges, and that movement of one edge or face of a composite geometry may be the result of a change to one of its neighbors. The changes to a cube, for example, that has had one of its faces twisted could be documented in a delta that records only the rotation of that one face. The naïve approach, however, would result in a more complex delta that also records changes to the four adjacent faces that were deformed as a result of the rotation. The procedures and algorithms necessary for efficiently handling these types of situations and producing minimal deltas will need to be examined in future work.

$$25,400,000 = 254 \times 10^{5}$$
$$25,400 = 254 \times 10^{2}$$
$$25.4 = 254 \times 10^{-1}$$
$$0.0254 = 254 \times 10^{-4}$$
$$0.0000254 = 254 \times 10^{-7}$$

Figure 6.3. Examples of floating-point numbers

6.2 Floating Point Numbers

Storing integers in a computing system is a relatively simple matter of converting the integer to a base two representation and writing the binary digits to an appropriately-sized section of computer memory. Non-integral real numbers, however, are not so straightforward to deal with. Most computer programs use a *floating-point* representation for such numbers, which breaks them into two integers called a *significand* and an *exponent*. The significand stores the significant digits of the number, while the exponent controls the position of the radix point, which separates the integer part of the number from its fractional part. Increasing or decreasing the exponent causes the radix point to "float" to the right or left, respectively, relative to the number's significant digits. Figure 6.3 shows several examples of floating-point numbers in base ten; floating-point figures in a computer program would, of course, use base two ("IEEE Standard for Binary Floating-Point Arithmetic" 1985).

Floating-point representation allows for space-efficient storage and fast computation of a wide range of numerical values. Unfortunately, limits on the

precision of floating-point numbers due to a finite number of bits being allocated to store the significand and exponent in computer memory can cause them to suffer from round-off errors. In the same way that certain fractions cannot be represented exactly in a limited number of decimal digits (0.333, for example, is close but not precisely equal to $\frac{1}{3}$), other numerical values cannot be represented exactly in base two floating point. The rounding errors that result from approximating such numbers can accumulate during mathematical operations, leading to surprising or unexpected results such as the ones below.

```
print(0.1 + 0.2 == 0.3)  # outputs False
print(0.1 + 0.2)         # outputs 0.30000000000000004
```

The equality operator (==) in Python and most other languages performs a bit-for-bit comparison of floating-point numbers, ignoring the potential for round-off errors and leading to situations where two numbers that should be considered equal are reported as not being so. A better approach to comparing floating-point numbers involves the use of a tolerance: if the absolute value of the difference between the two numbers is less than the tolerance, then they are considered equal. OpenNURBS itself uses this method frequently, and `3dmdiff` should adopt it as well in order to produce more meaningful deltas. The openNURBS format stores tolerances for distances and angles within each model that would be ideally suited for this purpose. Alternatively, a command line option could be added to `3dmdiff` and `3dmdiff3` to explicitly set a tolerance to be used for comparing floating-point values.

6.3 Beyond UUIDs

The `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs rely on UUIDs to quickly and unambiguously find correspondences between the components of openNURBS models. This reliance is not without its drawbacks, however, as it leaves the programs unable to detect semantic relationships between objects in different versions of a model. If two editors, for example, were each to add an object of roughly the same size and shape to the same place in their respective versions of a model, a human mind would recognize that these objects are likely to represent the same thought or entity and attempt to reconcile their differences to arrive at a more complete understanding of that thought or entity. On the other hand, `3mddiff` would treat the two objects as completely unrelated because they do not share a common UUID, and after merging the two versions with `3mddiff3`, the model would be left with two of what should have been recognized as the same object overlapping one another in three-dimensional space.

Temporal and evolutionary relationships are also lost through a singular reliance on UUIDs. When a component in an openNURBS model is duplicated, it receives a new UUID that is unrelated to that of its predecessor. Ideally, `3dmdiff` would be able to detect the relationship between the predecessor and the copy and use that information to construct a more meaningful delta, but its current dependence on UUIDs to detect object correspondences makes this impossible.

The utility of `3dmdiff`, `3dmpatch`, and `3dmdiff3` would be greatly enhanced by incorporating some heuristic to check for object correspondences based on

attributes other than the component's UUID. Unlike the future work discussed in the previous two sections, detection of semantic relationships is a non-trivial problem and a subject of active research with respect to many different fields and types of data. Carra and Pellacini (2019) developed a set of heuristics for matching scene objects, shape data, textures, and animation frames in SceneGit which may be adaptable for the types of components present in openNURBS models. Future work must also evaluate other potential strategies for object correspondence detection, including point cloud registration and machine learning techniques.

6.4 Conflict Resolution

The current behavior of `3dmpatch` and `3dmdiff3` is to abort a patch or merge operation when conflicting edits are detected. Such a reaction is unacceptable for most real-world scenarios, so future work must include the ability to handle conflicts in a more intelligent fashion. One option would be to set aside the conflicting edits for manual inspection and proceed to apply the non-conflicting edits to the patch or merge as usual. This is the approach taken by GNU `patch`, which writes a description of each conflict to a reject file.

