GEOMETRYEDITOR: A WEB-BASED SYSTEM FOR AUTHORING, SHARING AND SUPPORT OF PLANE GEOMETRY MANIPULATIVES FOR MATHEMATICS EDUCATION

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Xun Lai
August 2010
Dissertation written by

Xun Lai

B.S., Zhongshan University, CHINA 1997
M.S., Zhongshan University, CHINA, 2000
M.A., Kent State University, USA, 2004
Ph.D., Kent State University, USA 2010

Approved by

Dr. Paul S. Wang___________________, Co-Chair, Doctoral Dissertation Committee
Dr. Michael Mikusa_________________, Co-Chair, Doctoral Dissertation Committee
Dr. Arvind Bansal___________________, Members, Doctoral Dissertation Committee
Dr. Hassan Peyravi_________________
Dr. Donald L. White_________________

Accepted by

Dr. Jonathan Maletic_________________, Chair, Department of Computer Science
Dr. John Stalvey_____________________, Dean, College of Arts and Sciences
TABLE OF CONTENTS

LIST OF FIGURES .......................................................................................................................... XI

LIST OF TABLES ............................................................................................................................ XV

DEDICATION .................................................................................................................................... XVIII

ACKNOWLEDGEMENTS .................................................................................................................. XIX

CHAPTER 1 INTRODUCTION .......................................................................................................... 1

1.1 The WME Project ...................................................................................................................... 1

1.2 Geometry Manipulatives and Authoring .................................................................................. 1

1.2.1 Manipulatives ...................................................................................................................... 1

1.2.2 Geometry Manipulative Authoring System ................................................................. 3

1.3 The Need for Web-based Geometry Manipulative Authoring Systems ................................. 4

1.4 Motivations for GeometryEditor ............................................................................................. 6

1.4.1 Existing Systems and Their Limitations ............................................................................ 6

1.4.2 GeometryEditor Research Goals ...................................................................................... 7

1.5 Research Results and Contributions ....................................................................................... 8

1.6 Organization of the Dissertation ............................................................................................. 9

CHAPTER 2 OVERVIEW OF GEOMETRYEDITOR ......................................................................... 12

2.1 How GeometryEditor Started .................................................................................................. 12

2.2 Overview of GMAS ..................................................................................................................... 17

   2.2.1 Fundamental Principle of GMAS ................................................................................ 17

   2.2.2 Authoring Support of GMAS ....................................................................................... 18

   2.2.3 Graph Theory Background for GMAS ........................................................................ 20
2.3 Use Scenarios and User’s Perspectives of GeometryEditor ........................................ 21
2.3.1 Author’s Perspective ........................................................................................................... 22
2.3.2 GeoSite and Client Web Site ............................................................................................ 23
2.3.3 Learner’s Perspective ........................................................................................................ 25
2.3.4 Developer’s Perspective .................................................................................................... 25
2.4 Technical Architecture of GeometryEditor .......................................................................... 25
2.4.1 The Reusable Implementation Layer and GeometryEditor ............................................. 27
2.4.2 Object Data Generators and GeometryEditor ................................................................. 28
2.4.3 GeometryEditor’s Kernel ................................................................................................. 29
2.5 SVG and Geometry Engine Implementation ......................................................................... 30
2.6 GUI Implementation and Some History ............................................................................... 31

CHAPTER 3 MANIPULATIVE AUTHORING SUPPORT .................................................. 33
3.1 Geometric Objects and Expression Objects ........................................................................ 33
3.2 Object Creation ................................................................................................................... 33
3.2.1 General Rule for Object Creation ................................................................................... 34
3.2.2 Tools with Expression Object(s) as Input(s) ................................................................. 34
3.2.3 Expression Object Creation ............................................................................................ 35
3.3 Two Object Creation Modes ............................................................................................... 36
3.4 Tool Signature ..................................................................................................................... 36
3.5 Object Redefinition ............................................................................................................. 38
3.6 Expression Support ............................................................................................................. 39
3.6.1 Expression is everywhere .............................................................................................. 39

iv
3.6.2 Calculator ............................................................................................................. 40
3.6.3 Consistent User Interface to Define an Expression ............................................. 42
3.6.4 Redefinition of the Expression of an Input Expression Object ......................... 45
3.7 Macro Support ......................................................................................................... 47
  3.7.1 Macro Creation Wizard ....................................................................................... 47
  3.7.2 Identical Behavior between System Tools and User-defined Macros .......... 49
  3.7.3 Rollback the Whole Macro ............................................................................... 49
3.8 Coordinate System Support ..................................................................................... 49
3.9 Customizable Menu and Toolbar ........................................................................... 50
  3.9.1 Customizable Menu and Toolbar Saved with Manipulative ......................... 50
  3.9.2 Customizable Menu and Toolbar for the Authoring Environment ............... 51
3.10 Event Handling on Geometric Objects ................................................................ 52
3.11 Protractor and Ruler ............................................................................................ 53

CHAPTER 4 GEOMETRYEDITOR: A COMPLETELY WEB-ORIENTED
INTERACTIVE GEOMETRY SYSTEM ........................................................................ 55

  4.1 Installation and Access of GMAS Packages ....................................................... 55
  4.2 Manipulative Publishing .................................................................................... 59
  4.3 Manipulatives in Two Modes ............................................................................ 60
  4.4 Manipulative Sharing ....................................................................................... 65
  4.5 Scripting with Manipulative Programming Interface ...................................... 68
  4.6 Educational Page Authoring ........................................................................... 69
  4.7 Submittable Manipulatives ............................................................................... 69
4.8 Keywords and Search ......................................................................................................................................... 71
4.9 Web Service by GeoSite .................................................................................................................................... 71
4.10 Integration with Backend Computation Engine .............................................................................................. 72
4.11 GeometryEditor and MathML .......................................................................................................................... 75
4.12 A Library for Building Web Applications ........................................................................................................ 75
4.13 XML Standards .................................................................................................................................................. 75
4.14 Mobile Device ................................................................................................................................................... 76
4.15 Contribution by Everyone ..................................................................................................................................... 76

CHAPTER 5 MANIPULATIVE AND WEB PAGE INTEROPERATION MODEL AND GEOSITE'S PAGE AUTHORING .......................................................................................................................... 77

5.1 Overview ............................................................................................................................................................ 77
  5.1.1 Scenarios of Interoperation .......................................................................................................................... 77
  5.1.2 Continuous Interoperation vs. Interoperation Triggered by Button Clicks ............................................. 81

5.2 Manipulative and Web Page Interoperation Model ............................................................................................ 82
  5.2.1 HTML Contents for Interoperation ........................................................................................................... 82
  5.2.2 Expression in the Interoperation Model ....................................................................................................... 83
  5.2.3 Dependency Granularity .............................................................................................................................. 86
  5.2.4 Interoperation Triggered by Button Clicks ................................................................................................. 87

5.3 GeoSite's Page Authoring ................................................................................................................................ 88
  5.3.1 Authoring of Dynamic Text, Input Boxes, and Buttons ............................................................................. 89
  5.3.2 Authoring of Mathematical Relation ........................................................................................................... 90

CHAPTER 6 TOOLS, MACROS AND EXPRESSIONS ................................................................................................. 92
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Tools with Same Semantics</td>
</tr>
<tr>
<td>6.2</td>
<td>Expressions</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Syntax</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Functions Mapped from Tools</td>
</tr>
<tr>
<td>6.3</td>
<td>Macros</td>
</tr>
<tr>
<td>6.4</td>
<td>Problems in Existing GMAS</td>
</tr>
<tr>
<td>6.5</td>
<td>GeometryEditor’s Approach for Macros and Expressions</td>
</tr>
<tr>
<td>6.5.1</td>
<td>System Macros</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Identification of Acceptable Input Object Types for an Expression</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Identification of Acceptable Input Object Types for a Macro</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Running Time vs. Creation Time Analysis of Macros</td>
</tr>
<tr>
<td>6.5.5</td>
<td>Algorithm for Running Time Analysis of Input Object Types</td>
</tr>
</tbody>
</table>

**CHAPTER 7 GEOMETRY CONSTRUCTION MARKUP LANGUAGE**

**AN INTRODUCTION**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>The Goals of GCML</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Extensibility of Object Types</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Extensibility of ODGs</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Minimum Programming Effort to Add a New ODG</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Consolidation of ODGs with Same Semantics</td>
</tr>
<tr>
<td>7.2</td>
<td>The Simplified Language Grammar</td>
</tr>
<tr>
<td>7.3</td>
<td>A Simple Example</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Object Types and ODGs</td>
</tr>
</tbody>
</table>
7.3.2 Manipulative Data ................................................................. 118
7.3.3 Macro Data ........................................................................... 120

CHAPTER 8 GEOMETRY CONSTRUCTION MARKUP LANGUAGE

THE LANGUAGE AND IMPLEMENTATION ........................................... 124

8.1 Unit Systems ............................................................................ 124
8.1.1 System Units ........................................................................ 124
8.1.2 Origin of the Canvas ............................................................. 124
8.2 Object Types and Core Data for an Object’s State ......................... 125
8.2.1 Primitive Types (Non-Object Types) ...................................... 125
8.2.2 Two Categories of Object Types ........................................... 125
8.2.3 Geometric Object Types ....................................................... 126
8.2.4 Non-Geometric Object Types .............................................. 129
8.3 ODG Instance ........................................................................... 131
8.4 Element <ODG> and the Child Elements ..................................... 134
8.4.1 Element <inputs> ............................................................... 135
8.4.2 Element <parameters> ......................................................... 135
8.4.3 Element <outputs> ............................................................... 136
8.4.4 Element <computation> ....................................................... 136
8.4.5 Element <parameter_update> ............................................. 137
8.4.6 Element <interaction_rule> ............................................... 139
8.4.7 Element <valid_inputs_rule> ............................................. 140
8.4.8 Element <identical_rule> .................................................... 140
8.4.9 ODGs without Inputs but with Only Parameters ........................................ 141
8.4.10 ODGs That Have More Than One Output Objects .................................. 141
8.4.11 ODGs with Indefinite Number of Input Objects ...................................... 143
8.4.12 ODG Expression ...................................................................................... 143
8.4.13 ODG with Indefinite Number of Outputs .................................................. 144
8.5 ODGs Having the Same Name ...................................................................... 145
  8.5.1 Overview .................................................................................................. 145
  8.5.2 Example ODGs PointOnObject ................................................................. 146
  8.5.3 Example ODGs IsIntersect ........................................................................ 147
  8.5.4 Flexibility in Defining ODGs Having the Same Name .............................. 148
  8.5.5 Restrictions on ODGs Having the Same Name ......................................... 150
  8.5.6 User Friendly in Implementation of GCML ............................................. 150
8.6 Run Time Loading of ODG ............................................................................ 151
8.7 Manipulative Data ........................................................................................ 152
  8.7.1 Elements <construct> .............................................................................. 153
  8.7.2 Attributes Color, Visible, Manipulable, and Label for Child Elements of <outputs> .................................................................................................................. 156
8.8 Macro Data .................................................................................................... 157
  8.8.1 Starting with Object Elements ................................................................. 157
  8.8.2 System Macros: Macro Mapped from ODGs with the Same Name .......... 160
  8.8.3 Macros Mapped from an ODG That Has Output Object of Non-geometric Type  161
CHAPTER 9 REUSABLE IMPLEMENTATION LAYER FOR WEB-BASED MANIPULATIVE AUTHORING OR CUSTOMIZATION

9.1 Web-based Manipulative Authoring or Customization Systems .................................. 164
9.2 Overview of the Reusable Implementation Layer......................................................... 166
9.3 Manipulative Instantiation and Management............................................................... 167
  9.3.1 Manipulative Class .............................................................................................. 167
  9.3.2 Manipulative Instantiation .................................................................................. 168
  9.3.3 Synchronization ................................................................................................. 169
9.4 Serialization of a Manipulative .................................................................................... 170
9.5 Dialogs Management ................................................................................................. 171
9.6 Expression Support ..................................................................................................... 172
  9.6.1 Basic Expression Support ................................................................................... 172
  9.6.2 Extensibility ....................................................................................................... 173
  9.6.3 Expression in the Interoperation Model ............................................................. 173
9.7 Converting a Manipulative System to Use the RIL ..................................................... 174

CONCLUSIONS, LIMITATIONS, AND FUTURE WORK ............................................. 175

APPENDIX A GCML XML SCHEMA ............................................................................. 177

APPENDIX B HOW EXPRESSIONS ARE PROCESSED ............................................. 185

REFERENCES .................................................................................................................. 196
LIST OF FIGURES

Figure 1.1 Circumcircle of a triangle ................................................................. 3
Figure 2.1 Manipulative Area of Triangle .......................................................... 12
Figure 2.2 Manipulative Area of Parallelogram .................................................. 13
Figure 2.3 GeometryEditor: Synchronized Copy ............................................... 14
Figure 2.4 GeometryEditor: Synchronized Copy II ........................................... 15
Figure 2.5 SketchPad: Difficult to Make Synchronized Copy .............................. 16
Figure 2.6 Circumcircle of a triangle ................................................................. 18
Figure 2.7 DAG of objects in a manipulative ..................................................... 20
Figure 2.8 Use Scenarios and User’s Perspectives ............................................. 21
Figure 2.9 GeometryEditor Manipulative Authoring Environment on GeoSite .... 22
Figure 2.10 Technical Architecture of GeometryEditor ..................................... 26
Figure 2.11 Object Data Generator .................................................................. 28
Figure 3.1 Widget to Supply an Expression Object ............................................ 34
Figure 3.2 Dynamic Calculator ....................................................................... 35
Figure 3.3 Tool Signature for “ArcThroughThreePoints” ................................... 37
Figure 3.4 Tool Signature for “Translate” ........................................................... 37
Figure 3.5 Pop-up Dialog to Supply Expressions for Number Type Object ......... 38
Figure 3.6 Calculator ...................................................................................... 41
Figure 3.7 UI of Calculator ............................................................................. 42
Figure 3.8 Consistent User Interface to Supply an Expression .......................... 43
Figure 3.9 GeometryEditor Radius of a Circle ................................................................. 43
Figure 3.10 GeometryEditor: Dialog to Plot a Point ..................................................... 44
Figure 3.11 GeometryEditor: Dialog to Translate an Object ......................................... 45
Figure 3.12 GeometryEditor: Property Dialog to Redefine an Object ......................... 46
Figure 3.13 SketchPad: Radius of a Circle (need a new one with measurement on it) .... 47
Figure 3.14 GeometryEditor: Macro Wizard ................................................................. 48
Figure 3.15 GeometryEditor Dialog for Creating Coordinate System I ....................... 49
Figure 3.16 GeometryEditor Dialog for Creating Coordinate System II ..................... 50
Figure 3.17 GeometryEditor: Customizable Menu and Toolbar for a Manipulative ...... 51
Figure 3.18 GeometryEditor: Customizable Menu and Toolbar for the Authoring
Environment .................................................................................................................. 52
Figure 3.19 Dialog to Define How to Handle Events ....................................................... 53
Figure 3.20 Rulers and Protractors .................................................................................. 54
Figure 4.1 GeometryEditor Architecture and Components ........................................ 57
Figure 4.2 GeometryEditor Authoring Environment ..................................................... 57
Figure 4.3 GeoSite – File Organization ........................................................................... 58
Figure 4.4 GeoSite – File Management .......................................................................... 58
Figure 4.5 Default Page for a Manipulative on GeoSite ................................................. 60
Figure 4.6 Manipulative in Learning Mode ..................................................................... 61
Figure 4.7 Manipulative in the Authoring Environment of Geometer’s SketchPad ...... 62
Figure 4.8 Applet-version Manipulative without the Authoring Environment Involved . 63
Figure 4.9 Add Authoring Supports in Manipulative ..................................................... 65
Figure 7.1 A Simple Example of a Manipulative ................................................... 114
Figure 7.2 Object types and ODGs in the Simple Example .................................. 116
Figure 7.3 Construction Flow in the Simple Example.......................................... 118
Figure 7.4 Macro Generated in the Simple Version .......................................... 120
Figure 7.5 Another Two ODGs named “CircleWithCenterAndOnePoint” ............. 121
Figure 7.6 Constructions by Three CircleWithCenterAndOnePoint ODGs .......... 122
Figure 7.7 Internal Interpretation of the Macro Generated in the Simple Example .... 123
Figure 8.1 ODG instance .............................................................................. 132
Figure 8.2 Angle to Determine the Point on a Circle ....................................... 133
Figure 8.3 Pre-defined Variables for <parameter_update> ................................. 138
Figure 8.4 Tool Signature When Only One “IsIntersect” ODG is Loaded .......... 151
Figure 8.5 Tool Signature When Two “IsIntersect” ODGs are Loaded .......... 151
Figure 8.6 Authoring Time Loading of ODG.................................................. 152
Figure 8.7 Menu after an ODG is loaded .......................................................... 152
Figure 8.8 Manipulative with Circle through Three Points and a Point On it .... 153
Figure 8.9 Dependency among ODG instances and Objects .............................. 154
Figure 8.10 Object color determined by an expression ...................................... 157
Figure 8.11 Non-geometric Type Output.......................................................... 162
Figure 9.1 MathEdit in Authoring Mode ....................................................... 164
Figure 9.2 Manipulative Created by MathEdit in Learning Mode ...................... 165
Figure 9.3 Customization of a Manipulative ................................................... 165
Figure 9.4 Interoperation between MathEdit and GeometryEditor .................. 174
LIST OF TABLES

Table 1-1 Steps to construct a circumcircle ................................................................. 4
Table 2-1 Simple interoperation between Client Web Site and GeoSite ......................... 24
Table 2-2 Advanced Interopation between Client Web Site and GeoSite ....................... 24
Table 5-1 HTML dynamic texts and input boxes aware ................................................. 85
Table 5-2 Manipulative expression object names aware ................................................. 86
Table 6-1 Pseudo-data for the macro that creates a circle through three points ............... 96
Table 6-2 System Macro for tool “CircleWithCenterAndOnePoint” .............................. 100
Table 6-3 Identification of acceptable input object types .............................................. 102
Table 6-4 One step macro .............................................................................................. 104
Table 6-5 Multi-step macro ............................................................................................ 105
Table 6-6 Macro with expression .................................................................................... 105
Table 6-7 Macro saved with specific input types ............................................................ 106
Table 6-8 Macro saved without using specific input types ............................................. 106
Table 6-9 Sorting of steps in a macro .............................................................................. 107
Table 6-10 Backtracking algorithm to identify all the acceptable object types ............. 108
Table 7-1 Simplified Grammar for GCML ..................................................................... 113
Table 7-2 XML Schema for <Point> ............................................................................. 115
Table 7-3 XML Schema for common attributes of object types .................................... 115
Table 7-4 GCML for ODGs ............................................................................................ 117
Table 7-5 XML Schema for <ODG> ................................................................. 118
Table 7-6 GCML for manipulatives ............................................................. 119
Table 7-7 GCML for Macro Data ................................................................. 121
Table 8-1 GCML ODG “PointOnObject” ..................................................... 134
Table 8-2 GCML Element <inputs> ............................................................ 135
Table 8-3 GCML Element <parameters> .................................................... 135
Table 8-4 GCML Element <outputs> .......................................................... 136
Table 8-5 GCML Element <computation> .................................................. 136
Table 8-6 GCML Element <parameter_update> ......................................... 138
Table 8-7 GCML ODG “FreePoint” ............................................................... 141
Table 8-8 GCML ODG “CircleThroughThreePoints” ................................... 142
Table 8-9 GCML ODG “Polygon” ............................................................... 143
Table 8-10 GCML ODG “Expression” ........................................................ 144
Table 8-11 GCML ODG “Intersect” ............................................................. 145
Table 8-12 Second GCML ODG “PointOnObject” ....................................... 147
Table 8-13 GCML ODG “IsIntersect” .......................................................... 147
Table 8-14 Second GCML ODG “IsIntersect” ............................................ 148
Table 8-15 GCML ODG “Translate” ............................................................. 149
Table 8-16 Second GCML ODG “Translate” .............................................. 149
Table 8-17 GCML for Saving a Manipulative .............................................. 155
Table 8-18 GCML for Macro “CircleThroughThreePoints” .......................... 158
Table 8-19 GCML for a Multi-step Macro ................................................... 160
Table 8-20 Macro Mapped from ODG “PointOnObject” ......................................................... 160
Table 8-21 Macro Mapped from ODG “CircleThroughThreePoints” .......................... 161
Table 8-22 Macro Mapped from ODG “Polygon” .......................................................... 161
Table 8-23 Macro Mapped from ODG “IsIntersect” ......................................................... 162
Table 8-24 Macro Mapped from ODG “IsIntersect” Using Expression ...................... 163
Table 9-1 Manipulative class inheriting from the RIL’s base class ............................... 168
Table 9-2 Codes for Manipulative Instantiation .............................................................. 168
Table 9-3 A dialog for a manipulative customization ..................................................... 172
DEDICATION

I would like to dedicate this dissertation to my parents who brought me up with their love and encouraged me to pursue knowledge.

To my wife for her support, understanding and encouragement in the past few years.

To my sister who is very close to me, and helped with my studies when I was young.