Alternatively, `3dmpatch` and `3dmdiff3` could ask the user to choose between predefined methods of resolving a conflict. Git provides a precedent for such behavior: During a merge, a user can use the command `git checkout --ours` to select the version of a conflicted file from the current branch, and the command `git checkout --theirs` to select the version from the branch being imported. A

conflict found during an openNURBS patch operation would need to present three options:

1. To apply the conflicting update, if possible, and proceed with the remainder of the patch.
2. To skip the conflicting update and proceed with the remainder of the patch.
3. To abort the entire patch operation.

A conflict found during a merge, on the other hand, would need to present four alternatives:

1. To use the value of the conflicting property from "my" version of the model.
2. To use the value of the conflicting property from "your" version of the model.
3. To keep the original value of the conflicting property and discard the changes made in "my" and "your" versions of the model.
4. To abort the entire merge operation.

Asking the user to choose between options such as these would be a simple method of implementing interactive conflict resolution, at least from the programmer's standpoint. Unfortunately, this strategy ignores the possibility that the most appropriate resolution to a conflict may be a novel creation of the user rather than a preexisting choice. Furthermore, text prompts in a command-line application do a poor job of explaining where in a model a conflict has been detected or what the various alternatives look like. Git and GNU `diff3` handles these problems by inserting *conflict markers* around problematic sections of a file that prevented it from being automatically merged. For example, suppose two programmers both decide to augment a rather simplistic function:

```
# A function that divides one number by another
def divide(numerator, denominator):
    return numerator / denominator
```

One adds a new parameter that, when set to `False`, prevents the function from raising an exception when the denominator is zero:

```
# A function that divides one number by another
def divide(numerator, denominator, shouldRaise=True):
    if not shouldRaise and denominator == 0:
        return None
    return numerator / denominator
```

The other adds a parameter that specifies an alternative value to return in the same situation:

```
# A function that divides one number by another
def divide(numerator, denominator, whenDividingByZero=math.nan):
    if denominator == 0:
        return whenDividingByZero
    return numerator / denominator
```

The changes made by these two programmers conflict with one another, and would result in conflict markers being added to the source file during a merge:

```
# A function that divides one number by another
<<<<<<< HEAD
def divide(numerator, denominator, shouldRaise=True):
    if not shouldRaise and denominator == 0:
        return None
=======
def divide(numerator, denominator, whenDividingByZero=math.nan):
    if denominator == 0:
        return whenDividingByZero
>>>>>>> other-branch
    return numerator / denominator
```

The programmer in charge of resolving the conflict would then have the option choosing what aspects of each version to discard or retain before completing the merge, perhaps resulting in a hybrid version such as:

```
# A function that divides one number by another
def divide(numerator, denominator,
           shouldRaise=True, whenDividingByZero=math.nan):
    if not shouldRaise and denominator == 0:
        return whenDividingByZero
    return numerator / denominator
```

The insertion of conflict markers is an acceptable means of highlighting merge conflicts in plain text files, but is unsuitable for handling conflicts in openNURBS models because the openNURBS file format has no provision for storing or displaying alternative or conflicting information about a component. Furthermore, resolving conflicts in 3D geometric data demands the ability to visualize the conflict graphically and in three dimensions, a facility that line-based solutions such as conflict markers cannot provide. Future work should therefore devise a way to visualize and resolve conflicts from within an openNURBS modeling application, possibly through a plugin for McNeel's Rhinoceros software.

6.5 Optimism in the Real World

Finally, the efficacy of optimistic version control techniques within an architectural workflow must be validated under real-life conditions. Although this thesis has shown that it is possible to diff, patch, and merge openNURBS

models in a version control repository, it remains to be seen how architectural professionals would react to this promising yet unfamiliar workflow and whether its adoption would result in quantifiable benefits to the industry. Furthermore, it remains possible (although unlikely, in the opinion of this researcher) that the potential for edit conflicts during concurrent editing of a file as complex as a 3D model is simply too great for optimistic version control to be a viable method of enhancing collaboration in architectural practice.

6.6 Conclusion

This thesis has demonstrated the possibility of enabling optimistic version control for openNURBS models by implementing a suite of programs that provide facilities for diffing, patching, and three-way merging of files in that format and describing how those programs can be integrated into a Git repository. Furthermore, it has delineated an abstract model through which the capabilities of those programs can be expanded, and support for diffing, patching, and three-way merging of other file formats may be realized. By pursuing and building upon the techniques presented in this thesis, the architecture community can gain the benefits of concurrent work, enhanced collaboration, and rapid exploration of design alternatives that optimistic version control has to offer.

REFERENCES

Aish, Robert. 2000. "Collaborative Design Using Long Transactions and "Change Merge"." In *Promise and Reality: State of the Art versus State of Practice in Computing for the Design and Planning Process*, 107–111. 18th eCAADe Conference. Weimar, Germany, June. ISBN: 0-9523687-6-5.