To my aunt who gave much support to my family.
ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisors, Dr. Paul Wang and Dr. Michael Mikusa, for their support, advice, and encouragement throughout my graduate studies.

I would like to thank my committee members for their support and encouragement: Dr. Bansal, Dr. Peyravi, and Dr. White. Without your suggestions, this dissertation might not have been finished.

Finally, I would like to thank the staff of the Department of Computer Science, Kent State University, and my fellow WME members.
CHAPTER 1
INTRODUCTION

1.1 The WME Project

At the Institute for Computational Mathematics (ICM/Kent), we are developing WME (Web-based Mathematics Education) as a distributed system for supporting, enhancing, and delivering mathematics education at all levels. A pilot project puts WME to in-class trials at Kimpton Middle School (Munroe Falls, Ohio) [46]. Among the many features and advantages of WME [12][61][62], hands-on learning with interactive manipulatives, specifically tailored for individual concepts and skills being taught, remains critically important. To support diagramming, interactive learning, and easy authoring of geometrical manipulatives, we created a system called GeometryEditor (previous code name GeoSVG). Simply put, the purpose of GeometryEditor is to enable anyone, from specialists to school teachers, to easily create, share, modify, and deploy geometry-based interactive manipulatives to teach students mathematical concepts. Developed as part of the entire WME system, GeometryEditor also can be used independently.

1.2 Geometry Manipulatives and Authoring

1.2.1 Manipulatives

Manipulatives provide hands-on experiments to learn mathematical concepts. Manipulatives in classrooms have long been recommended by NCTM [48]. Physical
Manipulatives are physical objects, such as base-ten blocks, algebra tiles, Unifix Cubes, Cuisenaire rods, fraction pieces, pattern blocks and geometric solids that can make abstract ideas and symbols more meaningful and understandable to students [14].

Widespread use of virtual manipulatives began with the advent of personal computers during the late 1970s. Some of them simulate physical manipulatives such as algebra tiles [1], base-ten blocks [5], fraction pieces [18], and pattern blocks [50]. Some provide activities that physical manipulatives are not capable of.

Logo [40] [41] probably is the first well-known virtual manipulative. It allows a user to control the movement of a turtle to form different geometric curves or shapes.

Manipulatives that demonstrate concepts and theorems of geometry shapes like lines, circles, triangles, etc. are called Geometry Manipulatives.

Fundamental principle of geometry manipulatives: geometric objects of a manipulative maintain their mathematical relations under mouse operations or driving data changes.

The dynamic behavior is a great improvement over traditional paper-and-pencil geometry.

Figure 1.1 shows a manipulative of the circumcircle of a triangle. Manipulation of any vertex in the construction will cause the circumcircle to be updated accordingly.
1.2.2 Geometry Manipulative Authoring System

Manipulatives can be created in two ways:

- Developed by ad hoc programming
- Generated by manipulative authoring software

Most manipulatives were created in the first way. But, the second way is more desirable because no programming is involved so that any educator who masters the authoring software can create new manipulatives.

Because geometry is a discipline with strict underlying logic and patterns, it is possible to develop *Geometry Manipulatives Authoring System* (GMAS). Some researchers call this kind of systems *Interactive Geometry Software* (IGS), or *Dynamic Geometry Software* (DGS).
The manipulative in Figure 1.1 can be created by the following construction steps shown in Table 1-1, which can be done in all the well-known Geometry Manipulative Authoring Systems.

1. Draw three independent points A, B, and C.
2. Draw three segments AB, BC, and CA.
3. Draw the mid-points D and E of segments AB and BC.
4. Draw the perpendicular bisectors l1 and l2 of segments AB and BC.
5. Draw the intersection point O of two perpendicular bisectors.
6. Draw a circle centered at O and through point A.

Table 1-1 Steps to construct a circumcircle

1.3 The Need for Web-based Geometry Manipulative Authoring Systems

With the advent of Internet, people began to place virtual manipulatives on the Web. Web-based manipulatives can be Java applet, Flash, SVG, pure HTML based, or of a composite type. They are a self-contained unit in a Web page.

Thousands of manipulatives have been created on the Web since the mid 1990s and more are being created. An example is a geometry applet created in 1996 [29]. Utah State University hosts a national library of virtual manipulatives [49]. However, there is little research on how to best use manipulatives on the Web from the Computer Science point of view, such as how to make manipulative sharing easier, and how to enable educators (not programmers) to compose Web pages that interact with the embedded manipulatives.
There is little attempt to build Web-based manipulative authoring systems, which has many advantages over desktop-application-based authoring software. There are several levels of Web orientations of manipulatives and manipulative authoring systems.

- Can a manipulative be placed online for learners around the world to play with?
- Is it easy for an author to publish a manipulative online?
- Can the authoring be done online instead of using installed software on a computer?
- Is it possible or easy for an author to make an educational Web page around a manipulative?
- Can the HTML contents of the enclosing page interact with the enclosed manipulative?
- Can other users easily embed a manipulative or a page section you make into his/her page?
- Do sharing and searching for useful manipulatives become easy?
- Can a user with some programming knowledge provide extensions to the authoring system? Can other users use the extension immediately?

GeometryEditor is the first attempt to build a fully Web-oriented geometry manipulative authoring system. GeoSite is the Web site that hosts GeometryEditor. Although it focuses on geometry manipulatives authoring, the same ideas hold for any Web-based manipulative authoring software. The experience we gained during building GeometryEditor helps us analyze the common functionality for building other Web-based manipulative authoring systems in the long term.
1.4 Motivations for GeometryEditor

1.4.1 Existing Systems and Their Limitations

Since 1980, Geometry Manipulative Authoring Systems such as Cabri Geometry II [7], Geometer’s SketchPad [22], Cinderella [10], Euclides [16], C.a.R.[9], GeoGebra [21], Kig [31], KSeg [32], etc. have been widely used in schools and colleges all over the world.

Web Orientation: Some of the GMAS packages, especially those developed in non-Java languages were initially designed without considering any usage with the Web. They have inherent difficulties to be tailored for Web usage. For those developed in Java, theoretically, with a little change, the whole package is able to run in a browser. Some of them do have the capability to run the whole system online via Web Start. However, they were still mainly designed as desktop-based systems. There are still many disadvantages or limitations for them to have complete Web orientation. They do not have the answers to most of the Web orientation requirement questions proposed in section 1.3.

Authoring Support: Existing systems vary in the strength of authoring support. Some very useful authoring features are missing in existing systems. For example, none of them supports Synchronized Copy (section 2.1) feature required by the first geometry manipulative deployed on WME. CHAPTER 3 gives more detail on authoring support provided by GeometryEditor but missing in existing systems.
**Extensibility:** Existing systems are closed system. Users are limited to the system built-in tools provided by the software vendors. We want an open ecosystem for interested users with basic programming knowledge to contribute extensions.

**Macro and Expression Support:** Most of the interactive geometry systems support macros. A macro records a sequence of construction steps. But all existing systems have the same problem for macro support:

- Inconsistent behavior between system built-in tools and user-defined macros
- No intelligence to apply a macro or expression to the broadest range of objects
- Expressions are not well integrated with macro support

As a result of the problems and limitations of existing systems, it was a motive to design a new GMAS with complete Web orientation, enhanced authoring support, extensibility and intelligent macro and expression support.

1.4.2 **GeometryEditor Research Goals**

The goals of this research are to:

- Provide complete Web orientation
- Provide well-expected and enhanced authoring support with customizable authoring environment
- Design schema to describe geometric object types, and markup language GCML to describe *Object Data Generator* (*ODG*), which play the core role in system extensibility
- Provide intelligent macro and expression support
• Provide reusable implementation, which has been factored out as one layer, for other
  Web-based manipulative authoring systems

1.5 Research Results and Contributions

This dissertation has the following research contributions:

**Web Orientation:** GeometryEditor is the first completely Web-based GMAS. It
tries to utilize the Web to the greatest extend. Not only can manipulative authoring be
done online, but also models and solutions for the following Web orientation topics are
investigated and proposed:

• Demonstration of the possibility of using SVG in interaction and computation intense
  Web application.

• Manipulative authoring, storing, sharing, publishing and searching.

• Customizable authoring environment by Web masters, and customizable UI for
  manipulatives rendered in learning mode by manipulative authors.

• Interoperability between a manipulative and the enclosing page, and educational page
  authoring that makes scripting manipulatives transparent.

• Interoperability between two different Web sites that both host the GeometryEditor
  for accessing self-contained units, which can be a manipulative or an educational
  page fragment.

• A package for building Web-based application that needs mathematical drawing
  support.

**Authoring Support:** In addition to providing basic authoring support that most
existing systems have, GeometryEditor has a few unique and very important authoring
features such as Synchronized Copy, multiple coordinate systems, event handling on geometric objects, tool signature, and strong support for expressions and macros.

**Extensibility:** GeometryEditor defines XML schema for geometric object types such as Point, Circle, etc, and also non-geometric object types such as Number, Boolean, Color, and Text. GCML is the language to describe an *Object Data Generator (ODG)*, which takes some input object(s) and generates output object(s). From the user’s point of view, ODGs are represented by construction tools from the menu items or the toolbar buttons in the GUI. GCML allows users with some basic programming background to provide extension of authoring support to the system.

**Macro and Expression Support:** GeometryEditor solves a few macro and expression problems that exist in other systems. GeometryEditor is able to apply a macro or an expression to the broadest range of objects by analyzing the macro or the expression during running time instead of creation time so that the behavior between system built-in tools and user-defined macros is consistent. Unlike the other systems, GeometryEditor is able to roll back a macro completely instead of partially.

**Reusable Implementation:** GeometryEditor has factored out a layer that forms the infrastructure for building Web-based manipulative authoring systems. This layer has nothing to do with geometry, but provides reusable solutions for a few commonly encountered problems such as expression parsing, manipulative data serialization, dialog management, etc. CHAPTER 9 will give more detail.

### 1.6 Organization of the Dissertation
The dissertation is organized as follows:

Chapter 2 gives the overview of GeometryEditor: how the research on GeometryEditor started; architecture of GeometryEditor from user’s point of view; technical architecture of GeometryEditor; SVG and Geometry Engine implementation.

Chapter 3 introduces the authoring support provided by GeometryEditor, and describes various use cases from the user’s perspective. It emphasizes authoring features that are important and other existing systems are not capable of.

Chapter 4 explains in detail all of the Web orientation aspects of manipulative authoring in GeometryEditor. It explores all of the utilization of the Web by GeometryEditor and the GeoSite, and points out the advantages over existing systems.

Chapter 5 goes further in Web orientation with emphasis in manipulative and enclosing page interoperation model and educational page authoring.

Chapter 6 looks into two critical components for authoring support: macros and expressions. It points out the deficiency and problems in existing systems, and identifies the root cause. The solution provided by GeometryEditor is given with theoretical analysis.

Chapter 7 describes the goals of Geometry Construction Markup Language (GCML) and gives examples for the schema describing object types and GCML describing Object Data Generator (ODG), manipulative data, and macro data.

Chapter 8 gives detailed introduction of the GCML language. The markup language is able to describe many common scenarios: ODG with expression(s) involved,
multiple ODGs with the same semantic name, indefinite number of input objects or
output objects, etc.

Chapter 9 introduces the reusable implementation layer GeometryEditor is built
upon. This layer provides solutions for a few common problems encountered in the
development and customization of Web-based manipulative authoring systems.

I will complete the dissertation with conclusions, future work, limitations, and the
References.
CHAPTER 2
OVERVIEW OF GEOMETRYEDITOR

This chapter will introduce the history of GeometryEditor, and its architecture.

2.1 How GeometryEditor Started

Back in 2004, the first WME Kimpton Middle school site needed a few geometry manipulatives such as “Area of a Triangle” shown in Figure 2.1, and “Area of a Parallelogram” shown in Figure 2.2.

![Figure 2.1 Manipulative Area of Triangle](image)

Figure 2.1 Manipulative Area of Triangle
Manipulatives shown in these two figures were generated by the earliest version of GeometryEditor. They can still be visited by:

- **Area of a Triangle**: http://wme.cs.kent.edu/kimpton/measure_area_triangle.html
- **Area of a Parallelogram**: http://wme.cs.kent.edu/kimpton/measure_parallel.html

The functionality we needed in these two manipulatives was duplication of a shape. We called it *Synchronized Copy*. In Figure 2.1, we have a duplication of a triangle. In Figure 2.2, we used an altitude of the parallelogram to cut a parallelogram into two parts. Each part has its duplicated copy. The Synchronized Copy functionality has to meet the following four requirements:

- (R1) Whenever the original one changes, the duplicated one should change correspondingly.
• (R2) While maintaining the same shape as the original one, the duplicated one should change in an understandable way.
• (R3) The duplicated shape can also rotate around one point freely.
• (R4) The duplicated shape can move around freely.

To illustrate the idea of *Synchronized Copy* better, let us take a look at the example below (http://wme.cs.kent.edu/geosite/view.php?id=_mB59VSKTUGMLdX):

![Diagram](http://wme.cs.kent.edu/geosite/view.php?id=_mB59VSKTUGMLdX)

**Figure 2.3 GeometryEditor: Synchronized Copy**

We want a duplication of pentagon ABCDE to meet the four requirements (R1 to R4)

• (R1) Whenever ABCDE changes, the duplication A’B’C’D’E’ should change correspondingly, as shown in the Figure 2.4 below.
• (R2) When the user moves point A, we expect point A’ instead of the other four duplicated points to move to maintain the shape. The same applies to any other vertices.
(R3) A'B'C'D'E’ can rotate around one of its vertices.

(R4) A'B'C'D'E’ can move freely without any restriction by the original pentagon.

Without the native support from a GMAS, it is difficult even to duplicate a triangle to barely meet our requirements. Figure 2.5 shows how to make such duplication in SketchPad. Suppose C’ is the free point corresponding to point C. We need to measure one angle of the triangle, make two concentric circles, rotate point B’, make an intersection point to find out A’. The same construction can be done in other GMAS such as GeoGebra.
Figure 2.5 SketchPad: Difficult to Make Synchronized Copy

Constructed in this way, A’B’C’ is a duplication of ABC. It can rotate around point C’, and move around freely. However, it does not meet R2. When we move point B, points A’ and B’ both have to move to maintain the shape, which is not in an intuitive and understandable way. We expect only point B’ to move.

Imagine you want to duplicate a quadrilateral shape, or even a pentagon shown before. Our investigation showed that none of the existing GMAS was able to provide such capability.

That is how we started to develop our own GMAS: GeometryEditor with the *Synchronized Copy* capability.
2.2 Overview of GMAS

2.2.1 Fundamental Principle of GMAS

The fundamental principle of GMAS is: geometric objects of a construction maintain their mathematical relations under mouse operations or driving data changes.

Each object is created by applying one mathematical relation to some other objects created earlier. Once the relation among a few objects is determined, it will not get changed until redefinition occurs. For example, the mid-point of a segment remains at the middle of the segment regardless of the angle or length of the segment being changed by the user interaction. A point in a 2-D plane is uniquely determined by an ordered pair \((x, y)\). To maintain the relation invariance, the system needs to update \((x, y)\) by algebraic computation. So each mathematical relation is represented by an algebraic computation, which we call Object Data Generator (ODG) to be introduced in section 2.4.2.

A manipulative is a set of objects and relationships such that mathematical relations remain invariant while the objects may change.
Figure 2.6 Circumcircle of a triangle

Figure 2.6 shows the same manipulative as in Figure 1.1: the circumcircle of a triangle. Manipulation of any vertex in the construction will cause the circumcircle to be updated accordingly. All the mathematical relations among objects are invariant during the user interaction.

The dynamic behavior is a great improvement over traditional paper-and-pencil geometry.

Gawlick [19], Kortenkamp [33][34] and Laborde [35] laid the mathematical foundations of the dynamic behavior.

2.2.2 Authoring Support of GMAS

Here is a list of user-level features any reasonable GMAS should offer.

- **Drawing primitives**: Making it simple to create basic geometric shapes such as points, lines (segments, rays and vectors), circles (ellipses and arcs), polygons, conics, etc.
• **Geometric object construction**: Constructing a new geometric object subject to mathematical relations with existing objects, for example, creating a line parallel to an existing line and through an existing point.

• **Measurement**: Measuring length, slope, radius, distance, area, circumference, perimeter, angle, coordinates, and parallel, perpendicular, and tangent relations.

• **Loci and Envelopes**: Constructing loci of moving points and envelopes of moving lines.

• **Animation**: Moving and changing objects to illustrate and to demonstrate.

• **Calculation**: Creating and evaluating mathematical expressions based on existing measurements and calculations.

• **Graphing**: Plotting points and function graphs in coordinate systems.

• **Geometric transforms**: Translation, reflection, dilation, and rotation of objects.

• **Defining Macros**: Grouping several steps of a construction into one command.

• **Recursion**: Repeated execution of one command.

• **Defining GUI Operations**: Creating a variety of buttons, user inputs, and sliders in a manipulative.

A series of papers [57][2][30][26] published in a same issue stated that an interactive geometry package should provide authoring support of Euclidean Geometry in dragging mode, macro-construction, trace of points and locus. References [25] [58] discuss how dragging affects the learning of geometry.
2.2.3 Graph Theory Background for GMAS

All the objects in a GMAS generated manipulative form an acyclic directed graph (DAG). Let us still use the circumcircle of a triangle in Figure 2.7 as the example:

![DAG of objects in a manipulative](image)

**Figure 2.7 DAG of objects in a manipulative**

A directed edge from object \( u \) to \( v \) means object \( v \) depends on object \( u \). In the circumcircle example, segment \( AB \) depends on points \( A \) and \( B \); mid-point \( D \) depends on segment \( AB \); perpendicular line \( l_1 \) depends on segment \( AB \) and mid-point \( D \); etc.

If there is a directed edge from \( u \) to \( v \), we call \( u \) a parent of \( v \), and \( v \) a child of \( u \). If there is a directed path from \( u \) to \( v \), we call \( u \) an ancestor of \( v \), and \( v \) a descendent of \( v \).

Algorithms in graph theory for DAG have been studied and modified or enhanced to solve problems in GMAS such as:

- How the user’s movement of an ancestor object affects its descendent object(s)
- How the user’s movement of a descendent object affects its ancestor object(s)
- How objects are updated from top to bottom during interaction
- How to detect and avoid circular dependency among objects
• What are the restrictions to re-define an object
• How to abstract a macro out of a construction

2.3 Use Scenarios and User’s Perspectives of GeometryEditor

![Diagram showing different user perspectives and scenarios involving GeometryEditor](image)

**Figure 2.8 Use Scenarios and User’s Perspectives**

This section will explain different use scenarios and user’s perspectives as shown in Figure 2.8.
2.3.1 Author’s Perspective

The GeoSite (demo site: http://wme.cs.kent.edu/geosite): a Web site that makes the GeometryEditor toolkit available as well as stores manipulatives for access, searching, and sharing. It is a repository of manipulatives and educational pages.

Users, typically educators, can author, customize, and save manipulatives and educational pages on GeoSite. Publishing is instant. GeoSite provides an interface for authors to organize manipulatives and pages similar to a file system. Figure 2.9 shows the GeometryEditor manipulative authoring environment on GeoSite.

![Figure 2.9 GeometryEditor Manipulative Authoring Environment on GeoSite](image-url)
GeometryEditor is a Web-based system. Users need just a Web browser in order to author manipulatives or employ them for teaching and learning. GeometryEditor is composed of

a. An SVG-coded *Plane Geometry Engine* for authoring and viewing manipulatives (creating, moving, and animating geometric objects).

b. GUI for the authoring environment providing authoring logic, a variety of dialogs assisting authoring, publishing, and communications with the server side.

The Geometry Engine is an SVG and JavaScript based library, and occupies a rectangular canvas area on the screen. A manipulative in either authoring mode or learning mode needs the engine for rendering and interaction.

The GUI consists of the main menu and toolbar, and a variety of dialogs. Both the engine and the GUI are easily extensible to add new features and functionalities.

CHAPTER 3 will introduce manipulative authoring, and CHAPTER 5 will introduce educational page authoring in detail.

### 2.3.2 GeoSite and Client Web Site

GeoSite is a repository of manipulatives and educational pages, which other Web sites can leverage. There are two ways for a *Client Web Site* to interoperate with GeoSite: simple and advanced interoperations.

**Simple interoperation:** For each manipulative or educational page, GeoSite provides a HTML fragment for users to embed it into their pages hosted on a client Web site.
Table 2-1 Simple interoperaion between Client Web Site and GeoSite

Table 2-1 shows the HTML fragment. The highlighted src attribute refers to GeoSite. In this way, the embedding of a manipulative or educational page fragment from GeoSite into any Web pages in a client Web site can be done by any users with a decent online or offline HTML page editor.