"IEEE Standard for Binary Floating-Point Arithmetic." 1985. *ANSI/IEEE Std 754-1985* (October 12, 1985). https://doi.org/10.1109/IEEESTD.1985.82928.

Armstrong, Deborah J. 2006. "The Quarks of Object-Oriented Development." *Communications of the ACM* 49 (2): 123–128. https://doi.org/10.1145/1113034.1113040.

Ashtari, Narges. 2018. "Interacting with Design Alternatives: Towards New Tasks and Tools." Thesis, Simon Fraser University, November 28, 2018. Accessed December 23, 2021. https://summit.sfu.ca/item/18716.

Berliner, Brian. 1990. "CVS II: Parallelizing Software Development," 341–352. Winter 1990 USENIX Conference. January 22–16, 1990.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. 1996. "Layers." In *Pattern-Oriented Software Architecture: A System of Patterns*, 1:31–51. New York: John Wiley & Sons. ISBN: 978-0-471-95869-7.

Carra, Edoardo, and Fabio Pellacini. 2019. "SceneGit: A Practical System for Diffing and Merging 3D Environments." *ACM Transactions on Graphics* 38, no. 6 (November 8, 2019): 159:1–159:15. ISSN: 0730-0301. https://doi.org/10.1145/3355089.3356550.

Chacon, Scott, and Ben Straub. 2014. *Pro Git.* 2nd ed. Apress. ISBN: 978-1-4842-0077-3. https://git-scm.com/book/en/v2.

Chen, Hsiang-Ting, Li-Yi Wei, and Chun-Fa Chang. 2011. "Nonlinear Revision Control for Images." *ACM Transactions on Graphics* 30, no. 4 (July 25, 2011): 105:1–105:10. ISSN: 0730-0301. https://doi.org/10.1145/2010324.1965000.

Collins-Sussman, Ben, Brian W. Fitzpatrick, and C. Michael Pilato. 2011. *Version Control with Subversion.* https://svnbook.red-bean.com/.

Cristie, Verina, Nazim Ibrahim, and Sam Conrad Joyce. 2021. "Capturing and Evaluating Parametric Design Exploration in a Collaborative Environment

- A Study Case of Versioning for Parametric Design." In *PROJECTIONS - Proceedings of the 26th CAADRIA Conference,* edited by A. Globa, J. van Ameijde, A. Fingrut, N. Kim, and T.T.S. Lo, 2:131–140. 26th CAADRIA Conference. The Chinese University of Hong Kong and Online, Hong Kong, March 29–April 1, 2021.

Cristie, Verina, and Sam Conrad Joyce. 2017. "Capturing And Visualising Parametric Design Flow Through Interactive Web Versioning Snapshots." *Proceedings of IASS Annual Symposia* 2017, no. 5 (September 28, 2017): 1–8.

———. 2021. "Versioning for Parametric Design Exploration Process." *Automation in Construction* 129 (September). ISSN: 0926-5805. https://doi.org/10.1016/j.autcon.2021.103802.

Cross, Nigel. 2004. "Expertise in Design: An Overview." *Design Studies* 25:427–441. https://doi.org/10.1016/j.destud.2004.06.002.

———. 2006. *Designerly Ways of Knowing.* London: Springer. ISBN: 1-84628-300-0.

Daum, Simon, and André Borrmann. 2016. "Enhanced Differencing and Merging of IFC Data by Processing Spatial, Semantic and Relational Model Aspects." In *Proc. of the 23rd International Workshop of the European Group for Intelligent Computing in Engineering.*

Denning, Jonathan D., and Fabio Pellacini. 2013. "MeshGit: Diffing and Merging Meshes for Polygonal Modeling." *ACM Transactions on Graphics* 32, no. 4 (July 21, 2013): 35:1–35:10. ISSN: 0730-0301. https://doi.org/10.1145/2461912.2461942.

Doboš, Jozef, Carmen Fan, Sebastian Friston, and Charence Wong. 2018. "Screen Space 3D Diff: A Fast and Reliable Method for Real-Time 3D Differencing on the Web." In *Proceedings of the 23rd International ACM Conference on 3D Web Technology,* 1–9. Web3D '18. New York, NY, USA: Association for Computing Machinery, June 20, 2018. ISBN: 978-1-4503-5800-2. https://doi.org/10.1145/3208806.3208809.

Doboš, Jozef, and Anthony Steed. 2012. "3D Diff: An Interactive Approach to Mesh Differencing and Conflict Resolution." In *SIGGRAPH Asia 2012 Technical Briefs,* 1–4. SA '12. New York, NY, USA: Association for Computing Machinery, November 28, 2012. ISBN: 978-1-4503-1915-7. https://doi.org/10.1145/2407746.2407766.

Firmenich, B, C Koch, T Richter, and D G Beer. 2005. "Versioning Structured Object Sets Using Text Based Version Control Systems." In *Proceedings of the*

22nd CIB-W78 Conference on Information Technology in Construction, Institute of Construction Informatics, 8. Dresden.

Fowler, Martin. 2002. "Layering." In *Patterns of Enterprise Application Architecture,* in collaboration with David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford, 17–24. New York: Addison Wesley. ISBN: 978-0-321-12742-6.

Grune, Dick. 1986. *Concurrent Versions Systems, a Method for Independent Cooperation.* Technical Report IR 113. Amsterdam: Vrije Universiteit.

Hunt, Andrew, and David Thomas. 2000. "The Power of Plain Text." In *The Pragmatic Programmer: From Journeyman to Master,* 73–77. New York: Addison-Wesley. ISBN: 0-201-61622-X.

Hunt, James Wayne, and M Douglas McIlroy. 1976. *An Algorithm for Differential File Comparison.* Computing Science Technical Report 41. Bell Telephone Laboratories, July. https://www.cs.dartmouth.%20edu/~doug/diff.pdf.

Koch, Christian, and Berthold Firmenich. 2011. "An Approach to Distributed Building Modeling on the Basis of Versions and Changes." *Advanced Engineering Informatics,* Information Mining and Retrieval in Design, 25, no. 2 (April 1, 2011): 297–310. ISSN: 1474-0346. https://doi.org/10.1016/j.aei.2010.12.001.

Kung, H. T., and John T. Robinson. 1981. "On Optimistic Methods for Concurrency Control." *ACM Transactions on Database Systems* 6, no. 2 (June 1, 1981): 213–226. ISSN: 0362-5915. https://doi.org/10.1145/319566.319567.

MacKenzie, David, Paul Eggert, and Richard Stallman. 2021a. *Comparing and Merging Files.* Free Software Foundation, January 2, 2021. https://www.gnu.org/software/diffutils/manual/.

———. 2021b. "Detailed Description of Unified Format." In *Comparing and Merging Files.* Free Software Foundation, January 2, 2021. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html.

Meagher, Mark, Jeffrey Huang, Nathaniel Zuelzke, Trevor Patt, Guillaume Labelle, and Julien Nembrini. 2015. "Code and Its Image: The Functions of Text and Visualisation in a Code-Based Design Studio."

Digital Creativity 26, no. 2 (April 3, 2015): 92–109. ISSN: 1462-6268. https://doi.org/10.1080/14626268.2015.1045620.

Mens, Tom. 2002. "A State-of-the-Art Survey on Software Merging." *IEEE Transactions on Software Engineering* (New York, United States) 28, no. 5 (May): 449–462. ISSN: 00985589. https://doi.org/10.1109/tse.2002.1000449.

Myers, Eugene W. 1986. "An O(ND) Difference Algorithm and Its Variations." *Algorithmica* 1 (1-4): 251–266. https://doi.org/10.1007/BF01840446.

Peters, Tim. 2004. "PEP 20 – The Zen of Python," August 19, 2004. https://peps.python.org/pep-0020/.

Robert McNeel & Associates. n.d. "The openNURBS Initiative." https://www.rhino3d.com/opennurbs/.

Rochkind, Marc J. 1975. "The Source Code Control System." *IEEE Transactions on Software Engineering* SE-1, no. 4 (December): 364–370. https://doi.org/10.1109/TSE.1975.6312866.

Rönnau, Sebastian, Jan Scheffczyk, and Uwe M. Borghoff. 2005. "Towards XML Version Control of Office Documents." In *Proceedings of the 2005 ACM Symposium on Document Engineering*, 10–19. DocEng '05. New York, NY, USA: Association for Computing Machinery, November 2, 2005. ISBN: 978-1-59593-240-2. https://doi.org/10.1145/1096601.1096606.

Ruparelia, Nayan B. 2010. "The History of Version Control." *ACM SIGSOFT Software Engineering Notes* 35 (1): 5–9. https://doi.org/10.1145/1668862.1668876.

Sakai, Yasushi, and Daisuke Tsunoda. 2015. "Implementation of Decentralized Version Control in Collective Design Modelling." In *Modelling Behaviour: Design Modelling Symposium 2015*, edited by Mette Ramsgaard Thomsen, Martin Tamke, Christoph Gengnagel, Billie Faircloth, and Fabian Scheurer, 383–395. Cham: Springer International Publishing. ISBN: 978-3-319-24208-8. https://doi.org/10.1007/978-3-319-24208-8_32.

Salus, Peter H. 1994. *A Quarter Century of UNIX.* New York: Addison-Wesley.

Salvati, Gabriele, Christian Santoni, Valentina Tibaldo, and Fabio Pellacini. 2015. "MeshHisto: Collaborative Modeling by Sharing and Retargeting Editing Histories." *ACM Transactions on Graphics* 34, no. 6 (October 26, 2015): 205:1–205:10. ISSN: 0730-0301. https://doi.org/10.1145/2816795.2818110.