**Advanced interoperaion:** This option is for Web developers who want to build math content rich Web sites. WME and online assessment Web sites like DMAD [2][3] are such examples. GeometryEditor as a JavaScript package is installed on the server side of the client Web site, and loaded into the Web pages.

```javascript
<script type="text/javascript"
    src="/installationPath/RIL.js"></script>
<script type="text/javascript"
    src="/installationPath/GeometryEditor/GeometryEditor.js"></script>
<script type="text/javascript">
    var m = new GeometryEditor( {instanceName: "m"} );
    m.createManipulative({
        renderTo: "container",
            getMan.php?id=_mN5Z6wykfP6M2H"
    });
</script>
<div id="container">
</div>
```

Table 2-2 Advanced Interoperaion between Client Web Site and GeoSite

GeometryEditor provides APIs for Web developers to integrate it into a Client Web site. Table 2-2 shows the JavaScript codes on a Client Web site to render a manipulative whose data comes from the GeoSite. The highlighted part in Table 2-2

```javascript
<script type="text/javascript"
    src="/installationPath/RIL.js"></script>
<script type="text/javascript"
    src="/installationPath/GeometryEditor/GeometryEditor.js"></script>
<script type="text/javascript">
    var m = new GeometryEditor( {instanceName: "m"} );
    m.createManipulative({
        renderTo: "container",
            getMan.php?id=_mN5Z6wykfP6M2H"
    });
</script>
<div id="container">
</div>
```
refers to the manipulative data from the GeoSite, which provides a Web service for data access.

2.3.3 Learner’s Perspective

Users who only do interactive learning can access manipulatives by visiting GeoSite directly or through a Web page on a client Web site that embeds manipulatives stored on a GeoSite. WME lesson pages are examples that use the latter technique which makes GeoSite completely transparent to visitors.

To a learner, a manipulative is displayed in learning mode, with the GUI customized by the manipulative author.

2.3.4 Developer’s Perspective

GeometryEditor distinguishes from other geometry systems for its extensibility by users who have basic programming training. Geometric object types are defined by the GeometryEditor system in XML Schema. Object Data Generators (ODG) described in GCML defines construction tools for authoring such as CircleWithCenterAndOne, IsParallel, etc. CHAPTER 7 and CHAPTER 8 will introduce GCML in great detail, and how users can contribute ODGs.

2.4 Technical Architecture of GeometryEditor

Figure 2.10 shows the technical architecture of the design of GeometryEditor. It shows the relation between the GeometryEditor’s kernel and other components. The Reusable Implementation Layer (RIL) is a layer the kernel is built upon. GCML (Geometry Construction Markup Language) (CHAPTER 7 and CHAPTER 8) plays an
important role. Object Types are defined in XML Schema, and GCML is an XML language to describe ODGs and store manipulative and macro data.

Figure 2.10 Technical Architecture of GeometryEditor
2.4.1 The Reusable Implementation Layer and GeometryEditor

During the development of GeometryEditor, common functionalities that are useful for building other manipulative authoring systems have been identified and factored out as a Reusable Implementation Layer (RIL) (CHAPTER 9).

Reusable Implementation Layer’s Object Types and GeometryEditor’s Specific Object Types

The Reusable Implementation Layer defines object types that are independent from any manipulative authoring systems, such as Number, Boolean, Color, and Text types (section 8.2.4). Upon it, GeometryEditor adds its own specific object types such as Point, Circle, etc (section 8.2.3).

During GeometryEditor’s startup, GeometryEditor’s Object Type Loader loads support for selected object types to the environment. The GeometryEditor’s kernel is not aware of any object types before loading the descriptors. It allows extensibility of object types in the future.

Reusable Implementation Layer’s Expression Support and GeometryEditor’s Extended Support

One of the most important supports by the Reusable Implementation Layer is expression parsing and evaluation. The Reusable Implementation Layer provides support for expression in the following aspects:

- Non-geometric object types introduced in section 8.2.4.
- Units (distance units, angle units, … )
• Functions \((abs, sqrt, rgb, \ldots)\)

• Math symbols \((PI, e, \ldots)\)

GeometryEditor extends the Reusable Implementation Layer’s expression support to have names of geometric ODGs become legal function names, and have geometric objects become legal operands in the expressions.

2.4.2 Object Data Generators and GeometryEditor

An Object Data Generator (ODG) can create output objects in terms of parameters, input objects, and given mathematical and geometric computation rules as shown in Figure 2.11.

![Figure 2.11 Object Data Generator](image-url)
The GeometryEditor’s kernel does not have any built-in ODG. For example, the kernel does not have the algorithm to compute the data to render a segment that connects two points. The algorithm is provided by an ODG.

All the ODGs are external to the GeometryEditor’s kernel. Each ODG is described in a GCML (section 8.4), and the kernel has an ODG Loader module to load the selected ODG.

Each Object Type, either geometric types such as Point, or non-geometric types such as Number, is associated with a data structure, which is described by XML Schema. An ODG provides the algebraic computation to populate the data structure.

2.4.3 GeometryEditor’s Kernel

The GeometryEditor’s kernel consists of the following major modules.

- Object Type Loader
- ODG Loader
- Macro Loader
- System-Macro Generator
- Expression Support
- Macro Execution Support
- Data Parser
- Object Dependency Analyzer
- User Interaction Support
Figure 2.10 shows the relation between the kernel and other components of the system. The object types, ODGs, and macros to be loaded are determined by the combination of the running environment’s configuration and the manipulative to be loaded into the environment. *Object Type Loader* loads the basic object types from the Reusable Implementation Layer and all the selected object types. *ODG Loader* loads all the selected ODGs. Each ODG will be converted into a macro with only one construction step (section 8.8). *Macro Loader* loads all the selected macros and macros converted from ODG. The kernel relies on the Reusable Implementation Layer’s expression support and other support.

### 2.5 SVG and Geometry Engine Implementation

The geometry engine is the kernel of GeometryEditor. It is implemented as an SVG file less than 70KB gzipped for Web delivery containing SVG and JavaScript code. When displayed, it appears as the drawing canvas to the user.

The logic control part of the engine is written in JavaScript and the drawing part, including all geometric objects, is in SVG. Animation uses a combination of SVG and JavaScript. Object-oriented design and implementation techniques help make the programs well-organized and easily extensible.

SVG is critical and it brings these important advantages:

- An open W3C standard and an XML application with wide acceptance and readily available (free) browser support
- Compact vector notation and easy scalability
- Fast loading and easy textual code generation
- Sophisticated coordinate transformation and animation support

Displaying SVG requires a browser that handles SVG either natively, such as Firefox 1.5+, Safari 3.1+, Opera 9+, or via a plug-in such as the Adobe SVG Viewer. The browsers’ native SVG support provides much better communication between SVG objects and the enclosing Web page, an ability we require for the authoring environment of GeometryEditor.

It is worth mentioning that JSXGraph [36] is another geometry package implemented in SVG. It focuses on scripting via APIs without providing any GUI authoring support.

2.6 GUI Implementation and Some History

While the engine provides the core functionalities for GeometryEditor, the GUI makes user control simple and intuitive, an important requirement especially for teaching and learning.

Since GeometryEditor runs in a browser, its GUI depends on the widgets made available by the browser to a Web page.

At the early development of GeometryEditor, the GUI implementation was greatly affected and limited by the browsers’ SVG capability.

In 2004, the only way to view SVG was to use Adobe SVG Viewer plugin to work with browsers on Windows, Linux and Mac. The problem was that except on Windows IE, the plugin lacked the ability to communicate with the enclosing HTML. I had to develop menus, toolbars and dialogs all in SVG. And the GUI communication
with the GeoSite had to be implemented using Adobe SVG Viewer’s `getURL/postURL`, which was a non-standard extension for SVG.

In May 2005, Mozilla Foundation released Firefox 1.5 Alpha, which is the first browser to have native support for SVG. The SVG-HTML cross communication problem is solved. Standard Ajax technique can be used instead of plugin’s non-standard extension. Furthermore, With Firefox, we can also utilize widgets made available in XUL (XML-based User Interface Language) [65] including different menus, listboxes, trees, and tabboxes. Further, our GUI implementation uses XBL (Extensible Binding Language) [64] for creating user-defined widgets. I started to migrate the GUI of GeometryEditor using Firefox’s XUL and XBL technology.

In 2007, Opera 9 started to support SVG natively, and so did Safari 3.1 in March 2008. SVG was well accepted and adopted by major browsers except only Microsoft Internet Explorer. In order to make GeometryEditor be compatible with Safari and Opera, I started to use HTML/CSS to develop the GUI. Widget libraries such as Dynarch.com DHTML menus, Ext JS, and FCKeditor are also used during the development.

The GUI works closely with the Geometry Engine. Some GUI widgets need users to select objects from the canvas controlled by the Geometry Engine. Some GUI widgets activate some actions or events in the Geometry Engine.
CHAPTER 3
MANIPULATIVE AUTHORING SUPPORT

Section 2.2.2 lists authoring features that a good GMAS is supposed to have. The strength to provide these features differs in different GMAS. For example, macro is supported in many GMAS, such as C.a.R., Cabri II, Cinderella 2, GeoGebra, and SketchPad. However, they have different user interface, working flow, and restrictions. Sections in this chapter will introduce authoring support by GeometryEditor from a user’s point of view. In addition to many basic authoring supports, GeometryEditor provides many unique and enhanced authoring features. Deficiency of existing systems regarding authoring support will also be pointed out, and approach by GeometryEditor will be introduced.

3.1 Geometric Objects and Expression Objects

Supported objects that can be created in the GeometryEditor system are categorized into

- **Geometric Objects**: Point, Circle, Segment, Ray, Line, Arc, Polygon, and Locus
- **Expression Objects**: an expression in GeometryEditor can have the following possible result types: Boolean, Number, Color and Text.

3.2 Object Creation

The system provides tools for users to create objects, such as

- creating a circle centered at one point and through another point
• measuring the length of segment

From the user’s point of view, each tool is represented by a menu item or a toolbar button.

From the GCML (CHAPTER 7 and CHAPTER 8) design’s point of view, each tool is backed by one system macro converted from an ODG or multiple ODGs with the same name. But ordinary users have no need to understand anything about system macros, GCML or ODG.

3.2.1 General Rule for Object Creation

In order to use one tool to create an object, a user needs to select one or more objects as input objects. For example, for tool CircleWithCenterAndOnePoint, a user needs to select two points.

3.2.2 Tools with Expression Object(s) as Input(s)

If a tool requires one or more expression objects as the input objects, such as CircleWithCenterAndRadius, a dialog with the widget shown in Figure 3.1 will pop up for users to enter the expression.

![Figure 3.1 Widget to Supply an Expression Object](image)

A user can either select an existing expression or define a new expression using the Dynamic Calculator as shown in Figure 3.2.
Figure 3.2 Dynamic Calculator

3.2.3 Expression Object Creation

For tools that generate *Boolean*, *Number*, *Color* or *Text* type result, an expression object will be created automatically without *Dynamic Calculator* being invoked. For example, the tool that measures the length of an object creates an expression object with *Number* type result; the tool that checks whether two lines are parallel creates an expression object with *Boolean* type result.

A user can also create an expression object with arbitrary complex expression by using the Dynamic Calculator.

CHAPTER 6 will introduce expression in more detail.
3.3 Two Object Creation Modes

In order to create a mid-point of a segment, Cabri II requires you to select the tool from the menu first, and then click a segment on the canvas. SketchPad requires you to select the segment first, and then go to the menu to select the operation. Basically there can be two possible operation modes in creating an object.

- Object selection from the canvas → tool from the menu (SketchPad in this way)
- Tool from the menu → object selection from the canvas (most other geometry software in this way).

GeometryEditor is the only interactive geometry system to allow users to switch between these two modes.

3.4 Tool Signature

Each tool in GeometryEditor expects users to provide some input object(s) to generate one or more output objects.

GeometryEditor provides a Tool Signature Widget to tell users what input object(s) are expected. For example, the ArcThroughThreePoints tool expects the user to select three points. Notice one of the expected input objects is highlighted to let the user know the current step of the tool.
An input of a tool may allow more than one object types. For example, Figure 3.4 shows the signature of the tool *Translation*, which can translate an object with some horizontal and vertical amounts.

The signature widget shows the first input object allows several types.
The second and third object types are *Number*, which means a dialog will pop up to ask users to supply an expression (section 3.4), and the result of the expression must be in the *Number* result.

### 3.5 Object Redefinition

GMAS should provide the most flexibility for redefining an object. From the user’s perspective, a user uses one tool to determine the data of an object. For example, the tool *CircleWithCenterAndOnePoint* with two points as input objects can determine the position of a circle. There are two kinds of redefinition:

- Change one or more input objects using the same tool. In the example, the redefinition can be change to another point for the center.
• Change to another tool with necessary input object changes. In the example, the redefinition can be change to another tool CircleWithCenterAndRadius that takes a Point and a Number as input objects.

GeoGebra has very flexible support for object redefinition among existing GMAS because it allows users to enter the signature.

GeometryEditor provides GUI for users to achieve both kinds of redefinitions. In the second case, the separation of objects and ODGs in GeometryEditor makes it very easy to redefine an object by simply assigning a new ODG to an object.

3.6 Expression Support

GeometryEditor supports calculation in numeric value with units, logical values (true/false), color values using RGB color model, and text value with string manipulation capability. It also provides lots of mathematical, measurement, logical and string manipulation functions.

3.6.1 Expression is everywhere

Many geometry objects in GeometryEditor require expression object(s) as input object(s), or have properties such as style defined by an expression.

Expression Object as input object

• Radius of a circle
• Marker length, and value between two markers of an axis or a ruler
• Dilation factor of a Synchronized Copy
• Distance, angle, or dilation factor of a transformation
• Numerical measurement such as length of a segment
• Boolean measurement such as whether a point is on a circle
• Coordinates of a point plotted in a coordinate system
• Unit length of a ruler
• Number of recursions in a recursive construction

We call it *Input Expression Object*.

*Property of an object*

• Color, width, visibility and other style information of an object
• Label of an object

Few existing geometry software has used expressions to this great extent.

3.6.2 **Calculator**

The calculator is where users enter an expression. It can be invoked from many places: from a tool execution, from an object property dialog, and from many dialogs such as object style setting dialog. Figure 3.6 shows one usage of the calculator.
The expression in this example is:

\[
\text{iff}(\text{distance}(A, BC) > 1.5\text{inches}, \text{rgb}(0, 255, 0), \text{rgb}(255, 0, 0))
\]

The expression uses measurement function, numerical value with units, logical calculation, logical value, color calculation, and an \textit{iff} function together. The expression’s meaning is if the distance from point A to segment BC is greater than 1.5 inches, the result is green color; otherwise the result is red color. The \textit{iff} function in the expression takes three parameters, if the first parameter has \textit{true} value, it returns the second parameter as the result, and otherwise it returns the third parameter as the result.

The author of the manipulative can use this expression to define the color of point A in the object color setting dialog. When point A moves around, we can observe its color changes when its distance to segment BC changes between greater than and less than 1.5 inches.
As shown in Figure 3.7, the calculator provides real-time error message when a user is typing an expression. The Values drop down list lists all the other existing expressions for the user to pick as an operand in the current expression. The Functions drop down list lists all the system functions and functions mapped from ODGs that generate non-geometric object type results.

![Image of Calculator UI](image)

**Figure 3.7 UI of Calculator**

### 3.6.3 Consistent User Interface to Define an Expression

When an input object is an expression object or a property of an object can be determined by an expression, GeometryEditor provides a very consistent user interface for users no matter what type the object is, and what property is. This is not the case in most geometry systems.

The user interface is shown as below. The “Calculator” button can bring up the *Dynamic Calculator.*
Figure 3.8 Consistent User Interface to Supply an Expression

Figure 3.9 shows the dialog for users to define the radius of a circle. The dialog in the middle has the component shown in Figure 3.8. You can either select an existing expression or click the “Calculator” button to define a new expression.

Figure 3.9 GeometryEditor Radius of a Circle

Figure 3.10 shows the dialog for users to define the coordinates of a point. Component shown in Figure 3.8 appears twice in the dialog for defining abscissa and ordinate.
Figure 3.10 GeometryEditor: Dialog to Plot a Point

Figure 3.11 shows the dialog for users to translate an object. The component shown in Figure 3.8 appears twice in the dialog for defining horizontal and vertical translation. This dialog looks exactly the same as the one in Figure 3.9 except that this one has two input expression objects to define. This standard dialog pops up when a tool requires input expression object(s).
From the three examples, we can see that the way to define input expression object is very consistent in GeometryEditor.

3.6.4 Redefinition of the Expression of an Input Expression Object

Also it is very easy to redefine the expression of an input expression object in the object’s property dialog as shown in Figure 4.11. This is the property dialog for the circle created in Figure 3.11. Users can redefine the expression of the two input expression objects for the horizontal and vertical amounts respectively.
In many geometry systems, a constant number, an *expression* and a *measurement* (such as the length of a segment) are not interchangeable when a user defines a property of an object. Users have to be very careful to select the right format to define an object’s input object.

Most geometry systems provide support for creating “*Circle with a Center and Radius*”. That is, by selecting a point and supplying a numeric value, users can create a circle centered at that point with radius equal to the specified numeric value.

In SketchPad, you need to have a measurement object or an expression object created in advance on the canvas. Then you can select a point and the measurement or expression object to create the circle. This is shown in Figure 3.13.

If you have selected a measurement for the radius, later there is no way to redefine the radius since a measurement object is different from an expression object in SketchPad. This problem exists in many GMAS.
C.a.R. and GeoGebra provides flexibility for redefinition of an object, hence solves this problem. They allow users to change the circle’s parents. You can replace the circle’s parent object, which is a measurement object, by another expression object.

Measurements in GeometryEditor are just expressions using measurement functions. So replacement of a measurement by an expression becomes trivial, without any need to change an object’s parent input object.

### 3.7 Macro Support

A macro records the steps of a construction. Macros and expressions play the core role in GeometryEditor.

#### 3.7.1 Macro Creation Wizard

In many GMAS, the support for macro creation is very simple. Users have to select the correct independent and dependent objects at the same time to create the macro. This is very error prone. It is very easy for users to miss some independent or dependent objects when the construction is very complicated, or include objects they do not want.
GeometryEditor provides a very powerful and intelligent wizard shown in Figure 3.14 for users to create macros. It can help users filter out objects they do not want to create, and find objects that users may miss.

Figure 3.14 GeometryEditor: Macro Wizard

Users only need to select independent geometric objects, and all *Input Expression Objects* will be added to the left panel, and the dependent objects will be listed on the right panel. Whenever the user checks or un-checks a dependent object, the wizard can automatically include missing objects, or exclude objects not needed.

Users also do not need to select any *Input Expression Objects*. The wizard is intelligent enough to put them into the inputs list depending on the checked output objects.
3.7.2 Identical Behavior between System Tools and User-defined Macros

All the existing GMAS systems fail to provide identical behavior between system tools invoked from UI and user-defined macros. GeometryEditor provides a solution to solve this problem. CHAPTER 6 will give theoretical analysis of the root cause, the detail of GeometryEditor’s approach, and a few examples.

3.7.3 Rollback the Whole Macro

GeometryEditor is able to do a complete rollback when a user cancels a macro execution in the middle. Many other GMAS leave what has been constructed before the cancel on the canvas, which may not be desirable.

3.8 Coordinate System Support

The coordinate system support in GeometryEditor is very sophisticated. Users can use expressions to define the distance of the first marker, and the value between two markers. Figure 3.15 and Figure 3.16 show the tabs in the coordinate system dialog.

![Figure 3.15 GeometryEditor Dialog for Creating Coordinate System I](image-url)
Multiple coordinate systems are also supported in GeometryEditor.

3.9 Customizable Menu and Toolbar

3.9.1 Customizable Menu and Toolbar Saved with Manipulative

As in several other geometry systems such as Cinderella and GeoGebra, menu and toolbar in GeometryEditor are customizable. During the saving of a manipulative, a user can select a group of tools to be saved with the manipulative. When the manipulative is embedded in a Web page, only the selected tools will be available. Separating authoring and learning GUI is very important.

This feature is very useful as shown in Figure 3.17. We want to provide limited but necessary tools to students to create a circle through three points.
In our research with the schools this feature of being able to save partial menus was highly demanded by our collaborating teachers. Students using GeometryEditor manipulatives will not get confused or side-tracked and will naturally focus on the intended learning activities.

3.9.2 Customizable Menu and Toolbar for the Authoring Environment

GeometryEditor is flexible enough to allow Web developers to customize menu and toolbar for the authoring environment. All the tools can be grouped in any way a Web developer wants.

Many geometry systems like to group tools according to their type, that is, tools to create points in a menu, tools to create circle in one menu, etc. Take a look at this GeometryEditor sample from

http://wme.cs.kent.edu/geosite/lx/GeometryEditor/samples/specifiedGUI3.html,

whose screenshot is shown in Figure 3.18.
3.10 Event Handling on Geometric Objects

Mouse events on geometric objects in most existing geometry software are limited to just selecting objects or dragging objects around. What if we want something to happen when the mouse clicks on or moves over or out of an object?

Figure 3.19 shows a manipulative from

http://wme.cs.kent.edu/geosite/view.php?id=_m8G1lvrnDBou5d
Figure 3.19 Dialog to Define How to Handle Events

The circle is cut into eight sectors. Users can click on a sector to mark or unmark it. There are two counters next to the circle to count how many sectors are marked or unmarked.

It is impossible for many geometry systems to create such a manipulative. GeometryEditor allows users to define an action represented by a button, and then associate a button with a mouse event on an object.

3.11 Protractor and Ruler

GeometryEditor provides movable and rotatable rulers and protractors. More importantly, the unit length of the rulers can be defined by an expression instead of just in centimeters or in inches, as shown in the Figure 4.2 below.

Students in class showed great interests playing with these virtual tools.
Figure 3.20 Rulers and Protractors
CHAPTER 4

GEOMETRYEDITOR: A COMPLETELY WEB-ORIENTED
INTERACTIVE GEOMETRY SYSTEM

Some of the GMAS packages, especially those developed in non-Java languages were initially designed without considering any usage with the Web. They have inherent difficulties to be tailored for Web usage. For those developed in Java, theoretically, with a little change, the whole package is able to run in a browser. Actually, some of them do have the capability to run the whole system online. However, they were still mainly designed as desktop-based systems. There are still many disadvantages or limitations for them to have complete Web orientation.

Sections in this chapter will explain what a completely Web-oriented geometry system should provide. Some of the Web orientation features, such as section 4.1 (Installation and access of GMAS packages), apply to many Web-based systems that are built against traditional desktop-based applications, such as Google Docs vs. Microsoft Office. Some other Web orientation features, such as section 4.3 (Manipulative in two modes), are more specific to GMAS packages.

4.1 Installation and Access of GMAS Packages

Existing GMAS

Existing GMAS packages are stand-alone applications requiring installation and update on each computer that uses the software.
A few GMAS are written in Java. GeoGebra allows users to run it using Java Web Start technology. The entire C.a.R. system can run as an Applet, but digital signature is required if you want to save a manipulative to your local disk.

Another problem of running these Java-based GMAS is that the whole system has to be downloaded at once. All the dialogs, GUI components, and the graphical kernel are bundled together. The size of the whole system can be very large.

**GeometryEditor and Geosite**

GeometryEditor is a Web-based GMAS package that runs on a Web server. The authoring environment can be accessed via a browser. The graphical kernel of the authoring environment is implemented in SVG. Hence unlike existing geometry systems, the GeometryEditor package only needs installation and update on the server, which is transparent to ordinary users.

The constructed manipulatives are stored on the server, and can be retrieved later for editing, configuring, or interactive learning.

In order to organize manipulatives in a systematic way, the server provides user account management, and folder/manipulative hierarchy management.

Figure 4.1 shows the interaction between browsers and the Geosite that hosts GeometryEditor. Browsers require SVG support for the graphical kernel of the authoring environment.
Figure 4.1 GeometryEditor Architecture and Components

Figure 4.2 shows the authoring environment of GeometryEditor served by the GeoSite. The graphical kernel renders geometric objects in SVG, and all the GUIs supporting authoring such as menus, toolbars, assistant dialogs and any other GUI components are written in HTML/CSS. Each component is only downloaded when it is needed.

Figure 4.2 GeometryEditor Authoring Environment

GeoSite organizes manipulatives similar to a file system. Users can organize the files (manipulatives or pages) and links in folders. Figure 4.3 shows the files under
registered user xunlai. In the folder there are a sub-folder “geometry”, and a few manipulatives, pages, and links. The “equilateral triangle with vertices on three circles” item holds a link to a manipulative physically located under another user “joe”. The item “circle and its equation” is an educational page, which will be introduced in detail in section 5.3. The other items are manipulatives physically located under user “xunlai”.

Figure 4.3 GeoSite – File Organization

Figure 4.4 shows the dual panel mode for a user to copy, move, or link files.

Figure 4.4 GeoSite – File Management
4.2 Manipulative Publishing

Existing GMAS

As said in last section, some existing GMAS packages can export manipulatives as Java applets. They may also generate a very simple HTML file with the applet embedded.

However, publishing the HTML file along with the applet to a Web server is still an obstacle for many school teachers.

GeometryEditor and GeoSite

Since a newly constructed manipulative is saved on the server, publishing the manipulative is automatically done. The GMAS system creates a default Web page for each manipulative. Users besides the author can immediately access the newly constructed manipulative via a browser.

A user goes into the folder in which she wants to create a manipulative. Then the user select the “GeometryEditor” from the drop down “New Manipulative” list to start authoring a new manipulative. Once the authoring is done, the user will be asked for the manipulative name. Then the manipulative will be saved/published to the server.

Figure 4.5 shows the default page for a manipulative named “nine point circle” under user “joe” in learning mode.
4.3 Manipulatives in Two Modes

It is very useful that a manipulative has two modes: authoring mode and learning mode.

Authoring Mode is a mode in which all the authoring supports are available for the author.

Learning Mode is a mode in which only the manipulative is present plus any necessary authoring supports learning requires.

Under learning mode, the user interface of the GMAS package is greatly simplified. Geometric objects in the manipulative can be immobilized on purpose.
On the other hand authors still want the flexibility to include any tool to be included with the manipulative. Figure 4.6 shows a manipulative from the GeoSite. The author wants the students to construct a circle through three given points. So some tools, but not all, need to be included with the manipulative.

![Manipulative in Learning Mode](image)

Please use the given tools to draw a circle through three given points.

**Figure 4.6 Manipulative in Learning Mode**

**Existing GMAS**

Prior to the popularity of Web, there is no easy solution to present a manipulative in two modes. Exactly speaking, a manipulative is always in the authoring mode.

A manipulative generated by a GMAS package always needs that particular package to run it as Figure 4.7 shows. There is no way to detach a manipulative from the complicated software environment. Learners who just want to play with the manipulative
are forced to study the whole package. The learning curve associated with such a package is high for students.

Figure 4.7 Manipulative in the Authoring Environment of Geometer’s SketchPad

It is very common in class that students accidentally have clicked some toolbar button, or accidentally have moved some object that is supposed to be immobilized. With the advent of Web, a natural solution is to place a manipulative in a Web page as Figure 4.8 shows, and hide all the unnecessary features of a GMAS package from a learner. Several geometry systems are able to export manipulatives as Java applets to be embedded in a Web page.
However, when exporting manipulatives as applets, most GMAS go into another dilemma. The applet version manipulative loses all the good features that can be invoked from the menu or toolbar in the authoring environment.

More specifically, only the geometric objects in the canvas area can be exported into the applet. Drawing tools, and many other useful tools available in the menu bar or toolbar in authoring environment cannot be exported. Users can do nothing more than dragging operations with the applet-version manipulative.

There is another noticed problem with some existing GMAS packages exporting manipulatives as applets. These geometry systems were not developed in Java. The vendors had to develop a separate Java library, such as CabriJava [8] for Cabri Geometry, and JavaSketchPad [27] for Geometer’s SketchPad, to render the applet-version manipulative. The Java library is not as good as the original geometry system.
Geometer’s SketchPad whose first version dates back to 1989 is such an example. Some geometric objects are not allowed to be exported into the applet [28].

**GeometryEditor and GeoSite**

A Web-based GMAS stores a manipulative to the server. The GMAS server can present the manipulative in authoring mode or learning mode depending on the request from the user. The author no longer needs to do something like creating applets as learning mode manipulatives.

Although all the GMAS packages use Java applet technology for manipulatives in a Web page, it does not mean Java is the only option. As stated in section 3.1, the graphical kernel can be written in SVG or Flash as well. And it is normal that both the authoring mode and learning mode use the same kernel.

A GMAS package should be flexible enough to allow an author to include as many or as few as needed the authoring tools in the learning mode manipulative. If very few tools will be included with the manipulative, it should be possible to display them as toolbar buttons instead of menu items to have a better user interface.

*A manipulative is the customized authoring software with some objects already pre-constructed on the canvas.*

A manipulative with all the authoring supports included is exactly the same as the authoring software.

GeoSite will present a manipulative in authoring mode only when the user clicks the buttons “New Manipulative” or “Edit Manipulative”. Otherwise, the manipulative in learning mode is returned whenever requested. A learning mode manipulative does not
have to be placed in a page located in the GeoSite. It can be embedded in any teaching Web pages located anywhere other than the GeoSite.

As stated, a manipulative may contain none or all of the authoring support features. Figure 4.9 shows the GUI provided by GeometryEditor for choosing additional features to include in a manipulative. Each item on the left column corresponds to one menu item from the menu bar in the authoring environment.

![Menu Customization GUI](image)

**Figure 4.9 Add Authoring Supports in Manipulative**

### 4.4 Manipulative Sharing

Manipulative sharing has several levels of meaning:

- Publishing the manipulative for other people to learn from (section 4.2).
- Allowing other users to easily get a copy of the manipulative and later build on it (this section).
- Allowing other pages, located in either the same Web site as the manipulative or not, to embed the manipulative (section 4.6 and section 5.3).

- Using the manipulative as an assignment, and later being able to collect manipulation results from different learners (section 4.7).

  This section will focus on the second bullet.

**Existing GMAS**

In existing GMAS, the applet-version manipulative is final, and cannot be further modified. Only the data file that is saved locally on a hard-disk can be used for further modification. In order for another person to get a copy of the manipulative for modification, the author has to upload this data file to a Web server for that person to download, or send it as an attachment by email. Neither is convenient enough.

**GeometryEditor and GeoSite**

Manipulatives can be copied across different users’ folders and even across two GeoSite servers without any difficulty.

Figure 4.4 shows manipulatives on Geosite can be copied across folders and across users.

GeometryEditor also allows users to provide the data URL of a manipulative as shown in Figure 4.10.
In the GeometryEditor GUI, it allows users to load a data URL as shown in Figure 4.11. By this means manipulatives can be copied across two Geosite servers.

After making copy of a manipulative from another user or from another GeoSite, a user can build a new manipulative upon the copy.
4.5 Scripting with Manipulative Programming Interface

Take a look at the thousands of manipulatives on the Web. Most of them do not have the capability to interact with their enclosing Web pages. They cannot easily take advantage of the Document Object Model (DOM) and JavaScript for dynamic interactions.

We want a manipulative to expose a programming interface for JavaScript in a Web page to interact with it. The idea applies to any manipulative, not just those generated by a GMAS package.

It is ideal that the programming interface allows scripts to do anything that a user can do in the GMAS graphical user interface environment. But at least the programming interface should provide the following support:

- Get and set a property of an object
- Register listener for the status change of an object, especially expression objects

**Existing GMAS**

Applet-version manipulatives generated by most existing GMAS packages do not provide any APIs to access the manipulatives. GeoGebra provides a very good scripting support.

**GeometryEditor and GeoSite**

A manipulative together with manipulative programming interface forms a self-contained unit that can be embedded in any Web page, not restricted to pages on a GeoSite. WME lesson pages are such examples. By using the scripting APIs from GeometryEditor, a Web page developer can access the manipulative.
In addition to accessor and event listening APIs, GeometryEditor provides serialization APIs, and APIs for building Web applications (section 4.12).

### 4.6 Educational Page Authoring

Scripting is for programmers. We can standardize some of the manipulatives scripting interfaces so that we can build WYSIWYG page authoring environment to allow users without any programming knowledge to author pages that can interact with the embedded manipulatives. Dynamic HTML text contents determined by manipulatives can be created. Question composition with answer checking is possible.

CHAPTER 5 will propose a manipulative interoperation model that can apply to not only Web-based geometry manipulative, but also any Web-based manipulatives. It will also demonstrate the page authoring on GeoSite based on that interoperation model.

### 4.7 Submittable Manipulatives

A great enhancement by the Web to manipulative usage is that teachers can ask students to play with a manipulative, and submit a manipulation result. For example, a teacher can ask students to visit a manipulative with three pre-drawn points as shown in Figure 4.12. Students are required to construct a circle through the points, and then submit it the teacher.
Figure 4.12 Summitable Manipulative

Figure 4.13 shows the interface to view the submissions by other users.
4.8 Keywords and Search

Existing GMAS

Manipulatives are scattered among computers. Searching for a needed manipulative is difficult.

Web-based GMAS

A Web-based GMAS server is a repository of manipulatives. The server can provide search function for manipulatives across the site. Authors can assign keywords to a manipulative.

GeometryEditor and GeoSite

GeoSite can do the searching based on one more criteria, text inside the SVG files, since SVG file is text-based.

4.9 Web Service by GeoSite

It is not trivial to deploy a manipulative on site A to site B. Neither is it easy to reuse part of a page on site A by site B since a manipulative and its enclosing page has mathematical connection.

Just duplicating the html contents along with applets, Flash, or SVG is not enough. JavaScript representing the mathematical connections also needs to be copied.

GeoSite provides Web service to provide the data describing manipulatives and page fragments in XML or JSON format. A client site can have the GeometryEditor library installed, and send data request to the GeoSite.
4.10 Integration with Backend Computation Engine

This section explains with the help of Web, how a smooth integration of Web-based GMAS and Computer Algebra System (CAS) can be achieved.

Current status of integration

GMAS and CAS have evolved independently. In recent years, researchers have been investigating how to integrate them.

Some projects focus on using GMAS as the graphical interface for CAS. The other projects including GeometryEditor focus on enhancing the functionalities of GMAS by CAS.

There are two approaches to integrate GMAS and CAS:

- Develop a brand new system with both native dynamic geometry support and computer algebra support.

- Develop new GMAS and CAS as two separate systems, or reuse existing GMAS and CAS, and then develop some translator to connect them.

We will use the second approach. Here I will introduce two projects that also use the second approach.

Botana [6] developed a Web-based CAS system called Geometric Discovery, which uses the construction data generated by Geometer’s SketchPad as input. The translator is inside the Geometric Discovery. They provide a Web page for users to upload the SketchPad file to get the result. Since the GMAS is not Web-based, every time a user modifies the construction, the user has to upload the file again.
Roanes-Lozano and Roanes-Macias [52] use existing CAS packages Maple and Derive, and also use GMAS package Geometer’s SketchPad. They built a translator to translate the output from SketchPad to the input of Maple or Derive. All the systems, the CAS, the GMAS, and the translator, run locally. Each step requires operations to save files, and supply files to the next step.

Users are aware of the existence of a CAS in the above two projects. In order to get computer algebra support, their use of a GMAS package is constantly interrupted by having to switch to use the CAS and the translator.

With a Web-bases GMAS and the CAS secretly running on the back-end server, users can focus on the use of the GMAS. The translator is built into the GMAS.

**Possible Enhancements to GMAS by CAS**

Below are listed the possible enhancements a back-end CAS can offer to the GMAS package. As said above, these requests to the CAS will be done in the background. Users cannot tell if the results are computed locally or from the server.

- **Symbolic computation**

  Geometer’s SketchPad has internal algorithms to find out the derivative of a function symbolically not numerically, but is not very reliable for complicated functions. It also cannot perform any integral computations. A CAS can do these computations for the GMAS.

- **Algebraic formulae**
Supplying the construction of a parallelogram ABCD and its two diagonals AC and BD to the CAS, the GMAS can receive the result back from the CAS showing the coordinates of the intersection of the two diagonals are \((Bx+Dx)/2, (By+Dy)/2\).

- **Equation of geometric loci**

  Cabri Geometer can find out the equation of a locus, but it is a numerical solution that loses the connection with the driving geometric objects. The CAS can give the equation of the loci in terms of the driving objects.

  The symbolic result has another advantage over the numerical one, which needs to be re-computed whenever the user drags the mouse.

- **Theorem proving**

  The GMAS sends a construction and a result to be proved to the CAS, and receives the true or false statement from the CAS. For example, to prove the circumcenter, the centroid, and the orthocenter of a triangle are collinear, the GMAS can submit the constructions and a result like `collinear(point1, point2, point3)` to the CAS.

- **Theorem discovery**

  Theorem discovery is also interesting. We want a statement to be true, but we do not know what condition is needed. So the GMAS sends the construction and the statement, and receives the condition from the CAS. For example, we have a construction of a triangle ABC, and the statement is \(AB^2 + BC^2 = CA^2\). The CAS will return something like `AB perpendicular to BC`.

  **GeometryEditor and GeoSite**
With the CAS system *Maxima* running on the server side, some simple experiments that can prove the concept have been done on GeometryEditor for the first three points listed above.

### 4.11 GeometryEditor and MathML

If the browser running GeometryEditor supports MathML [44] such as Firefox, the calculator displays an expression in MathML. GeometryEditor is able to convert an expression into MathML in presentation markup.

GeometryEditor is one of the few Web applications that demonstrate the integration of XHTML+SVG+MathML.

### 4.12 A Library for Building Web Applications

The GeometryEditor system can be used as a drawing library in other applications such as online assessment systems or bulletin board systems. GeometryEditor provides APIs for dynamic creation and removal of manipulative and instances, and data serialization. Applications that use GeometryEditor instantly have the capability of drawing and drawing submission.

### 4.13 XML Standards

CHAPTER 7 and CHAPTER 8 will introduce the markup language to define an ODG, equivalent to a tool from the user’s perspective. Users with basic programming knowledge can program, test, and save an ODG on Geosite, and other users can use it immediately.
4.14 Mobile Device

If the graphical kernel of the Web-based GMAS is implemented in SVG, it is possible that in the future the authoring environment can be run on many mobile devices. SVG has a tiny version for mobile devices: The Scalable Vector Graphics Tiny 1.2 Specification. http://svg.org/special/svg_phones has a list of all the SVG-enabled cell phones.

TI graphing calculators use Cabri Geometry. With SVG and Web-based GMAS, it is possible to turn smart phones into TI graphing calculators.

4.15 Contribution by Everyone

Educators can contribute manipulatives and educational pages. Developers can contribute ODGs.

GeoSite is able to host manipulatives and manipulative authoring systems built on the Reusable Implementation Layer (CHAPTER 9).

In this way, a GeoSite can become a repository of useful manipulatives and authoring software.
CHAPTER 5
MANIPULATIVE AND WEB PAGE INTEROPERATION MODEL AND
GEOSITE’S PAGE AUTHORING

5.1 Overview

One of the goals of developing a Web-based GMAS is to find out an
interoperation model that can describe the interoperability among manipulatives and the
enclosing Web page. The implementation of the model should allow users to define the
interoperation without any programming knowledge.

The educational page authoring environment hides the underlying
GeometryEditor scripting APIs and presents a WYSIWYG user interface to users to
compose Web pages and interaction among manipulatives and the enclosing page.

5.1.1 Scenarios of Interoperation

Below are examples to illustrate four possible interactions between a manipulative
and the enclosing page from a user’s point of view:

- Manipulative to HTML content
- HTML content to manipulative
- Manipulative to manipulative
- HTML content to HTML content

The example in Figure 5.1 shows that the manipulative drives the HTML content.
When point A or B moves around, the radius of the circle changes correspondingly, as
does the result of the expression object that measures the radius of the circle. This expression object is the bridge between the manipulative and the HTML content. In the HTML content, there are two highlighted dynamic texts. The value of each dynamic text is defined by an expression objects.

Here is an HTML section. We can show here the radius of the circle in the manipulative.

The radius is 0.81 inches.
The area is 2.08 inches^2.

**Figure 5.1 Manipulative to HTML**

The example in Figure 5.2 shows that the HTML content drives the manipulative. In the manipulative, there are four expression objects, whose labels are “a”, “b”, “c” and “d”. Each of them is driven by an input box in the HTML. A function is defined in terms of these four expression objects. Whenever the user changes the value in any of the input boxes, the plotting of the function will be updated.
Figure 5.2 HTML to Manipulative

The example in Figure 5.3 shows that a manipulative drives another manipulative. In the first manipulative, there is an expression object that measures the radius of the circle. In the second manipulative, an expression object is used in the construction CircleWithCenterAndRadius to define a circle. The expression object in the second manipulative is driven by the expression object in the first manipulative.
Figure 5.3 Manipulative to Manipulative

The example in Figure 5.4 shows one part of the HTML content drives another part of the HTML content. It is interesting because there is no manipulative developed in SVG, Flash or Java Applet at all. The highlighted dynamic text in the “Amount” column is defined as the value in the input box in the first column multiplied by the highlighted dynamic text in the second column.
When a source drives a target, there are two update scenarios:

- The target gets continuously updated whenever there is change in the source.
- The update of the target is triggered by the user clicking a button

In the above four examples, each one can have a button in the Web page, and the update is delayed until the user clicks the button. Figure 5.5 shows a similar page as Figure 5.1 with an addition of a button labeled “Update”. The two highlighted span elements will not get updated until the user clicks the button.
5.2 Manipulative and Web Page Interoperation Model

The analysis in this chapter and the interoperation model are GMAS-independent. The model applies to any Web-based manipulative, whether it is created by authoring systems or by ad-hoc programming.

5.2.1 HTML Contents for Interoperation

There are three special HTML contents in a page defined by the interoperation model:

- Dynamic Text: represented by HTML `<span/>` elements. A dynamic text object is associated with an expression object, which controls the value of the dynamic text.
• Input boxes: represented by HTML `<input type="text"/>` element. An input box is associated with an expression object, whose expression is controlled by the content in the input box.

• Button: represented by HTML `<input type="button" />` element. Each button is associated with a sequence of assignments, which use the result of one expression object to redefine another expression object.

The concept of expression objects mentioned above will be explained in the next section.

5.2.2 Expression in the Interoperation Model

Expression plays the core role in the Interoperation Model. Expression is the only bridge that connects two components, which can be either a manipulative or an HTML fragment, together.