Schneider, Sven; Braunes Joerg. 2011. "Design Versioning – Problems and Possible Solutions for the Automatic Management of Distributed Design Processes." In *Computer Aided Architectural Design Futures 2011* [*Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures / ISBN 9782874561429*] *Liege* (*Belgium*) *4-8 July 2011, Pp. 669-681.* 669–681. Liege, Belgium, July 4–8, 2011. ISBN: 978-2-87456-142-9.

Schön, Donald A. 1984. "The Architectural Studio as an Exemplar of Education for Reflection-in-Action." *Journal of Architectural Education* 38, no. 1 (October 1, 1984): 2–9. ISSN: 1046-4883. https://doi.org/10/gnxg3v.

Sveen, Atle Frenvik. 2020. "GeomDiff — an Algorithm for Differential Geospatial Vector Data Comparison." *Open Geospatial Data, Software and Standards* 5, no. 1 (July 10, 2020): 3. ISSN: 2363-7501. https://doi.org/10.1186/s40965-020-00076-4.

The Unicode Consortium. 2021. *The Unicode Standard.* 14.0.0. Mountain View, CA: The Unicode Consortium. ISBN: 978-1-936213-29-0. https://www.unicode.org/versions/Unicode14.0.0/.

Tichy, Walter F. 1982. "Design, Implementation, and Evaluation of a Revision Control System." In *Proceedings of the 6th International Conference on Software Engineering.* Tokyo: IEEE, September.

———. 1985. "RCS — A System for Version Control." *Software: Practice and Experience* 15 (7): 637–654. ISSN: 1097-024X. https://doi.org/10.1002/spe.4380150703.

TIOBE. 2022. "TIOBE Index." TIOBE, December. Accessed December 10, 2022. https://www.tiobe.com/tiobe-index/.

Zhang, Qiao. 2021. "ShapeHub: An Online Collaboration Platform Implementing Version Control for Collaborative Design." Thesis, Carnegie Mellon University, May 24, 2021.

APPENDIX A: REPRINT PERMISSIONS

Portions of this thesis are currently being considered for publication in the *International Journal of Architectural Computing*, published by SAGE Journals. As of this writing, SAGE's Green Open Access[1] policy states that original manuscripts may be reused "in your dissertation or thesis, including where the dissertation or thesis will be posted in any electronic Institutional Repository or database."

1. https://us.sagepub.com/en-us/nam/journal-author-archiving-policies-and-re-use

APPENDIX B: OPENNURBS GEOMETRY TYPES

This appendix outlines the breadth of geometric constructs supported by the openNURBS file format by enumerating the subclasses of `ON_Geometry`.

`ON_Annotation` *extends* `ON_Geometry`
: The base class for annotation objects.

`ON_ArcCurve` *extends* `ON_Curve`
: An arc or circle.

`ON_Brep` *extends* `ON_Geometry`
: A boundary representation, or B-rep. In computer graphics, boundary representations are a method of defining three-dimensional shapes in terms of the surfaces that enclose them.

`ON_BrepEdge` *extends* `ON_CurveProxy`
: An edge between two faces in a B-rep.

`ON_BrepFace` *extends* `ON_SurfaceProxy`
: A face in a B-rep.

`ON_BrepLoop` *extends* `ON_Geometry`
: The set of curves that define the edges of a single face in a B-rep.

`ON_BrepTrim` *extends* `ON_CurveProxy`
: A curve that trims a face in a B-rep.

`ON_Centermark` *extends* `ON_Dimension`
: An annotation that marks the center of a circle or arc.

`ON_ClippingPlaneSurface` *extends* `ON_PlaneSurface`
: A planar surface that is used as a clipping plane in one or more viewports.

`ON_Curve` *extends* `ON_Geometry`
: The base class for curves. In openNURBS, all one-dimensional geometric objects are considered to be curves even if they do not “curve” in the conventional sense.

`ON_CurveOnSurface` *extends* `ON_Curve`
: A curve that lies on a surface.

ON_CurveProxy *extends* ON_Curve
: A curve-like structure that is stored as part of another type of geometry. Proxy objects are not saved in an openNURBS file and only exist at runtime.

ON_DetailView *extends* ON_Geometry
: A viewport bounded by a closed planar curve.

ON_DimAngular *extends* ON_Dimension
: An annotation that shows the angle between two curves or surfaces.

ON_Dimension *extends* ON_Annotation
: The base class for dimension annotations.

ON_DimLinear *extends* ON_Dimension
: An annotation that shows the distance between two points.

ON_DimOrdinate *extends* ON_Dimension
: An annotation that shows the distance from the model's origin point.

ON_DimRadial *extends* ON_Dimension
: An annotation that shows the radius or diameter of a circle or arc.

ON_Extrusion *extends* ON_Surface
: A surface created by extruding a curve along a linear path.

ON_Hatch *extends* ON_Geometry
: A planar region filled with a hatch pattern.