The expression support in GeometryEditor actually extends the basic expression support provided by the Reusable Implementation Layer (CHAPTER 9) so that it is aware of the geometric objects and ODG. For example `len([Segment AB])` becomes a legal expression since the expression support understands symbol `len` refers to an ODG and “Segment AB” refers to a geometric object.

The Interoperation Model extends the expression support from the Reusable Implementation Layer to understand the syntax referring to HTML span elements and input elements, and also expression objects from manipulatives.

Figure 5.6 shows the relation among the expression support in manipulatives authoring systems, the Interoperation Model and the Reusable Implementation Layer.
Expressions support in GeometryEditor
Geometric objects and construction tools aware

Expression Support from the Reusable Implementation Layer

Expressions support in Interoperation Model
HTML dynamic texts, input boxes and manipulative expression names aware

Expressions support in another manipulative authoring system
Local object types and symbols aware

Figure 5.6 Expression support in manipulatives and HTML

HTML dynamic texts and input boxes aware

Table 5-1 shows the HTML fragment that comes from the example in Figure 5.4 (HTML to HTML).

```
<table>
  <tbody>
    <tr>
      <th>Quantity</th>
      <th>Item</th>
      <th>Amount</th>
    </tr>
    <tr>
      <td>
        <input id="prefix_c0" class="lxInput" size="2">
      </td>
      <td>
        Chicken Noodle Soup (\<span id="prefix_p0" class="lxHtmlField">1.95</span>)
      </td>
    </tr>
  </tbody>
</table>
```
There is one input box with id “prefix_c0”, and two span elements with ids “prefix_p0” and “prefix_sub0”. The prefix is some string randomly generated so that the whole HTML fragment can be ported to another page without causing any conflicts. The page author and the learners do not need to care about this prefix.

The input boxes and two span elements are each associated with a different expression object. In a page authoring environment, the expression object backing the span element with id “prefix_sub0” can be defined as

\[ c_0 \cdot p_0 \]

The **Interoperation Model** understands what symbols \( c_0 \) and \( p_0 \) refer to. Symbol \( c_0 \) represents the expression object backing the input box with id “prefix_c0”, while symbol \( p_0 \) represents the expression object backing the span element with id “prefix_p0”.

**Manipulative expression object names aware**

Table 5-2 shows the HTML fragment that comes from the example in Figure 5.1 (Manipulative to HTML) or Figure 5.5.
Table 5-2 Manipulative expression object names aware

The expression object backing the span element with id “prefix_area” is defined as:

\[ \pi \times m0\{Radius \text{ Circle AB}\} \times m0\{Radius \text{ Circle AB}\} \]

The Interoperation Model understands \( m0\{Radius \text{ Circle AB}\} \) refers to the expression object named “Radius Circle AB” in the manipulative named “m0”. The result is of Number type with distance unit of order 2.

5.2.3 Dependency Granularity

The Interoperation Model does not know the internal implementation of a manipulative system. The expressions defined inside the Interoperation Model may form circular dependency as shown in Figure 5.7.

In the figure the object at the end of an arrow depends on the object at the starting point of the arrow. If expression object 2 depends on expression object 1 in the manipulative, a circular dependency is formed.
The Interoperation Model provides a solution at the coarse granularity level. It treats a manipulative as a black box. It simply does not allow the situation in Figure 5.7 to happen regardless of dependency between two expression objects in the manipulative.

A manipulative can provide an API

\[
\text{dependsOn(name\_of\_object1, name\_of\_object2)}
\]

about which the Interoperation Model can query. In this way, the Interoperation Model can provide a fine granularity level of control of the dependency.

### 5.2.4 Interoperation Triggered by Button Clicks

A button is associated with a sequence of assignments. For example, the button in Figure 5.5 is associated with the following assignments.

\[
r = m0(\text{Radius Circle AB})
\]

\[
area = \pi m0(\text{Radius Circle AB})^2 m0(\text{Radius Circle AB})
\]

where symbols \( r \) and \( area \) represent the two dynamic text fields each backed by an expression object.
In continuous interoperation, the expression object \( area \) is bound to the expression \( PI \times m0\{Radius\ Circle\ AB\} \times m0\{Radius\ Circle\ AB\} \) so that \( area \) depends on \( m0\{Radius\ Circle\ AB\} \), and any change of \( m0\{Radius\ Circle\ AB\} \) causes the re-evaluation of the expression.

In the button-trigger-interoperation, the expression object \( area \) has nothing to do with expression \( PI \times m0\{Radius\ Circle\ AB\} \times m0\{Radius\ Circle\ AB\} \). The expression on the right of an assignment is stored in the data of the button, not the data of the dynamic text \( area \).

The *Interoperation Model* uses the result from the expression on the right to form a trivial expression to redefine the expression object on the left whenever the button is clicked. For example, suppose the radius of the circle in the manipulative is 0.8 inches. Evaluating expression \( PI \times m0\{Radius\ Circle\ AB\} \times m0\{Radius\ Circle\ AB\} \) will give a result of 2.0106192982974678 with unit “inches” of order 2. The Interoperation Model forms a trivial expression \( 2.0106192982974678 \times (1\ inches^2) \) to redefine the expression object \( area \).

In this way, it is possible to have multiple buttons to update the same dynamic text field.

### 5.3 GeoSite's Page Authoring

GeoSite provides WYSIWYG user interface for authoring of

- Regular HTML
- Dynamic text, input boxes, and buttons introduced in section 5.2.1 for interoperation
- Mathematical relation between two expression objects that are from within a manipulative or are associated with a dynamic text or input box.

### 5.3.1 Authoring of Dynamic Text, Input Boxes, and Buttons

An HTML editor based on FCKEditor is provided for users to compose not only regular HTML, but also `<span>` elements for dynamic text, `<input>` elements for input boxes and buttons.

Figure 5.8 shows the WYSIWYG UI for authoring dynamic text. An author can give the name for the dynamic text. The name will be referenced later for the expression object backing the dynamic text.

![Figure 5.8 Authoring of dynamic text](image)

Figure 5.9 shows the WYSIWYG UI for authoring of input boxes. An author can give the name for the dynamic text. The name will be referenced later for the expression object backing the input box.
5.3.2 Authoring of Mathematical Relation

Figure 5.10 and Figure 5.11 show the authoring environment for defining mathematical relation between two components, either dynamic text or input boxes from HTML, or expression objects from within a manipulative.

The panel on the left shows different sections that a user will see for the page. The upper right panel lists all the components in the page, while the lower right panel has a calculator for an author to define the relation between two components.
Here is an HTML section. We can show here the radius of the circle in the manipulative.

The radius is 0.81 inches.
The area is 2.08 inches$^2$.

**Figure 5.10 Page Authoring Environment**

**Figure 5.11 Page Authoring Environment II**
CHAPTER 6
TOOLS, MACROS AND EXPRESSIONS

*Tool* is the term from user’s perspective. A tool is represented by a menu item or a toolbar button from the UI. A tool takes some input objects, applies a computation to them, and generates the data for output object(s).

A macro records a sequence of construction steps. Each step applies one tool to object(s) created in previous steps.

An expression can use a tool as a function. The operands of the functions are input objects for the tool, and function result is the output object of the tool. Each expression defines an expression object, which can be an input object of a tool.

An expression can also be one step in a macro, which takes objects created in previous steps as operands for the expression.

Macros and expressions play the core roles in GeometryEditor. This chapter will give the theoretical analysis of macros and expressions for GMAS to achieve the following goals

- Consolidation of tools with same semantics and being able to maintain this consolidation in macros and expressions
- Able to identify all the acceptable input object types, no wider or narrower for a macro or an expression
- Identical behavior between tools and user-defined macros
To the best of my knowledge, all existing geometry systems fail to achieve these goals to different extents. GeometryEditor is able to achieve these goals by

- Backing each tool by a system macro
- Running time rather than creation time analysis of the input object types of macros and expressions

6.1 Tools with Same Semantics

Different tools with the same semantics should be presented to the users as one tool, although in the background different computations are used depending on the types of the input objects. For example, *Measuring Area* is one tool from the user’s point of view regardless of the type of the object whose area is measured. In the background, different computation applies depending on whether the object being measured is a circle, a polygon, etc.

A GMAS must consolidate different tools with the same semantics as one tool under any circumstances:

- UI (menu and toolbar)
- Expressions
- Macros

CHAPTER 7 and CHAPTER 8 will introduce *Object Data Generator* (ODG) defined in GCML. Each ODG is the formal description of one computation of a tool. It describes the required input object type(s), the algebraic computation, and output object type(s).
6.2 Expressions

6.2.1 Syntax

Expressions are written in terms of operators and functions, which must have the correct number of operands, and correct type for each operand. Functions come from two sources:

- Built-in function such as abs, sqrt, iff, rgb, etc.
- Function mapped from a tool

Operands can be objects, literal values of number, bool, color, and text types, or result from a function.

Below is one example that generates a Color result type.

\[
\text{iff}(\text{distance}(A,BC)>1.5\text{inches},\text{rgb}(0,255,0),\text{rgb}(255,0,0))
\]

Symbols in the expression are:

- \text{iff} and \text{rgb} are built-in functions
- \text{A} and \text{BC} are the labels of two geometric objects
- \text{1.5inches}, 0, and 255 are literal values
- \text{distance} is a function mapped from the tool named “distance”

6.2.2 Functions Mapped from Tools

For each tool from the UI, there is a corresponding function that can be used in expressions. A tool may have multiple computations in the background. As shown in Figure 6.1, the tool “distance” has different computations to compute the distance depending on the types of the two objects.
The combination of the object types of the operands will be used to match which computation to be used. The operands will be supplied as the inputs for that matched computation.

### 6.3 Macros

Most of the interactive geometry systems support macros. A macro records a sequence of construction steps. Each step applies one tool or an expression to object(s) created in previous steps.

A group of related objects in a manipulative can be divided into two groups: objects that are used as the inputs of the macro, and objects that are outputs of the macro depending on inputs. Figure 6.2 shows the idea of macros.
Figure 6.2 Steps to Create a Circle through Three Points

We have gone through several steps to create a circle passing through three points. We want to save these steps into a macro so that next time we can simply click on three points, and the circle can be created immediately.

The pseudo-data for the macro are shown in Table 6-1. All the objects are indexed, and the symbol “$” followed by an index number refers to an indexed object. The macro expects three inputs.

<table>
<thead>
<tr>
<th>0: Point: as input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Point: as input</td>
</tr>
<tr>
<td>2: Point: as input</td>
</tr>
<tr>
<td>3: Segment: SegmentWithTwoPoints($0, $1)</td>
</tr>
<tr>
<td>4: Point: MidPoint($3)</td>
</tr>
<tr>
<td>5: Line: PerpendicularLine($4, $3)</td>
</tr>
<tr>
<td>6: Segment: SegmentWithTwoPoints($1, $2)</td>
</tr>
<tr>
<td>7: Point: MidPoint($6)</td>
</tr>
<tr>
<td>8: Line: perpendicularLine($7, $6)</td>
</tr>
<tr>
<td>9: Point: Intersect($5, $8)</td>
</tr>
<tr>
<td>10: Circle: CircleWithCenterAndOnePoint($9, $1)</td>
</tr>
</tbody>
</table>

Table 6-1 Pseudo-data for the macro that creates a circle through three points

A macro does not reduce the number of objects that need to create in order to create the target object(s). It simply performs those intermediate steps for you.
6.4 Problems in Existing GMAS

All the existing GMAS systems fail to provide identical behavior between system tools invoked from UI and user-defined macros. Identical behavior between system macros and user-defined macros needs to meet the following two requirements.

- The acceptable types of an input object should not be narrowed.
- The way how an input object is supplied should not change.

Let us use examples to explain these two requirements.

*The acceptable types of an input object should not be narrowed.*

 GeoGebra has the built-in mirror tool for users to reflect an object with respect to a line. The same built-in tool can mirror point, line, circle, polygon, etc as shown in Figure 6.3. If we use GeometryEditor’s tool signature idea, it can be described as

```plaintext
Reflect( AnyObjectThatCanBeReflected, Line )
```

![Figure 6.3 Translation in GeoGebra](image)
Objects on the left of the straight line AB are the original objects, and objects on the right are the mirrored objects. All the mirrored objects are created by the same tool. As shown in Figure 6.4, we chose the circle on the left and the line AB as the input objects in the macro, and the circle on the right as the output objects.

The macro we created can only be used to reflect circle as shown in the box on the top in Figure 6.3. The tool signature is

\texttt{Reflect( Circle, Line )}

The supported types of the first input object of the user-defined macro have been drastically narrowed down to only one type, \textit{Circle}.

SketchPad has better support than GeoGebra for most user-defined macros. But in some cases it still narrows down the allowed object types, such as user-defined macros for measuring distance. Among the existing GMAS, only GeometryEditor is able to fully meet the first requirement: The allowed types of an input object should not be narrowed.

\textit{The way an input object is supplied should not change.}

Input objects of a system tool from UI can be supplied in many ways
- By selecting an object from the canvas
- By providing an expression in a dialog
- By using a system default object

However when a user-defined macro is created, most GMAS fail to maintain the second and third ways.

For example, in SketchPad when we try to translate a circle, we have this pop up dialog shown in Figure 4.13 to enter the horizontal and vertical translation amount.

![Figure 6.5 SketchPad: Built-in tool to Create a Translation](image)

After the construction is done, we select the two circles to create a macro. We expect the user-defined macro to allow us to click an object, and a dialog will pop up for us to enter the translation values. But this macro created in SketchPad does not have the pop up dialog anymore and always use the same values we entered before to create the translated object.
So the system tool has the pop up dialog for us to enter the numeric properties, while the user-defined macro does not. This is very inconsistent and confusing.

Another typical example is that when you plot a point, you only need to provide the coordinates. The built-in tool can pick up the default coordinate system for you. But if you make it into a macro, you have to select a coordinate system every time you run this macro.

This inconsistent behavior exists in all GMAS systems that support macros.

6.5 GeometryEditor’s Approach for Macros and Expressions

GeometryEditor is able to achieve these goals by

- Backing each tool by a system macro
- Running time rather than creation time analysis of the input object types of macros

6.5.1 System Macros

In GeometryEditor, each tool that can be invoked from UI is backed by a System Macro. A System Macro has only one step. For example, the pseudo-data of the macro backing the tool CircleWithCenterAndOnePoint is shown in Table 6-2. It only records one step plus the information of the inputs.

| 0: Point: as input |
| 1: Point: as input |
| 2: Circle: CircleWithCenterAndOnePoint($0, $1) |

Table 6-2 System Macro for tool “CircleWithCenterAndOnePoint”

When a user invokes a tool from the UI, what is executed is actually the macro backing the tool. This is different from many existing geometry systems, which have separate modules to execute a tool and a macro as shown in Figure 6.6. This is the root
cause that brings inconsistency between tools provided by the system and user-defined macros.

Figure 6.6 Other GMAS: separate modules execute tools and macros

The approach GeometryEditor takes is to generate a system macro in the background for every tool. Executing a tool is equivalent to executing a system macro. By using the same module to run tools and macros as shown in Figure 6.7, GeometryEditor is the only GMAS that brings identical behavior between system tools and user-defined macros.

Figure 6.7 GeometryEditor: using the same module to execute tools and macros
6.5.2 Identification of Acceptable Input Object Types for an Expression

Functions and operators used in an expression place the restriction of what types of input objects can be used in the expression. A GMAS should be able to identify the correct set of acceptable input object types for an expression under any circumstances.

Table 6-3 shows the acceptable types for each input object for a few expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>obj1</th>
<th>obj2</th>
<th>obj3</th>
</tr>
</thead>
<tbody>
<tr>
<td>area(obj1)</td>
<td>Circle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>translate(obj1, obj2, obj3)</td>
<td>Point</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>area(translate(obj1, obj2, obj3))</td>
<td>Circle</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>distance(obj1, obj2)</td>
<td>Point</td>
<td>Point</td>
<td></td>
</tr>
<tr>
<td>distance(obj1, obj2)^2 - area(obj2)</td>
<td>Point</td>
<td>Circle</td>
<td></td>
</tr>
</tbody>
</table>

Table 6-3 Identification of acceptable input object types

Example 1:

In expression

\[
\text{translate( obj1, obj2, obj3 )},
\]

the second and third operands are for the horizontal and vertical translation offset. The object type of first operand “\(obj1\)” can be many geometric object types such as Point, Circle, Line, and Polygon. The tool “translate” is a consolidation of tools that have individual computations to translate different types of objects.

The system goes through all the computations to see if any computation can be applied to “\(obj1\)”.
When the expression becomes

\[ \text{area}( \text{translate}( \text{obj1, obj2, obj3} )) , \]

although \textit{Point} and \textit{Line} are legal types for the \textit{translate} function, the translation result will be illegal for the \textit{area} function. So the legal types for the operand “\textit{obj1}” are only \textit{Circle} and \textit{Polygon}.

\textit{Example 2:}

As shown in Table 6-3, the acceptable types of the input objects of expression

\[ \text{distance}( \text{obj1, obj2} ) \]

are four combination (\textit{Point, Point}), (\textit{Point, Segment}), (\textit{Point, Circle}), and (\textit{Circle, Circle}).

When the expression is changed to

\[ \text{distance(obj1, obj2)}^2 - \text{area(obj2)} \]

the function \textit{area} places more restriction on “\textit{obj2}”. So the acceptable types of the input objects are only (\textit{Point, Circle}) and (\textit{Circle, Circle}).

Existing geometry systems usually are able to identify the correct set of acceptable input object types when an expression is created from the UI. But when an expression is reused as one step in a macro, existing geometry systems often fail to maintain the correct set of the acceptable input object types. They only allow one acceptable type for each input object. For example, when the above expression is created for a circle, it cannot be reused for a polygon. We will have similar topic for macros in section 6.3, which will give detailed analysis.
6.5.3 Identification of Acceptable Input Object Types for a Macro

Assume we have three tools: “Translate”, “Distance” and “Area” can take the following combination of input object types:

- **Translate**: \((\text{Point, Number, Number}), (\text{Circle, Number, Number}), (\text{Segment, Number, Number}), (\text{Polygon, Number, Number})\)

- **Distance**: \((\text{Point, Point}), (\text{Point, Segment}), (\text{Point, Circle})\) and \((\text{Circle, Circle})\)

- **Area**: \((\text{Polygon}), (\text{Circle})\)

For each combination of the input object types, there is one computation to support the tool. We will use these tools in the following examples.

*Example 1: One step macro including system macros*

Table 6-4 shows an example of a one step macro. All the system macros are such examples too. We have four computations behind tool “Translate” that can compute the state of the resulting object for translation of \(\text{Point, Segment, Circle, and Polygon}\). The macro requires three input objects. The acceptable input object types of the one step macro are the same as the tool.

<table>
<thead>
<tr>
<th>Macro</th>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Object: as input</td>
<td>Point</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>1: Object: as input</td>
<td>Segment</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>2: Object: as input</td>
<td>Circle</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>3: Translate($0, $1, $2$)</td>
<td>Polygon</td>
<td>Number</td>
<td>Number</td>
</tr>
</tbody>
</table>

*Table 6-4 One step macro*
Example 2: Multi-step macro

Table 6-5 shows a two step macro, which adds one more step to the previous macro. Two valid combinations of the input object types in the previous macro are no longer valid.

<table>
<thead>
<tr>
<th>Macro</th>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Object: as input</td>
<td>Point</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>1: Object: as input</td>
<td>Segment</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>2: Object: as input</td>
<td>Circle</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>3: Translate( $0$, $1$, $2$ )</td>
<td>Polygon</td>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>4: Area( $3$ )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-5 Multi-step macro

For multi-step macros, we need to go through all the steps to determine the types of the input objects. A later step may reveal that an object type acceptable at an earlier step needs to be dropped.

Example 3: Macro with expression

Table 6-6 shows a macro with an expression as one step. The acceptable combination of input object types can only be (Point, Circle) and (Circle, Circle) even though tools Distance and Area individually can take more input object types.

<table>
<thead>
<tr>
<th>Macro</th>
<th>$0$</th>
<th>$1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Object: as input</td>
<td>Point</td>
<td>Point</td>
</tr>
<tr>
<td>1: Object: as input</td>
<td>Point</td>
<td>Segment</td>
</tr>
<tr>
<td>2: Expression: “distance($0$, $1$)^2 - area($1$)”</td>
<td>Point</td>
<td>Circle</td>
</tr>
<tr>
<td></td>
<td>Circle</td>
<td>Circle</td>
</tr>
<tr>
<td></td>
<td>Point</td>
<td>Polygon</td>
</tr>
</tbody>
</table>

Table 6-6 Macro with expression

An expression is like a multi-step macro condensed into one step. It has its own acceptable input object types. Steps using expressions together with steps using tools determine the acceptable input object types of a macro.
6.5.4 Running Time vs. Creation Time Analysis of Macros

During creation time of a user-defined macro, each input is tied to a specific type of object.

Most GMAS systems save the input types into the serialized macro data similar to the following pseudo-data using specific object types.

```
0: Circle: as input
1: Number: as input
2: Number: as input
3: Translate( $0, $1, $2 )
4: Area( $3 )
```

Table 6-7 Macro saved with specific input types

As shown in Table 6-7, the first input object is tied to a Circle type. The types of inputs of a macro are saved, and will be used as the only source to strictly validate the types of allowed input objects. This is the root cause that other GMAS systems fail to identify all the acceptable input object types.

A more general macro is shown in Table 6-8. All the input objects are not tied to any specific types.

```
0: Object: as input
1: Object: as input
2: Object: as input
3: Translate( $0, $1, $2 )
4: Area( $3 )
```

Table 6-8 Macro saved without using specific input types

The set of acceptable objects for a macro should be determined during running time instead of creation time of a macro. The algorithm is given in the next section.

6.5.5 Algorithm for Running Time Analysis of Input Object Types

The algorithm to determine the correct acceptable set of object types is a backtracking algorithm.
The steps of a macro are divided into two categories: inputs and tool applications. The steps are sorted in the following way: first input, and all the tools that can run after first input is determined; second input, and all the tools that can run after the first and second inputs are determined; and so on. The sorting result is shown in Table 6-9.

<table>
<thead>
<tr>
<th>Step 0: input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step i1: tool</td>
</tr>
<tr>
<td>Step i2: input</td>
</tr>
<tr>
<td>Step i3: tool</td>
</tr>
<tr>
<td>Step i4: input</td>
</tr>
<tr>
<td>Step n-1: tool</td>
</tr>
</tbody>
</table>

Table 6-9 Sorting of steps in a macro

Assume we have \( m \) system supported object types, and there are \( n \) steps in the macro.

- \( \text{types}[m] \): array of all the system supported object types
- \( \text{steps}[n] \): array of step information of a macro

The pseudo code of the algorithm is

```cpp
// types[m]: array of all the object types
// steps[n]: array of step information of a macro

// choices[n]: for input it is the choice of a type;
// for a tool, it is the choice of a computation
// currentTypes[n]: array of the current object type for each step

for ( int i=0; i<n; i++ )
    choices[i] = -1;
for ( int i=0; i>=0; ) {
    bool stepBack = false;
    if ( i == n ) {
        // found one valid combination of input types
        // choices is one solution
        print array choices
        i--;
    }
    ```
Table 6-10 Backtracking algorithm to indentify all the acceptable object types

GeometryEditor implements the above algorithm with other complications:

- An expression can be a step of a macro. Expressions have to determine their own acceptable input object types first. The algorithm for expressions to identify acceptable object types is similar.

- Combination matching of object types instead of permutation matching

- Composite type, object type that can contain other types (section 8.2.3)
CHAPTER 7

GEOMETRY CONSTRUCTION MARKUP LANGUAGE

AN INTRODUCTION

GCML is a markup language to define object types, and ODGs in a GMAS. It is also the language to save manipulative data and macro data.

7.1 The Goals of GCML

Intergeo Consortium is trying to define a file format aiming at the interchange of content between two interactive geometry systems [24]. GCML is designed with a different goal focusing on extensibility.

One of the major goals of GCML and GeometryEditor is to make components pluggable so that the implementation of the kernel can be small while support for new object types and new tools and computations can be extensible. Each computation for a tool is defined by an Object Data Generator (ODG).

Macro and expression are the main components in the GeometryEditor kernel. Various object types and ODGs described by GCML are plugged into the system.

7.1.1 Extensibility of Object Types

A GMAS may evolve to support more and more geometric object types, and even object types in other subjects such as physics. Each object type defines some core data that can determine an object’s state. From programming’s perspective, an object type is a class that implements a standard interface.
Without touching the kernel at all, the system is able to be extended to support new object types so that new ODGs can be added for the new types. For example, as of the writing, GeometryEditor does not support conic sections yet. Once the Object Type for the conic sections is defined, new ODGs can be added for conic sections. Similarly, an Object Type for spring in the physics domain can be provided to extend the system to support spring.

### 7.1.2 Extensibility of ODGs

An ODG can generate one or more output objects in terms of one or more input objects and parameters.

How many built-in ODGs should be provided by a GMAS? Should the system provide a built-in ODG to create a circle through three given points?

Scher [55] discussed whether many or few menu items should be supported. It advocated only supporting atomic construction tools. Any construction that can be achieved by multiple atomic tools should not be provided.

GeometryEditor takes a different approach. Whether many or few tools should be supported depends on the focus of a manipulative. If the focus is to let students know the steps of how to create a circle through three given points: two mid-points, two perpendicular lines, one intersection point, and then the circle, an ODG for creating circle through three given points should not be provided as a built-in ODG. But if the focus of the manipulative is to investigate further features of triangle, probably we want this built-in ODG.
A macro can partially play this role. However, a macro brings in lots of auxiliary objects that may obscure the construction.

We want a way to extend the authoring system to provide an ODG that can do the algebraic calculation directly to calculate the center position and radius of the circle without going through all the intermediate steps.

GCML is the descriptive language to add new ODGs easily.

### 7.1.3 Minimum Programming Effort to Add a New ODG

To define a new ODG, an interested user needs to provide some computation on how to generate some output object(s) in terms of some input object(s) and parameters. GCML tries to minimize the programming knowledge a user needs. For example, to calculate the center and radius of a circle named “circle” centered at a point named “center”, and through a point named “point”, the codes an interested developer can write are like below:

```java
    circle.x = center.x;
    circle.y = center.y;
    circle.r = System.dist(center.x, center.y, point.x, point.y);
```

The codes are very straightforward. With some simple programming training, a user is able to define a new ODG.

### 7.1.4 Consolidation of ODGs with Same Semantics

An ODG named “PointOnObject” has very clear semantics. It can create a point on a straight line, a circle, or any geometric object that can have a point on it. The algorithm for placing a point on different objects is different, but they should be represented by one semantic name. From the user’s point of view, there is only one menu
item or toolbar button from the UI. From the ODG developer’s point of view, multiple ODGs share the same name. An ODG represents one computation, while multiple ODGs with the same name represent a tool. Although ODGs share the same name, the system should be smart enough to apply the correct ODG to selected objects.

7.2 The Simplified Language Grammar

Table 7-1 describes the simplified grammar for GCML in a BNF like format. Each GCML file can save multiple ODGs, multiple macros, and one manipulative. The structure of ODGs, macro data and manipulative data are very similar. They all take some parameters and input objects and generate some output objects. ODG has other child elements to describe the algorithms, rules, and patterns. Next section will give a simple system with only three object types, and two ODGs. CHAPTER 8 will describe GCML in detail.

```xml
<geometry>:=<geometry>('<ODGs>'?('<macros>'?('<manipulative>'?'</manipulative>'))?)?</geometry>
<ODGs>:=<ODGs>'('<ODG>')?'</ODGs>'
<macros>:=<macros>'('<macro>')?'</macros>'
<manipulative>:=<manipulative>'('<construct>')?'</manipulative>'
<ODG>:=<ODG>'('<parameters>'?('<inputs>'?('<outputs>'?('<parameter_update>'?('<computation>'?('<interaction_rule>'?('<valid_inputs_rule>'?('<identical_rule>')?)?)?)?)?)?)?)?</ODG>'
<macro>:=<macro>'('<primitiveList>'?('<objectList>')?)?<construct>'</macro>'
<construct>:=<construct 'construct_attr'>'<parameters>'?('<inputs>'?('<outputs>')?)?</construct>'
<construct_attr>:=<ODG_name>
<parameters>:=<parameters>'('<primitiveList>')?'</parameters>'
<inputs>:=<inputs>'('<objectList>')?'</inputs>'
<outputs>:=<outputs>'('<objectList>')?'</outputs>'
<computation>:=<computation>'<CDATA>'</computation>'
```
Table 7-1 Simplified Grammar for GCML
7.3 A Simple Example

We will use a small geometry system to illustrate object types, ODGs, and manipulative and macro data.

Suppose a GMAS system supports object types for points, circles, and numeric result of measurements. It provides one ODG to create a circle centered at one point and through another point, and another ODG to measure the area of a circle. Users can use these two ODGs to create manipulative shown in Figure 7.1.

![Figure 7.1 A Simple Example of a Manipulative Using Two ODGs](image)

7.3.1 Object Types and ODGs

Figure 7.2 shows the object types in the system.

- Type *Point*: the x and y coordinates determine a point.
- Type *Circle*: the center’s coordinates, and the length of the radius determine a circle
- Type *Number*: we need to know its value, and also the unit and order the value is tied to.
Each object type corresponds to an XML schema fragment that provides the structure for describing manipulatives and macros in GCML.

For example, the XML schema for point type is:

```xml
<xs:element name="Point" type="PointType"/>
<xs:complexType name="PointType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Table 7-2 XML Schema for <Point>

All the objects share some common attributes that are described by another XML schema fragment. All the object types inherit from it.

```xml
<xs:complexType name="ObjectType">
  <xs:attribute name="id" type="xs:string"/>

  <!-- used by ODGs -->
  <xs:attribute name="multiplicity" type="xs:string"/>
  <xs:attribute name="index" type="xs:integer"/>
  <xs:attribute name="ref" type="xs:integer"/>
  <xs:attribute name="cname" type="xs:string"/>

  <!-- style information -->
  <xs:attribute name="color" type="xs:string"/>
  <xs:attribute name="label" type="xs:string"/>
  <xs:attribute name="width" type="xs:string"/>
  <xs:attribute name="visibility" type="xs:string"/>
  <xs:attribute name="manipulability" type="xs:string"/>
</xs:complexType>
```

Table 7-3 XML Schema for common attributes of object types

Figure 7.2 also demonstrates the essential information in an ODG.

- ODG *FreePoint*: it takes two parameters for the computation to generate a point as the output.
- ODG *CircleWithCenterAndOnePoint*: it takes two points as inputs for the computation to generate a circle as output.
- **ODG Area**: it takes one circle as input for the computation to generate a numeric value as output.

```
<?xml version="1.0" encoding="UTF-8"?>
<geometry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://wme.cs.kent.edu/geosite/gcml.xsd">
<geometry>
<ODGs>
  <ODG name="FreePoint">
    <parameters>
      <number cname="x" />
      <number cname="y" />
    </parameters>
    <outputs>
      <Point cname="point" />
    </outputs>
    <parameter_update><![CDATA[
      x = _point.x + _dx;
      y = _point.y + _dy;
    ]]> 
    <computation><![CDATA[
      point.x = x;
      point.y = y;
    ]]> 
  </ODG>
</ODGs>
</geometry>
```

**Figure 7.2 Object types and ODGs in the Simple Example**

The GCML file that describes the three ODGs is...
Table 7-4 GCML for ODGs

The XML schema fragment for the ODG element is:
Neither an object type nor an ODG is built into a GMAS’s kernel. They should be pluggable into the system. The Type Loader and ODG Loader modules of a GMAS load the supported object types and ODGs during run time.

### 7.3.2 Manipulative Data

GCML is also the language to store manipulative data. Figure 7.3 shows the flow to create the manipulative shown in Figure 7.1

#### Figure 7.3 Construction Flow in the Simple Example

Objects in a manipulative are indexed for reference by ODGs. The information saved on the left in Figure 7.3 records the dynamic behavior of the manipulative. The information saved on the right Figure 7.3 records the static snapshot of the manipulative. The GCML file for the manipulative is:
Table 7-6 GCML for manipulatives
7.3.3 Macro Data

A macro records a sequence of construction steps. The data to save a macro is very similar to saving a manipulative. The difference is that it starts with some object elements.

A macro can be generated from the manipulative in the simple example.

Macro: given two points, we can create a circle and a measurement of the circle’s area.

The two points are the inputs of the macro, and the circle and the measurement are the output of the macro as shown in Figure 7.4.

![Diagram of macro inputs and outputs]

Figure 7.4 Macro Generated in the Simple Version

The GCML file for the macro is:

```xml
<xml version="1.0" encoding="UTF-8"/>
<geometry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://wme.cs.kent.edu/geosite/gcml.xsd">
  <macros>
    <macro>
      <Point index="0" />
      <Point index="1" />
      <construct ODG="CircleWithCenterAndOnePoint">
        <inputs>
          <Point cname="center" ref="0" />
          <Point cname="point" ref="1" />
        </inputs>
      </construct>
    </macro>
  </macros>
</geometry>
```
Table 7-7 GCML for Macro Data

Is this macro general enough? We have tied the inputs objects to \textit{Point} type. Assume we have another two ODGs using the same name \textit{“CircleWithCenterAndOnePoint”} as shown in

<table>
<thead>
<tr>
<th>CircleWithCenterAndOnePoint</th>
<th>CircleWithCenterAndOnePoint</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>inputs</strong></td>
<td><strong>inputs</strong></td>
</tr>
<tr>
<td>center: Circle point: Point</td>
<td>center: Segment point: Point</td>
</tr>
<tr>
<td><strong>parameters</strong></td>
<td><strong>parameters</strong></td>
</tr>
<tr>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>outputs</strong></td>
<td><strong>outputs</strong></td>
</tr>
<tr>
<td>circle: Circle</td>
<td>circle: Circle</td>
</tr>
<tr>
<td><strong>computation</strong></td>
<td><strong>computation</strong></td>
</tr>
<tr>
<td>circle.x = center.x; circle.y = center.y; circle.r = System.dist(center.x, center.y, point.x, point.y);</td>
<td>circle.x = segment.x+segment.cos_alpha<em>segment.maxT/2; circle.y = segment.y+segment.sin_alpha</em>segment.maxT/2; circle.r = System.dist(circle.x, circle.y, point.x, point.y);</td>
</tr>
</tbody>
</table>

**Figure 7.5 Another Two ODGs named “CircleWithCenterAndOnePoint”**

As shown in the “inputs” and “computation” blocks, the ODG on the left uses the position of an existing circle’s center for the center of the new circle while the ODG on the right uses the position of the middle of the segment for the center of the new circle.
Figure 7.6 shows the constructions by the three *CircleWithCenterAndOnePoint* ODGs. This reveals our macro shown in Figure 7.4 is not general enough. It cannot take a circle or a segment as the first input. It does not anticipate the evolvement of the system in the future as new ODGs may be plugged into the system.

![Figure 7.6 Constructions by Three CircleWithCenterAndOnePoint ODGs](image)

The GeometryEditor’s kernel internally interprets the macro in a more general way:

*Macro: given two objects, use ODGs CircleWithCenterAndOnePoint and Area to create two new objects.*

Figure 7.7 shows the more general macro interpreted by the system. Neither the inputs nor the outputs were tied to specific object types. During the run time, the system will analyze all the loaded ODGs, and maximize the range of the acceptable input types, and create the correct output types correspondingly. CHAPTER 6 already gave the theoretical analysis for macros to find out the correct set of acceptable input objects.
Figure 7.7 Internal Interpretation of the Macro Generated in the Simple Example

For readability, the macro is still saved as shown in Table 7-7. But internally, the system does not limit the input object types to the hard-coded types in the macro data. The system uses the algorithm given in section 6.5.5 to find out all the acceptable input object types.
CHAPTER 8

GEOMETRY CONSTRUCTION MARKUP LANGUAGE

THE LANGUAGE AND IMPLEMENTATION

CHAPTER 7 gives a brief introduction to GCML. This chapter will introduce the detail of the language and its implementation.

8.1 Unit Systems

8.1.1 System Units

The unit of angle can be in radians or in degrees. The System Angle Unit is the angle unit that is used for any angle-related computation result.

The unit of length can be in pixels, cm, inches, or even a user-defined unit length. The System Length Unit is the length unit that is used for any length-related computation result.

With the evolvement of the system, more units useful in physics can be introduced.

8.1.2 Origin of the Canvas

In contrast to real math coordinate system, the origin of the canvas is the top-left corner, and value on the vertical direction increases from top to bottom.
8.2 Object Types and Core Data for an Object’s State

8.2.1 Primitive Types (Non-Object Types)

Primitive (Non-Object) data types are not referring to any type of geometric objects, or the result of expression objects. They are used in GCML for core data of an object type (next two sections), and parameters of an ODG (section 8.4.2). The first letter of a Non-Object Type is always in lower case to distinguish from Object Types.

- **number**: either integer or floating point numbers
- **boolean**: true/false
- **color**: in the format of #rrggbb where each character can be a hexadecimal digit
- **text**: a string
- **numberList**: a list of elements of number type in csv format
- **textList**: a list of elements of text type in csv format

8.2.2 Two Categories of Object Types

There are two categories of object types: geometric object types (section 8.2.3) and non-geometric object types (section 8.2.4).

Objects of non-geometric types are the result of an expression object such as

- Number result: \(radius([\text{Circle #1}])+3\text{cm}\)
- Boolean result: \(isParallel([\text{Line #1}], [\text{Line #2}])\)
- Color result: \(rgb(0,0,\text{len}([\text{Segment AC}]/[\text{Segment AB}]*255)\)
- Text result: \(iff(\text{len}([\text{Segment AB}])>\text{len}([\text{Segment CD}]), "greater than", "less than or equal to")\)
Functions used in the expressions come from two sources:

- Built-in function such as `rgb` and `iff`
- ODGs that have results of non-geometric types such as `radius`, `isParallel`, and `len`.

GeometryEditor allows an ODG that has geometric object output(s) to be used as a function in expressions. But in this chapter, we will limit the ODGs to those that have non-geometric object output(s) in the context of expressions.

### 8.2.3 Geometric Object Types

So far GCML defines the following types: `Point`, `Segment`, `Ray`, `Line`, `Axis`, `Circle`, `Arc`, `ArcSector`, `ArcSegment`, `Polygon`, `Straight`, and `CoordinateSystem`. Types are not mutual exclusive. For example, `Segment`, `Ray`, `Line`, and `Axis` are all one kind of `Straight`. We call types that can contain other types as `Composite Types`. There is also another `Composite Type` “Object” that stands for any types.

Each geometric type has defined its own core data, each of which is of a `Primitive Type`. Once the values of the core data have been calculated, the state of a geometric object can be uniquely determined.

Below list the major geometric object types.

- **Point**
  - `x, y: (number)` coordinates of the point

```xml
<x:s:complexType name="PointType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<x:s:element name="Point" type="PointType"/>
```
• **Circle**
  - *x, y: (number)* coordinates of the center
  - *r: (number)* radius

```xml
<xs:complexType name="CircleType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
      <xs:attribute name="r" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

• **Straight (Segment/Ray/Line/Axis)**
  - A straight line is in parametric format
    \[
    x' = x + \cos\alpha t \\
    y' = y + \sin\alpha t
    \]
  - *x, y: (number)* coordinates the line passes
  - *\cos\alpha, \sin\alpha: (number)* cosine and sine values of the angle between a horizontal line and the subject line
  - *minT, maxT: (number)* the minimum and maximum the parameter *t* can have.
    For a Line type, minT is negative infinity, and maxT is infinity. For a Ray type, minT is 0, and maxT is infinity.
  - *ratioBase: (number)* used in the creation of point on a straight line

```xml
<xs:complexType name="StraightType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
      <xs:attribute name="cos\alpha" type="xs:decimal"/>
      <xs:attribute name="sin\alpha" type="xs:decimal"/>
      <xs:attribute name="minT" type="xs:decimal"/>
      <xs:attribute name="maxT" type="xs:decimal"/>
      <xs:attribute name="ratioBase" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
• **Arc:**
  - \( x_1, y_1 \) *(number)* coordinates of the starting position
  - \( x_2, y_2 \) *(number)* coordinates of the ending position
  - \( r \) *(number)* radius of the arc
  - \( x, y \) *(number)* coordinates of the center of the arc
  - \( \text{angle} \) *(number)* a signed angle formed by \((x_1, y_1), (x, y), (x_2, y_2)\)
  - \( \text{posAngle} \) *(number)* positive angle formed by \((x_1, y_1), (x, y), (x_2, y_2)\)
  - \( \text{posSweep} \) *(boolean)* whether the arc is traversed counter-clockwise
  - \( \text{largeArc} \) *(boolean)* whether to use the arc larger than a semi-circle

• **Polygon**
  - \( vX \) *(array of values of number type)* an array of abscissa of all the vertices
  - \( vY \) *(array of values of number type)* an array of ordinate of all the vertices
Core data are what an ODG can access and mutate with an object. Each ODG’s computation can access an object’s core data using the syntax: `object_name.core_data_name`. An ODG’s computation can also mutate the core data of output objects. Suppose we have a `Circle` type object named “circle”. By the following assignment, we can uniquely determine a circle’s position and radius.

```plaintext
circle.x = center.x;
circle.y = center.y;
circle.r = System.dist(center.x, center.y, point.x, point.y);
```

When a numeric literal is used in the computation of an ODG, the numeric literal is evaluated against the system units. So `circle.r = 200` is treated as 200 pixels if the system length unit is set to pixels. In most cases, a computation should be unit independent. For example, `circle.x = center.x` does not care about what system length unit is. It simply uses the x value of the point named “center” as the x value for the circle named “circle”.