ON_HLDCurve *extends* ON_CurveProxy
: A curve that is the result of projecting a three-dimensional object into a two-dimensional hidden line drawing.

ON_InstanceRef *extends* ON_Geometry
: A copy of an ON_InstanceDefinition that has been placed in the model.

ON_Leader *extends* ON_Annotation
: A line that points from an annotation toward the model entity it describes.

ON_Light *extends* ON_Geometry
: A source of light used when rendering a model.

ON_LineCurve *extends* ON_Curve
: A straight line between two points.

ON_Mesh *extends* ON_Geometry
: A polygonal mesh. In computer graphics, meshes are a method of

approximating a complex shapes through a network of planar, polygonal surfaces.

`ON_MeshComponentRef` *extends* `ON_Geometry`
A face, edge, or vertex within a mesh.

`ON_MorphControl` *extends* `ON_Geometry`
An object that controls the morphing behavior of another object.

`ON_NurbsCage` *extends* `ON_Geometry`
A network of control points that defines the shape of a NURBS surface.

`ON_NurbsCurve` *extends* `ON_Curve`
A non-uniform rational basis spline (NURBS) curve. All other types of curve are representable as and can be converted to a NURBS curve.

`ON_NurbsSurface` *extends* `ON_Surface`
A non-uniform rational basis spline (NURBS) surface. All other types of surface are representable as and can be converted to a NURBS surface.

`ON_OffsetSurface` *extends* `ON_SurfaceProxy`
A surface that is offset from another according to some algorithm.

`ON_PlaneSurface` *extends* `ON_Surface`
A planar surface.

`ON_Point` *extends* `ON_Geometry`
A point in three-dimensional space.

`ON_PointCloud` *extends* `ON_Geometry`
A collection of points such as those acquired by a 3D scanner.

`ON_PointGrid` *extends* `ON_Geometry`
A set of regularly-spaced points.

`ON_PolyCurve` *extends* `ON_Curve`
A curve that consists of other, simpler curves laid end-to-end.

`ON_PolyEdgeCurve` *extends* `ON_PolyCurve`
A multi-part edge of a face in a B-rep.

`ON_PolyEdgeSegment` *extends* `ON_CurveProxy`
A piece of a `ON_PolyEdgeCurve`.

`ON_PolylineCurve` *extends* `ON_Curve`
A curve that consists of one or more linear segments.

ON_RevSurface *extends* ON_Surface
: A surface created by rotating a curve about an axis.

ON_SubD *extends* ON_Geometry
: A subdivision surface. In computer graphics, subdivision surfaces are a method of defining complex shapes through a network of control points and a set of algorithms that interpolate the surfaces between those control points.

ON_SubDComponentRef *extends* ON_Geometry
: A face, edge, or vertex within a subdivision surface.

ON_SumSurface *extends* ON_Surface
: A surface created by extruding a curve along a curved path.

ON_Surface *extends* ON_Geometry
: The base class for surfaces.

ON_SurfaceProxy *extends* ON_Surface
: A surface-like structure that is stored as part of another type of geometry. Proxy objects are not saved in an openNURBS file and only exist at runtime.

ON_Text *extends* ON_Annotation
: A textual annotation.

ON_TextContent *extends* ON_Geometry
: A textual label that is part of an annotation component.

ON_TextDot *extends* ON_Geometry
: A small textual label that keeps itself aligned with the viewing plane so it is always legible.

ON_Viewport *extends* ON_Geometry
: The field of view of a virtual camera.

APPENDIX C: OPEN SOURCE CONTRIBUTIONS

The rhino3dm package provides bindings for the openNURBS library for the Python programming language. Unfortunately, those bindings are incomplete; a number of functions and data types defined in the openNURBS C++ library are not available in Python. During the course of this thesis, seven pull requests containing enhancements and corrections were submitted to the rhino3dm Github repository[1]. As of this writing, four of those pull requests have been accepted by Robert McNeel & Associates and merged into the rhino3dm source.

Linetype Bindings

Pull request #477 seeks to enable inspection and manipulation of line types by adding the `Linetype` and `File3dmLinetypeTable` classes. It has not yet been accepted.

Editorconfig File

Pull request #478 added an `.editorconfig` file[2] to the repository to help ensure the project's coding conventions. It was accepted on September 5, 2022.

1. https://github.com/mcneel/rhino3dm

2. https://editorconfig.org/

Equality Operators

Pull request #479 added support for the equality (`==`) and inequality (`!=`) operators on the `Interval`, `Point2d`, `Point2f`, `Point3d`, `Point3f`, `Point4d`, `Vector2d`, `Vector3d`, and `Vector3f` data types. It was accepted on September 8, 2022.

Arc Plane Property

Pull request #480 added a `Plane` property to the `Arc` class. It was accepted on September 18, 2022. An earlier version of this pull request also included a Normal property for the `Arc` class and added a setter to the `Arc` property of the `ArcCurve` class. Those changes were rejected because they lacked precedent in the RhinoCommon API[3] after which rhino3dm is modeled.