### 8.2.4 Non-Geometric Object Types

There are four non-geometric object types: `Number`, `Color`, `Boolean`, and `Text`. Non-geometric object types are actually supplied by the Reusable Implementation Layer instead of the GeometryEditor’s kernel. But how to render them visually is determined by the GeometryEditor’s core. Similar to geometric types, each non-geometric type has defined its own core data.
- **Number**
  - **value**: *(number)* the quantity part
  - **unit**: *(text)* the unit can be length ("pixels", "inches", and "cm"), angle ("radians", and "degrees") or none
  - **order**: *(number)* the degree of the unit; for example, the measurement of an area has the result of the unit of length with the order of 2
  - **units**: *(textList)* units and orders are needed when the result contains multiple units. For example, the famous mc\(^2\) has result of units: “kg”, “meters”, and “seconds” with corresponding orders 1, 2, and -2.
  - **orders**: *(numberList)*: as explained above.

```xml
<xs:complexType name="NumberType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:decimal"/>
      <xs:attribute name="unit" type="xs:string"/>
      <xs:attribute name="order" type="xs:decimal"/>
      <xs:attribute name="units" type="xs:string"/>
      <xs:attribute name="orders" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Number" type="NumberType"/>
```

- **Boolean**
  - **value**: *(boolean)* can only be true or false

```xml
<xs:complexType name="BooleanType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Boolean" type="BooleanType"/>
```

- **Color**
  - **r**: *(number)* red component of the color
- \( g: \text{(number)} \) green component of the color

- \( b: \text{(number)} \) blue component of the color

```
<xs:complexType name="ColorType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="r" type="xs:decimal"/>
      <xs:attribute name="g" type="xs:decimal"/>
      <xs:attribute name="b" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

- **Text**

  - \( \text{value: (text)} \) the text string

```
<xs:complexType name="TextType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### 8.3 ODG Instance

An ODG *Instance* is one usage of an ODG, that is, we apply an ODG to the given input object(s) and the given values of parameters to create the output object(s).
A manipulative is composed of a sequence of ODG instances.

The core data of an object can uniquely determine an object’s state while the state of an ODG instance is determined by the input objects, and parameters. For example, a point on a circle can be determined by a given circle and cosine and sine values of the angle as shown in Figure 8.2. The given circle is the input object, while the cosine and sine values of the angle are the two parameters for the ODG instance.
Figure 8.2 Angle to Determine the Point on a Circle

The difference between inputs and parameters is:

- An input is an object, either a geometric object, or a non-geometric object from an expression result. A parameter is of primitive (non-object) data types:
  - `number`: 3
  - `boolean`: true/false
  - `color`: #ff0000
  - `text`: “abcd”

- An ODG instance cannot change the state of the inputs, but it may change the value of a parameter. For example, a point on a circle is semi-free. The value of the parameters “cos_angle” and “sin_angle” get changed when a user moves the point on the circle.

  When a parameter is of `number` type, it is evaluated against the system unit.
8.4 Element <ODG> and the Child Elements

The element <ODG> defines an ODG. An ODG requires one or more inputs and parameters, and generates one or more outputs. The input and output objects can be any geometric or non-geometric object types. The parameters must be primitive types.

Each <ODG> element has a <computation> child element, which defines the algorithm of how to generate the output objects based on input objects and parameters.

Table 8-1 shows an ODG named PointOnObject. Given one circle, and two number literals, the ODG can create a point on the circle. The later sections will explain each element inside the <ODG> element.

```
<ODG name="PointOnObject">
  <parameters>
    <number cname="cos_angle" />
    <number cname="sin_angle" />
  </parameters>
  <inputs>
    <Circle cname="circle" />
  </inputs>
  <outputs>
    <Point cname="point" />
  </outputs>
  <parameter_update><![CDATA[
    var tx = _point.x + _dx;
    var ty = _point.y + _dy;
    var d = System.dist( tx, ty, circle.x, circle.y );

    if ( d < 1e-2 ) {
      var angle = 2*System.PI*System.random();
      cos_angle = System.cos(angle);
      sin_angle = System.sin(angle);
    } else {
      cos_angle = (tx - circle.x) / d;
      sin_angle = (ty - circle.y) / d;
    }
  ]]>></parameter_update>
  <computation><![CDATA[
    point.x = circle.x + circle.r * cos_angle;
    point.y = circle.y + circle.r * sin_angle;
  ]]>></computation>
</ODG>
```

Table 8-1 GCML ODG “PointOnObject”
8.4.1 Element <inputs>

An ODG may have one or more inputs. Inside the <ODG> element, there is one and only one <inputs> element, which can have one or more child elements.

```
<inputs>
  <Circle cname="circle" />
</inputs>
```

**Table 8-2 GCML Element <inputs>**

Child elements of <inputs> can be any object types defined in the XML schema sections 8.2.3 and 8.2.4. Each object type element has an attribute `cname`. Attribute `cname` gives the name of the input object. It will be used as reference to the input object in the <computation> element.

8.4.2 Element <parameters>

An ODG may have one or more parameters.

```
<parameters>
  <number cname="cos_angle" />
  <number cname="sin_angle" />
</parameters>
```

**Table 8-3 GCML Element <parameters>**

A <parameter> element has two attributes: `type` and `cname`. The attribute `type` defines the type of the parameter. A parameter must be a primitive type. The attribute `cname` gives the name of the parameter. It will be used as reference to the parameter in the <computation> element.
8.4.3 Element <outputs>

An ODG must have one or more outputs. Inside the <ODG> element, there is one and only one <outputs> element, which can have one or more child elements.

```
<outputs>
  <Point cname="point" />  
</outputs>
```

Table 8-4 GCML Element <outputs>

Child elements of <outputs> can be any object types defined in the XML schema sections 8.2.3 and 8.2.4. Each object type element has an attribute `cname`. The attribute `cname` gives the name of the output object. It will be used as reference to the output object in the <computation> element.

8.4.4 Element <computation>

The element <computation> defines the algorithms to calculate the state of output objects in terms of input objects and parameters. The syntax is in JavaScript wrapped in a CDATA section.

```
<computation><![CDATA[
  point.x = circle.x + circle.r * cos_angle;
  point.y = circle.y + circle.r * sin_angle;
]]></computation>
```

Table 8-5 GCML Element <computation>

Syntax `object_cname.core_data_name`, where `object_cname` is the `cname` specified in the child elements of the <inputs> and <outputs> elements, and `core_data_name` is introduced in section 8.2, is used to

- access the core data of input or output objects
- assign new values to the core data of output objects
There are a couple of pre-defined functions that can be used in a computation, for example, `System.dist(x1, y1, x2, y2)` can calculate the distance between coordinates \((x1, y1)\), and \((x2, y2)\) with result in the system length unit.

Existence of output objects

The existence of the output objects depends on the state of the input objects and the values of the parameters. The syntax to set the existence of an output object in the computation is `object_cname.exist = true/false`. Below is an example:

### 8.4.5 Element `<parameter_update>`

An ODG may have some output object that has some degree of freedom. The freedom is controlled by the parameters of the ODG. On one hand, with specific values of parameters, an output object is completely determined; on the other hand, the output object can still be moved under some restriction set by the input object(s) while changing the values of the parameters.

A point on a circle is a typical example. Its movement is limited on the circle, and would change the values of parameters `cos_angle`, and `sin_angle`.

```xml
<parameter_update><![CDATA[
  var tx = _point.x + _dx;
  var ty = _point.y + _dy;
  var d = System.dist(tx, ty, circle.x, circle.y);
  if ( d < 1e-2 ) {
    var angle = 2*System.PI*System.random();
    cos_angle = System.cos(angle);
    sin_angle = System.sin(angle);
  }
  else {
    cos_angle = (tx - circle.x) / d;
    sin_angle = (ty - circle.y) / d;
  }
]]>
</parameter_update>
```
Table 8-6 GCML Element <parameter_update>

Similar to the <computation> element, the contents in a <parameter_update> element is also a piece of JavaScript codes wrapped in a CDATA section.

There are a few pre-defined variables used in the example: _point, _dx, and _dy.

- For each output object, before any user’s interaction, the system makes a copy of it as backup. It can be referenced as underscore plus the cname of the output object.

- For mouse movement
  - _dx, and _dy record the mouse cursor position horizontal and vertical offset.
  - _mx and _my record the current mouse cursor position

Now we are able to explain the algorithm in the <parameter_update>.

![Diagram showing the calculation of mouse movement](image)

Figure 8.3 Pre-defined Variables for <parameter_update>

Mouse drags point A around. The highlighted point is where point A originally is, and the square is where point A should be moved to. The mouse movement is from point A to the “new mouse position” whose coordinates is (_point.x+_dx, _point.y+_dy). In the algorithm, d calculates the distance between the center and the new mouse position.
Then $\cos \angle$ and $\sin \angle$ can be calculated by dividing the two sides of the larger triangle by $d$.

The system calls the algorithm in `<parameter_update>` first to update the parameters, and then calls the algorithm in `<computation>`, which calculates the coordinates of point A’s new position.

### 8.4.6 Element `<interaction_rule>`

`<interaction_rule>` tells the system how the movement of an object should affect its parents. It is easy to determine how the movement of an object affects its children. We simply do the computation again to get the new values or new positions. It is difficult for the other way around.

`<interaction_rule>` can have the following child elements to specify the rules.

- `<replace_by_all_parents />`

  Selection and movement of an object is equivalent to selection and movement of all of its parents

  Example: mid-point of a segment

- `<replace_by_parent>cname_of_input_object</replace_by_parent>`

  Selection and movement of an object is equivalent to selection and movement of its parent(s) listed

  Example: parallel line passing through one point and parallel to a given line. Moving it is equivalent to moving the point it goes through.

- `<replace_by_parents_if_any_parent_selected />`
Selection and movement of an object is equivalent to selection and movement of all of its parents if any parent is selected together with the object.

Example: point on a circle

8.4.7 Element <valid_inputs_rule>

<valid_inputs_rule> can have the following child elements:

- <allow_duplicate_inputs />: whether to allow two input objects to be the same.
- <number_of_inputs_range />: the allowed number of inputs. It has the attributes min and max.
- <even_number_of_inputs />: requiring even number of inputs.

8.4.8 Element <identical_rule>

<identical_rule> gives the criteria to check whether two objects should be considered as the same object. For example, there is always only one line that is parallel to a given line and passes through a given point. If a user tries to create it a second time, the system should be able to use the existing one. Most GMAS packages are not able to do this checking.

<identical_rule> can have the following child elements:

- <identical_if_inputs_in_same_order />

Two ODG instances of the same ODG would have the same output object(s) if all the input object(s) are supplied in the same order.

Example: Circle with center and one point

- <identical_if_inputs_in_same_combination />
Two ODG instances of the same ODG would have the same output object(s) if all the input object(s) are supplied in the same combination, regardless of the order.

Example: Segment connecting two points

### 8.4.9 ODGs without Inputs but with Only Parameters

An ODG can have only parameters, but do not have any input object. In this case, the mouse position usually determines the values of the parameters, and hence determines the state of the output objects. ODG `<FreePoint>` is one example:

```xml
<ODG name="FreePoint">
  <parameters>
    <number cname="x" />
    <number cname="y" />
  </parameters>
  <outputs>
    <Point cname="point" />
  </outputs>
  <parameter_update><![CDATA[
    x = _point.x + _dx;
    y = _point.y + _dy;
  ]]>></parameter_update>
  <computation><![CDATA[
    point.x = x;
    point.y = y;
  ]]>></computation>
</ODG>
```

Table 8-7 GCML ODG “FreePoint”

### 8.4.10 ODGs That Have More Than One Output Objects

An ODG may generate more than one output objects. Here is an example, `CircleThroughThreePoints` that can generate the circle going through three given points, and also the center of the circle.

```xml
<ODG name="CircleThroughThreePoints">
  <inputs>
    <Point cname="point1" />
    <Point cname="point2" />
    <Point cname="point3" />
  </inputs>
</ODG>
```
Keep in mind that any ODG is a one step construction. With given input objects
and parameter values, there are no auxiliary objects created in the middle in order to
create the output objects. This "CircleThroughThreePoints" ODG is different from the
process in section 2.2.1 to create the circle going through three points. The process
shown in section 2.2.1 needs to create a couple of auxiliary objects (midpoints, and
perpendicular lines) to get to the center and the circle. The steps in section 2.2.1 can be
saved as a macro (section 8.8), which also allows one step creation of the circle and the
center. But inherently, a macro simulating one step construction is different from an
ODG.

Table 8-8 GCML ODG “CircleThroughThreePoints”

<table>
<thead>
<tr>
<th>Table 8-8 GCML ODG “CircleThroughThreePoints”</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Keep in mind that any ODG is a one step construction. With given input objects and parameter values, there are no auxiliary objects created in the middle in order to create the output objects. This “CircleThroughThreePoints” ODG is different from the process in section 2.2.1 to create the circle going through three points. The process shown in section 2.2.1 needs to create a couple of auxiliary objects (midpoints, and perpendicular lines) to get to the center and the circle. The steps in section 2.2.1 can be saved as a macro (section 8.8), which also allows one step creation of the circle and the center. But inherently, a macro simulating one step construction is different from an ODG.</td>
<td></td>
</tr>
</tbody>
</table>
8.4.11 ODGs with Indefinite Number of Input Objects

A typical example is an ODG for creating a polygon. The variable with the name referenced by the attribute *cname* is treated as an array in JavaScript. The value of the attribute *multiplicity* will be referenced as a variable in the JavaScript for the size of the array.

```xml
<ODG name="Polygon">
  <inputs>
    <Point multiplicity="n" cname="vertices" />
  </inputs>
  <outputs>
    <Polygon cname="polygon" />
  </outputs>
  <computation><![CDATA[
    for ( var i=0; i<n; i++ ) {
      vX[i] = vertices[i].x;
      vY[i] = vertices[i].y;
    }
  ]]>></computation>
</ODG>
```

Table 8-9 GCML ODG “Polygon”

As shown in the example, *cname*’s value *vertices* is treated as an array, while the attribute *multiplicity*’s value *n* is treated as the size of the array.

8.4.12 ODG Expression

The ODG named “Expression” is the only ODG for which the system provides special support.

```xml
<ODG name="Expression">
  <parameters>
    <text cname="expression" />
  </parameters>
  <inputs>
    <Object multiplicity="n" cname="operands" />
  </inputs>
  <outputs>
    <output type="Object" cname="result" />
  </outputs>
  <computation><![CDATA[
    // the computation is provided by the system’s native support
  ]]>></computation>
</ODG>
```
The ODG takes indefinite number of input objects, and also one parameter for the expression. It has one output object named “result” that can be any non-geometric types. The system provides native codes for the <computation> part.

The value for the parameter “expression” can be any valid expression of various complexities.

8.4.13 ODG with Indefinite Number of Outputs

When two geometric objects intersect, the number of intersection points may differ.

- 1: two straight lines
- 2: two circles, or one circle and one straight line
- 4: two conic sections

Section 8.5 will talk about multiple ODGs sharing the same name. Multiple ODGs sharing the same name “Intersect” generate the intersection points described above. Section 8.5.5 points out the restriction on ODGs sharing the same name: they have to have the same number of outputs. We cannot have ODGs that have 1 output, 2 outputs, and 4 outputs respectively to share the same name. The attribute *multiplicity* for the output element is introduced to solve this problem.
<Point cname="points" multiplicity="n" />
</outputs>
<computation><![CDATA[
var d1 = System.distSquare( circle1.x,circle1.y,circle2.x,circle2.y );
vary = circle1.r;
var r2 = circle2.r;
if ( r1+r2 < d2 || System.abs( r2-r1 ) > d2 )
{
    points[0].exist = false;
    points[1].exist = false;
    return;
}
var tmp = r1*r1 + d1 - r2*r2;
var cos_alpha = tmp/2/r1/d2;
var sin_alpha = System.sqrt( 1 - cos_alpha*cos_alpha );
var cos_beta = (circle2.x-circle1.x)/d2;
var sin_beta = (circle2.y-circle1.y)/d2;
var cos_angle0 = cos_beta*cos_alpha - sin_beta*sin_alpha;
var sin_angle0 = sin_beta*cos_alpha + cos_beta*sin_alpha;
var cos_angle1 = cos_beta*cos_alpha + sin_beta*sin_alpha;
var sin_angle1 = sin_beta*cos_alpha - cos_beta*sin_alpha;
points[0].x = circle1.x + cirlce1.r * cos_angle0;
points[0].y = circle1.y + circle1.r * sin_angle0;
points[1].x = circle1.x + cirlce1.r * cos_angle1;
points[1].y = circle1.y + cirlce1.r * sin_angle1;

n = 2;
]]></computation>

Table 8-11 GCML ODG “Intersect”

8.5 ODGs Having the Same Name

8.5.1 Overview

It is very common to use the same semantic name for two ODGs that apply to different types of input objects. For example:

- *PointOnObject* for a point on a line, a point on a circle, etc.
- *Translate* for translating any geometric object with some horizontal and vertical amounts.
• *Area* for measuring area of a circle, area of a polygon, area of an arc sector, etc.

• *Distance* for distance between two points, distance between a point and a circle, etc.

• *IsIntersect* for checking whether two lines intersect, whether two circles intersect, whether a circle and a line intersect, etc.

GCML allows different ODGs to have the same name. Each ODG provides one computation. From the user’s point of view, there is only one menu item or toolbar button to represent the <ODGs> with the same semantic name. It is not user friendly to have one menu item for “Point On Circle” ODG, and another menu item for “Point on Line” ODG.

### 8.5.2 Example ODGs *PointOnObject*

We have already given one ODG “PointOnObject” in Table 8-1 that creates a point on a circle. Below is another ODG with the same name that can create a point on a straight line.

```xml
<ODG name="PointOnObject">
  <inputs>
    <Straight cname="line" />  
  </inputs>
  <parameters>
    <number cname="ratio" />
  </parameters>
  <outputs>
    <Point cname="point" />  
  </outputs>
  <parameter_update><![CDATA[
    if ( line.ratioBase < 1e-2 ) {
      ratio = 0.2+0.6*System.random();
      return;
    }
    var tx = _point.x+_dx;
    var ty = _point.y+_dy;
    var t = (ty-line.y)*line.sin_alpha + (tx-line.x)*line.cos_alpha;
    ratio = System.max( System.min( line.maxT, t ),
                       line.minT )/line.ratioBase; )))>
  </parameter_update>
  <computation><![CDATA[
    var t = line.ratioBase * ratio;
  ]]>
</ODG>
```
\[
\text{point.x} = \text{line.x} + t \times \text{line.cos_alpha}; \\
\text{point.y} = \text{line.y} + t \times \text{line.sin_alpha};
\]

Table 8-12 Second GCML ODG “PointOnObject”

Two ODGs that have the same names must have at least one different input type; otherwise the system is confused about which ODG should be used for given object(s).

8.5.3 Example ODGs IsIntersect

We will have another example IsIntersect. Table 8-13 is the ODG that returns a Boolean type object telling whether two circles intersect.

Table 8-13 GCML ODG “IsIntersect”

Table 8-14 is another ODG with the same name “IsIntersect” that tells whether one straight line and a circle intersect.
Table 8-14 Second GCML ODG “IsIntersect”

8.5.4 Flexibility in Defining ODGs Having the Same Name

GCML allows a certain degree of flexibility to define two ODGs that have the same names. They do not interfere with each other in the following aspects

Different cname of input objects

The ODG “PointOnObject” in Table 8-1 has an input with $cname$ “circle”, while the one in Table 8-12 has an input with $cname$ “line”. By allowing different $cnames$ for input objects, the algebraic computations in ODGs are more readable.
Different types of output objects

ODGs “Translate” can translate different types of geometric objects. The output type is the same as that of the input geometric object. For example, translating a point will have an output type Point.

```xml
<ODG name="Translate">
  <inputs>
    <Point cname="srcPoint" />
    <Number cname="horizontal" />
    <Number cname="vertical" />
  </inputs>
  <outputs>
    <Point cname="point" />
  </outputs>
  <computation><![CDATA[
    point.x = srcPoint.x + horizontal.value;
    point.y = srcPoint.y - vertical.value;
  ]]>]
</computation>
</ODG>
```

Table 8-15 GCML ODG “Translate”

Translating a circle will have an output type Circle.

```xml
<ODG name="Translate">
  <inputs>
    <Circle cname="srcCircle" />
    <Number cname="horizontal" />
    <Number cname="vertical" />
  </inputs>
  <outputs>
    <Circle cname="circle" />
  </outputs>
  <computation><![CDATA[
    circle.r = srcCircle.r;
    circle.x = srcCircle.x + horizontal.value;
    circle.y = srcCircle.y - vertical.value;
  ]]>]
</computation>
</ODG>
```

Table 8-16 Second GCML ODG “Translate”
Different cnames for the output

As seen from the previous “Translate” example, the *cname* for the output is different for two ODGs named “Translate”, which increases the readability of the ODG descriptor.

8.5.5 Restrictions on ODGs Having the Same Name

There is restriction on ODGs that have the same name.

- Same number of input objects
- Same number of output objects

The restriction is imposed by macros to be introduced in section 8.8. ODGs having the same name will be mapped to only one macro. If they do not have the same number of input objects or output objects, it would bring confusion to the macro. We need some structural symmetry in ODGs so that they can be represented by the same macro.

8.5.6 User Friendly in Implementation of GCML

An implementation of GCML should be user friendly enough to tell the users what types of objects are being expected at a certain step depending on what ODGs have been loaded.