Spelling Corrections

Pull request #481 fixed a few spelling mistakes that were present in the rhino3dm code. It was accepted on October 31, 2022.

3. https://developer.rhino3d.com/guides/rhinocommon/

Group Deletion

Pull request #482 seeks to add a `Delete` method to the `File3dmGroupTable` class.

It has not yet been accepted.

Transform Properties and Methods

Pull request #483 seeks to add the following properties and methods to the

`Transform` class:

`IsAffine` : *boolean*
Indicates whether the `Transform` is affine.

`IsLinear` : *boolean*
Indicates whether the `Transform` is linear.

`IsRotation` : *boolean*
Indicates whether the `Transform` is a proper rotation.

`RigidType` : *TransformRigidType*
Indicates whether the `Transform` is rigid.

`SimilarityType` : *TransformSimilarityType*
Indicates whether the `Transform` is an orientation-preserving or orientation-reversing similarity.

`Rotation` (`startDirection` : *Vector3d*, `endDirection` : *Vector3d*, `center` : *Point3d*) : *Transform*
Creates a `Transform` that rotates from the direction specified by one vector to the direction specified by another.

`PlaneToPlane` (`fromPlane` : *Plane*, `toPlane` : *Plane*) : *Transform*
Creates a `Transform` that moves and re-orients objects in one plane to another.

`Shear` (`plane` : *Plane*, `x` : *Vector3d*, `y` : *Vector3d*, `z` : *Vector3d*) : *Transform*
Creates a shear transformation.

This pull request has not yet been accepted.

APPENDIX D: SUPPORTED COMPONENT TYPES AND PROPERTIES

The `3dmdiff`, `3dmpatch`, and `3dmdiff3` programs described in this thesis currently support the eight component types listed below. The supported properties of each of these component types are enumerated in the sections that follow.

`ArcCurve`
: An arc or circle. Corresponds to an `ON_ModelGeometryComponent` containing an instance of `ON_ArcCurve`.

`Group`
: A named collection of geometric objects. Corresponds to an instance of `ON_Group`.

`Layer`
: A category into which geometric objects can be organized. Corresponds to an instance of `ON_Layer`.

`LineCurve`
: A straight line between two points. Corresponds to an `ON_ModelGeometryComponent` containing an instance of `ON_LineCurve`.

`Linetype`
: A pattern of dashes and spaces that can be applied to a curve. Corresponds to an instance of `ON_Linetype`.

`Material`
: A set of physical and optical properties that affect the appearance of a rendered object. Corresponds to an instance of `ON_Material`.

`Point`
: A point in three-dimensional space. Corresponds to an `ON_ModelGeometryComponent` containing an instance of `ON_Point`.

`TextDot`
: An arc or circle. Corresponds to an ON_ModelGeometryComponent containing an instance of `ON_TextDot`.

Common Properties

The following properties are supported on all component types:

`Name` : *string*
A human-readable label describing the component.

`Parent` : *uuid*
The ID of the component's parent object.

Common Geometric Properties

The following properties are supported on all geometric component types, including `Point`, `ArcCurve` `LineCurve`, and `TextDot`.

`CastsShadows` : *boolean*
Whether or not the object casts shadows during rendering.

`Color` : *color*
The color of the object when displayed on screen.

`ColorSource` : *keyword*
The source of the object's display color. Possible values are:

`layer`	The object takes on the color of its layer.
`material`	The object takes on the color of its material.
`object`	The object is displayed using the color set in its `Color` property.
`parent`	The object takes on the color of its parent.

`Decoration` : *keyword*
Indicates whether decorative marks such as arrowheads are applied to the ends of the object.

`none`	The object has no decorations.
`both`	Decorations are applied to both ends of the object.
`start`	Decoration is applied to the beginning of the object only.
`end`	Decoration is applied to the end of the object only.

DisplayOrder : *integer*
Controls the order in which objects are drawn. Objects with a lower display order appear beneath those with a higher one.

Groups : *set<uuid>*
The UUIDs of the groups that the object belongs to.

Layer : *uuid*
The ID of the layer the object belongs to.

Linetype : *uuid*
The UUID of the object's linetype.

LinetypeSource : *keyword*
The source of the object's plot color. Possible values are:

`layer`	The object takes on the linetype of its layer.
`object`	The object uses the linetype referenced in its `Linetype` property.
`parent`	The object takes on the linetype of its parent.

Material : *uuid*
The UUID of the object's material.

MaterialSource : *keyword*
The source of the object's material. Possible values are:

`layer`	The object takes on the material of its layer.
`object`	The object uses the material referenced in its `Material` property.
`parent`	The object takes on the material of its parent.

PlotColor : *color*
The color of the object when printed.

PlotColorSource : *keyword*
The source of the object's plot color. Possible values are:

`display`	The object is printed using the color set in the `Color` property.
`layer`	The object takes on the color of its layer.
`object`	The object is printed using the color set in its `PlotColor` property.
`parent`	The object takes on the color of its parent.

PlotWeight : *float*
The line weight of the object when printed.

PlotWeightSource : *keyword*
The source of the object's plot weight. Possible values are:

`layer`	The object takes on the line weight of its layer.
`object`	The object is printed using the weight set in its `PlotWeight` property.
`parent`	The object takes on the line weight of its parent.

ReceivesShadows : *boolean*
Whether or not the object receives shadows during rendering.

Space : *keyword*
The space the object belongs to. Possible values are:

`none`	The object does not belong to any space.
`model`	The object exists in model space.
`page`	The object exists in to page space.

Viewport : *uuid*
Restricts the object to a specific view.

WireDensity : *integer*
The density of isocurves used to display the object's surfaces.

Mode : *keyword*
The status of the object. Possible values are:

`hidden`	The object is hidden.
`instanceDefinition`	The object is part of an `ON_InstanceDefinition`.
`locked`	The object is visible but cannot be edited.
`normal`	The object is visible and can be edited.

URL : *string*
The object's URL.

Common Curve Properties

The following properties are supported on all curve types, including `ArcCurve` and `LineCurve`.

Degree : *integer*
The algebraic degree of the curve. The degree of a `LineCurve` is always 1

and for an `ArcCurve` it is always 2, but other types of curves may have higher degrees.

`Domain` : *interval*
The minimum and maximum values of the parameterization of the curve.

Properties of ArcCurve Components

`ArcCurve` components support the following in addition to the common properties listed in the first three sections of this appendix.

`Angle` : *interval*
The angles, in radians, of the start and end points of the arc relative to its center. Circles have an `Angle` value of $[0, 2\pi]$.

`Center` : *point3d*
The location of the center of the arc.

`Normal` : *vector3d*
The normal vector of the plane in which the arc lies.

`Radius` : *float*
The radius of the arc.

Properties of Group Components

`Group` components do not have any properties beyond the common ones listed in

the first section of this appendix.

Properties of Layer Components

`Layer` components support the following properties in addition to the common ones listed in the first section of this appendix.

Color : *color*
The layer's color.

IgesLevel : *integer*
The level assigned to the layer during IGES export.

Linetype : *uuid*
The UUID of the layer's linetype.

Material : *uuid*
The UUID of the layer's material.

PlotColor : *color*
The layer's plot color.

PlotWeight : *float*
The layer's plot weight.

Properties of LineCurve Components

`LineCurve` components support the following in addition to the common properties listed in the first three sections of this appendix.

EndPoint : *point*
The location at which the line ends.

StartPoint : *point*
The location at which the line begins.

Properties of Linetype Components

`Linetype` components support the following property in addition to the common ones listed in the first section of this appendix.

Segments : *list*
The lengths of the dashes and spaces that make up the linetype.

Properties of Material Components

`Material` components support the following properties in addition to the common ones listed in the first section of this appendix.

`AmbientColor` : *color*
The material's ambient color.

`DiffuseColor` : *color*
The material's diffuse color.

`DisableLighting` : *boolean*
Disables lighting for the material.

`EmissionColor` : *color*
The material's emission color.

`FresnelIndexOfRefraction` : *float*
The Fresnel index of refraction of the material.

`FresnelReflections` : *boolean*
Whether Fresnel reflections are used.

`IndexOfRefraction` : *float*
The index of refraction of the material.

`PreviewColor` : *color*
The color used for previewing the material in non-rendered contexts.

`ReflectionColor` : *color*
The material's reflection color.

`ReflectionGlossiness` : *float*
The layer's plot weight.

`Reflectivity` : *float*
The level of reflectivity of the material. A value of 0.0 indicates that the material is not reflective at all and a value of 1.0 indicates that it reflects all of the light that strikes it.

`RenderPlugIn` : *uuid*
The UUID of the rendering plug-in responsible for the material.

`Shine` : *float*
The shine factor of the material.

SpecularColor : *color*
The material's specular color.

Transparency : *float*
The level of transparency of the material. A value of 0.0 indicates that the material is fully opaque and a value of 1.0 indicates that it is fully transparent.

TransparentColor : *color*
The material's transparent color.

Properties of Point Components

`Point` components support the following in addition to the common properties listed in the first two sections of this appendix.

Point : *point3d*
The location of the point.

Properties of TextDot Components

`TextDot` components support the following in addition to the common properties listed in the first two sections of this appendix.

Point : *point3d*
The location of the text dot.

PrimaryText : *string*
The primary content of the text dot.

SecondaryText : *string*
The secondary content of the text dot.

FontFace : *string*
The font face used to render the text dot.

Height : *integer*
The size of the text dot.

APPENDIX E: SOURCE CODE

The complete source code for the abstract model and command-line tools developed for this thesis has been published on GitHub under the MIT license.

`https://github.com/coditect/opennurbs-diffutils`