For example, if only the first *IsIntersect* ODG in Table 8-13 and Table 8-14 is loaded, only *Circle* type is allowed for the first step. GeometryEditor’s Tool Signature window shows:
If both \textit{IsIntersect} ODGs are loaded, more object types are allowed for the first step. GeometryEditor’s Tool Signature window shows:

\begin{itemize}
\item \texttt{IsIntersect ( Object, Object )}
\item \texttt{Expecting: Circle}
\end{itemize}

\begin{figure}
\centering
\includegraphics[width=0.4\textwidth]{tool_signature_1}
\caption{Tool Signature When Only One “IsIntersect” ODG is Loaded}
\end{figure}

An implementation of GCML should allow ODGs to be dynamically loaded. Section 8.6 will show how users can load a new ODG in the middle of authoring. If a new ODG with a pre-existing name is loaded, the signature window should reflect the change immediately.

\section{8.6 Run Time Loading of ODG}

Without refreshing the running environment or making any configuration change, an author should be able to load an ODG during running time by

\begin{itemize}
\item uploading a GCML file describing an ODG
\item pasting into an input area
\item providing a URL of a GCML file
\end{itemize}
Figure 8.6 demonstrates in GeometryEditor a user loads an ODG from an input area.

After the ODG is loaded, the user can access it from the UI.

8.7 Manipulative Data

A GCML file for saving data contains a sequence of <construct> elements. <construct> represents an ODG instance.
8.7.1 Elements <construct>

Suppose we have created a manipulative in the following steps, three free points, and then the ODG \textit{CircleThroughThreePoints}, which generates a circle and its center, and then a point on the circle.

![Manipulative with Circle through Three Points and a Point On it](image)

The Object List window shows the index of each object.

Figure 8.9 shows the dependency relation between ODG instances and their output objects. Objects are indexed. The indices are used by ODG as reference to input objects.
Figure 8.9 Dependency among ODG instances and Objects

The piece of GCML to save the manipulative is as below:

```xml
<construct ODG="FreePoint">
  <parameters>
    <number cname="x" value="300" />
    <number cname="y" value="200" />
  </parameters>
  <outputs>
    <Point cname="point" index="0"
      x="..." y="..." />
  </outputs>
</construct>

<construct ODG="FreePoint">
  <parameters>
    <parameter cname="x" value="250" />
    <parameter cname="y" value="400" />
  </parameters>
  <outputs>
    <Point cname="point" index="1"
      x="..." y="..." />
  </outputs>
</construct>
```
The element `<construct>` has a very similar structure as the element `<ODG>`. It represents an ODG instance, so all the detailed information needed by an ODG is supplied.

- `<construct>` has an attribute `ODG` to associate itself with a `<ODG>`
- The child elements of `<inputs>` have an attribute `ref` to specify which object is used for this input
• The child elements of `<parameters>` have an attribute `value` to specify the value

• The child elements of `<outputs>` have an attribute `index` to index all the generated objects in a manipulative.

  `<construct>` records the dynamic information of a manipulative. `<outputs>` records a snapshot of the state of all the output objects.

  If two ODGs share the same name, the types of the input object(s) can uniquely determine which ODG should be used. In the above example, the name “PointOnObject” is used in two ODGs shown in Table 8-1 and Table 8-12. The input object is a circle, so the ODG should be the one in Table 8-1.

8.7.2 Attributes Color, Visible, Manipulable, and Label for Child Elements of `<outputs>`

The computation in an ODG computes the core data of an object. Other than core data, there are many common parameters for objects, such as width, color, visibility, manipulability, and label.

An output element has five corresponding attributes: width, color, visible, manipulable, and label to describe these parameters.

• Attribute `width`: number type, the width of the stroke to draw a geometric object

• Attribute `color`: in the `rgb` format in hexadecimal like `#ffeee`, or an integer that is the index of an expression object that has result of `Color` type.

• Attribute `visible`: `true/false`, or an integer that is the index of an expression object that has result of `Boolean` type.
• Attribute *manipulable*: true/false, or an integer that is the index of an expression object that has result of *Color* type.

• Attribute *label*: a text string, or an integer that is the index of an expression object that has result of *Text* type.

Figure 8.10 gives an example of a triangle that has its color controlled by an expression. When a user moves around any of the three points on the segments, the expression’s result gets updated. So does the color of the triangle.

![Figure 8.10 Object color determined by an expression](image)

8.8 Macro Data

A macro records a sequence of construction steps. The data format describing a macro is similar to that describing a manipulative, but with some difference.

8.8.1 Starting with Object Elements

When saving a manipulative, GCML always starts with a <construct> element because without using any ODG instance, there is no object created. When saving
macros it is different. It always starts with object elements. But they act like dummy placeholders to let the system know how many objects a user should select to complete a macro process.

For example, below shows a macro that is mapped from the ODG “CircleThroughThreePoints” in Table 8-8.

```
<Point index="0" />
<Point index="1" />
<Point index="2" />

<construct ODG="CircleThroughThreePoints" index="0">
  <inputs>
    <input cname="point1" ref="0" />
    <input cname="point2" ref="1" />
    <input cname="point3" ref="2" />
  </inputs>
  <outputs>
    <Point cname="center" index="3" />
    <Circle cname="circle" index="4" />
  </outputs>
</construct>
```

Table 8-18 GCML for Macro “CircleThroughThreePoints”

It starts with three object elements instead of any <construct> element. It expects a user to select three objects to have this macro go through. Three objects with indices 0, 1, and 2 are *inputs of the macro*, and two objects with indices 3 and 4 are *outputs of the macro*.

Compared with the above single step macro, Table 8-19 demonstrates another macro that represents the steps of how to create a circle through three points shown in Figure 2.6.

```
<Point index="0" />
<Point index="1" />
<Point index="2" />

<construct ODG="SegmentWithTwoPoints" index="0">
  <inputs>
    <input cname="point1" ref="0" />
    <input cname="point2" ref="1" />
  </inputs>
</construct>
```
<outputs>
  <Segment cname="segment" index="3" />
</outputs>
</construct>

<construct ODG="MidPoint" index="1">
  <inputs>
    <input cname="segment" ref="3" />
  </inputs>
  <outputs>
    <Point cname="point" index="4" />
  </outputs>
</construct>

<construct ODG="PerpendicularLine" index="2">
  <inputs>
    <input cname="point" ref="4" />
    <input cname="refLine" ref="3" />
  </inputs>
  <outputs>
    <Line cname="line" index="5" />
  </outputs>
</construct>

<construct ODG="SegmentWithTwoPoints" index="3">
  <inputs>
    <input cname="point1" ref="1" />
    <input cname="point2" ref="2" />
  </inputs>
  <outputs>
    <Segment cname="segment" index="6" />
  </outputs>
</construct>

<construct ODG="MidPoint" index="4">
  <inputs>
    <input cname="segment" ref="6" />
  </inputs>
  <outputs>
    <Point cname="point" index="7" />
  </outputs>
</construct>

<construct ODG="PerpendicularLine" index="5">
  <inputs>
    <input cname="point" ref="7" />
    <input cname="refLine" ref="6" />
  </inputs>
  <outputs>
    <Line cname="line" index="8" />
  </outputs>
</construct>

<construct ODG="Intersect" index="6">
  <inputs>
    <input cname="line1" ref="5" />
    <input cname="line2" ref="8" />
  </inputs>
  <outputs>
8.8.2 System Macros: Macro Mapped from ODGs with the Same Name

We have introduced the concept of System Macro in 6.5.1. In the language of GCML, all ODGs with the same name are automatically mapped to a system macro. Unlike general macros that record a sequence of construction steps, it only records one step of construction. Table 8-20 shows a system macro with one output, while Table 8-21 shows a system macro with two outputs.
8.8.3 Macros Mapped from an ODG That Has Output Object of Non-geometric Type

Let us consider an ODG whose output object is of non-geometric type. We use IsIntersect in Table 8-14 as the example. As shown in Figure 8.11, when ODG IsIntersect is applied to two objects, Segment #0 and Circle #0, an object of Boolean type is created.
According to the previous section, a macro as below mapped from the ODG should be generated.

```xml
<Circle index="0" />
<Circle index="1" />
<construct ODG="IsIntersect" index="0">
  <inputs>
    <Circle cname="Circle" ref="0" />
    <Circle cname="Circle" ref="1" />
  </inputs>
  <outputs>
    <Boolean cname="result" index="2" />
  </outputs>
</construct>
```

Table 8-23 Macro Mapped from ODG “IsIntersect”

The system actually converts the above macro into a macro using the Expression ODG (section 8.4.12) instead, as shown below.

```xml
<Circle index="0" />
<Circle index="1" />
<construct ODG="Expression" index="0">
  <inputs>
    <Circle cname="operands" cindex="0" ref="0" />
    <Circle cname="operands" cindex="1" ref="1" />
  </inputs>
  <parameters>
```

Figure 8.11 Non-geometric Type Output
The ODG *IsIntersect* becomes part of the parameter *expression*. Actually any ODG can be mapped to a macro using the *Expression* ODG. But by default, only those that have non-geometric type output objects are mapped to a macro using *Expression* ODG.

In GeometryEditor, an output object created by a macro using the *Expression* ODG can be modified in the calculator dialog (section 3.6.2). The modification is reflected on both the input objects and the parameter *expression* of the ODG instance.
CHAPTER 9

REUSABLE IMPLEMENTATION LAYER FOR WEB-BASED MANIPULATIVE AUTHORING OR CUSTOMIZATION

GeometryEditor is built upon a layer that is reusable for building any Web-based Manipulative Authoring or Customization System (WMACS).

9.1 Web-based Manipulative Authoring or Customization Systems

A WMACS can be as complicated as GeometryEditor, which provides sophisticated authoring support. MathEdit [59] is another manipulative authoring system hosted on GeoSite. Users can author mathematics formulae. Figure 9.1 shows the MathEdit authoring environment. Figure 9.2 shows a manipulative created by MathEdit in learning mode.

![Figure 9.1 MathEdit in Authoring Mode](image)

164
A WMACS can also focus on customization instead of authoring. It can be as simple as the example shown in Figure 9.3. An author can customize the number of the rows and columns of the grid by clicking the “Change Dimension” button. The manipulative in the learning mode does not have that button shown up.
9.2 Overview of the Reusable Implementation Layer

The Reusable Implementation Layer (RIL), which GeometryEditor is built on, provides a base class for developing a WMACS. A WMACS needs to inherit from the base class, and implement a few methods.

- Method `create()` to create HTML fragment for manipulative instantiation
- Method `update()` to update a manipulative for new data
- Getter `getPropertyNam()` to return the list of fields for serialization
- Getter `getProperty()` to return the value of a particular field for serialization
- Method `onGetData()` if some pre-processing needs to be done for serialization

The Reusable Implementation Layer provides the implementation for the following commonly needed support:

- Manipulative instantiation and management
  - Dynamic creation and destroy of a manipulative instance
  - Management of multiple manipulative instances
  - Callback after manipulative initialization
  - Dynamic loading of data from a URL and callback
  - Synchronization between data loading and manipulative instance initialization
  - Dynamic loading of necessary JavaScript/CSS files and callback
- Serialization of a manipulative
- Dialogs management
- Expression Support
– Expression parsing algorithms support
– Manipulative interoperation interface support
– Listener registration for expression value change in a manipulative

The Reusable Implementation Layer can greatly reduce the time a developer spends in solving some common issues when developing a Web-based Manipulative Authoring System.

9.3 Manipulative Instantiation and Management

9.3.1 Manipulative Class

Table 9-1 shows the sample codes for the class of manipulative `PercentageTable` inheriting from RIL’s base class:

```javascript
function PercentageTable( config ) {
  // this.authoring is set from config.authoring
  // by the constructor of superclass
  PercentageTable.superclass.constructor.call( this, config );

  this.bgImgPath = "url(\"" + PercentageTable.toolBasePath + 'img/pizza.gif' + ")\";
  this.nRows = 10;
  this.nCols = 10;

  this.nHidden = 0;
  this.nShown = 0;
}
RIL.extend( PercentageTable, BaseTool );

PercentageTable.fullName = "xun.algebra.percentage.PercentageTable";
PercentageTable.version = "0.1";
PercentageTable.defaultData = {nRows: 10, nCols: 10};

//create HTML fragment for manipulative instantiation
PercentageTable.prototype.create = function( config ) {
  var divElement = document.getElementById(config.renderTo);

  // generate HTML fragment for element divElement based on config.data
  if ( this.authoring ) {
    // generate HTML fragment for authoring mode
    ........
  }
```
else {
    // generate HTML fragment for learning mode
    ....
}

// call the API from RIL
this.onCreate();

// update the manipulative for new data
PercentageTable.prototype.update = function( config ) {
    // update the manipulative based on config.data
    ....

    // call the API from RIL
    this.onUpdated();
};

// getter of the list of fields for serialization
PercentageTable.prototype.getPropertyNames = function() { return ["nRows", "nCols"];
};

// getter for a particular field for serialization
PercentageTable.prototype.getProperty = function( name ) {
    if ( name == "nRows" )
        return this.nRows;
    else if ( name == "nCols" )
        return this.nCols;
};

// Other PercentageTable specific methods
......

Table 9-1 Manipulative class inheriting from the RIL’s base class

9.3.2 Manipulative Instantiation

An instance of the PercentageTable can be created in a Web page using the following JavaScript codes

```javascript
<script type="text/javascript">
var m1 = new PercentageTable( {instanceName: "m1", authoring: false} );
m1.createManipulative({
    renderTo: "div1",
    url: "http://wme.cs.kent.edu/geosite/1x/GeometryEditor/server/getMan.php?id=_mnaACVzHTTEQ5U"
    // data: ...... // either url or data needs to be provided
});
</script>
```

Table 9-2 Codes for Manipulative Instantiation

The two arguments taken by the constructor have the following meanings:
• **instanceName**: a name for later reference to the manipulative. The RIL manages all the manipulative instances in an associate array, and provides an API to access the instance with a specified name

• **authoring**: indicating whether the UI should behave as an authoring environment. In the *PercentageTable* example, the “Change Dimension” button shows up if `authoring` is set to `true`.

Method `createManipulative` is supplied by the super class, which in turn calls the method `create` supplied by the manipulative class *PercentageTable*. The arguments taken by the method have the following meaning:

• **renderTo**: the id of the HTML container element to render the manipulative

• **url**: the url of the data for the manipulative instance. The RIL can access the url to retrieve the data that will be passed into `create`. A manipulative class’s `create()` method does not need to deal with any url.

• **data**: the data for the manipulative instance. Usually the data is saved into a Web site’s database by using *RIL*’s serialization APIs.

  Either `url` or `data` must be supplied. Similar discussion can apply to `updateManipulative()` from the RIL.

### 9.3.3 Synchronization

There are several synchronization issues the RIL takes care of.
Synchronization in Manipulative Instantiation

If the HTML fragment of a manipulative contains SVG, Flash or Java Applet content, the manipulative may not be completely loaded before create() returns. Developers of the manipulative class need to have codes to indentify the loading is done and make a call to RIL’s onCreated() method.

Synchronization in JavaScript files Loading

The RIL can load a default JavaScript file for each manipulative class. If a manipulative class needs to load more JavaScript files, it can call the API RIL.loadScript() with a callback function.

Synchronization in Data URL Loading

When a URL is provided in creation or update of a manipulative, the RIL takes care of the synchronization. The methods createManipulative() and updateManipulative() will not call create() or update() until the data from the url is available.

9.4 Serialization of a Manipulative

A manipulative class can implement the getPropertyNames() to return a list of fields for serialization. The RIL will call getProperty() for each of the fields afterwards. The getProperty() method is allowed to return not only primitive type values such as numeric values or strings, but also arrays or objects. The RIL is able to serialize a returned array or object into JSON string.
On the GeoSite, the serialized JSON string of a manipulative is saved to the database. When a page with the manipulative embedded is loaded, the data is retrieved from the database for the manipulative instantiation.

9.5 Dialogs Management

It is not trivial to write HTML and JavaScript to manage pop up dialogs:

- Populating data to widgets when opening a dialog
- Collecting data when closing a dialog
- Modality
- Cross browser compatibility
- Managing layers of multiple dialogs: opening a dialog from a dialog
- Simulating OK, Cancel, and Apply behaviors
- Equivalency of closing a dialog to Cancel behavior

The RIL has built fundamental support for all the above needs. A manipulative only needs to call the API below to open a dialog:

```javascript
RIL.openDialog( ManipulativeInstance, dialogName, dialogURL )
```

The argument `dialogURL` is the URL of a HTML page. It only needs to provide functions `init()`, `onOK()`, `onCancel()`, and `onApply()` if needed. JavaScript file `dlgCommon.js` is provided by the RIL.

The HTML source codes for the dialog page are shown in Table 9-3. The resulting dialog is shown in Figure 9.3.

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<script type="text/javascript" src="dlgCommon.js"></script>
<script type="text/javascript"></script>
</head>
```
Table 9-3 A dialog for a manipulative customization

9.6 Expression Support

9.6.1 Basic Expression Support

The RIL provides the following basic support for expressions:

- Support for Number, Boolean, Color, and Text object types
- Built-in functions support such as mathematical functions, string manipulation functions and so on
• Expression parsing and evaluation

9.6.2 Extensibility

Various manipulative systems have their own specific object types and functions. For example, GeometryEditor has the “Circle” object type and function “area” to measure the area of a circle. A manipulative system can implement a few methods for the RIL to query about object types and function names.

9.6.3 Expression in the Interoperation Model

As discussed in section 5.2.2, expression plays the core role in the interoperation among manipulatives and HTML fragments.

GeometryEditor can provide an ODG to output the equation of a circle. The ODG can be used as a function in expressions to generate a result of Text type.

MathEdit can take an expression to define a mathematics formula.

Figure 9.4 shows one example. The manipulative marked as m0 was generated by GeometryEditor while the manipulative marked as m1 was generated by MathEdit. We can easily define the interoperation between them. User’s interaction of the circle can immediately change the MathML rendered by MathEdit.
9.7 Converting a Manipulative System to Use the RIL

For many existing Web-based authoring systems or manipulatives developed by ad-hoc programming, it is possible to just wrap a layer around them to reuse the codes, and make it to follow the standards required by the RIL. In this way, the adapted manipulative system can be deployed on GeoSite so that GeoSite can be a repository of not only manipulatives and educational pages, but also manipulative authoring systems.
CONCLUSIONS, LIMITATIONS, AND FUTURE WORK

The dissertation has the following contributions: GeometryEditor is the first completely Web-based GMAS. It tries to utilize the Web to the greatest extend. Not only can manipulative authoring be done online, but also models and solutions for various Web orientation topics are investigated and proposed. In addition to basic authoring support that most existing systems have, GeometryEditor provides a few unique and very important authoring features. The dissertation gives the theoretical analysis of the relation among tools, macros and expressions. GeometryEditor points out the root cause of a few macro and expression problems that exist in other systems, and provides the solution. A markup language, GCML, is defined to provide the extensibility support. Users with some basic programming background are able to provide extension of authoring support to the system. GeometryEditor also provides a valuable Reusable Implementation Layer that can be used as the infrastructure for building Web-based manipulative authoring systems.

On the other hand, this work has the following limitation: the research on the integration of GMAS and CAS by the Web is still at the early stage. GCML requires all ODGs with the same semantics to have the same number of inputs. There is lack of GCML definition for some special authoring tools such as Synchronized Copy. The Reusable Implementation Layer is still not well-designed enough to become a framework.
Future work will include: Using GeometryEditor as the front end for a backend CAS. More tags will be developed in GCML to describe more rules and patterns identified in geometry manipulative authoring. Further research will be done on GCML to allow native SVG tags to be mixed within GCML. More refactoring and enhancement will be done the Reusable Implementation Layer to make it a true reusable framework and library. Also it is interesting to investigate how to migrate GeometryEditor to handheld devices and tablets.
APPENDIX A

GCML XML Schema

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!--
-- A GCML root element can have one ODGs child element,
-- one macros child elements, and one manipulative element
-- Notice here: ODGs and macros are plural nouns, but manipulative
-- is a singular noun because one file can only save at most one manipulative.
-->
<xs:element name="geometry">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ODGs" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="macros" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="manipulative" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- Group all the primitive types so that it can be
-- reused in both primitiveList complexType and macro element
-->
<xs:group name="primitiveGroup">
  <xs:choice>
    <xs:element ref="number"/>
    <xs:element ref="boolean"/>
    <xs:element ref="text"/>
    <xs:element ref="color"/>
  </xs:choice>
</xs:group>

<!-- A list of arbitrary number of primitive types in arbitrary order -->
<xs:complexType name="primitiveList">
  <xs:group ref="primitiveGroup" minOccurs="1" maxOccurs="unbounded"/>
</xs:complexType>

<!-- Group all the object types so that it can be
-- reused in both objectList complexType and macro element
-->
<xs:group name="objectGroup">
  <xs:choice>
    <xs:element ref="Object"/>
    <xs:element ref="Point"/>
    <xs:element ref="Straight"/>
    <xs:element ref="Line"/>
    <xs:element ref="Segment"/>
    <xs:element ref="Ray"/>
    <xs:element ref="Circle"/>
    <xs:element ref="Arc"/>
    <xs:element ref="ArcSegment"/>
    <xs:element ref="ArcSector"/>
  </xs:choice>
</xs:group>
<xs:element ref="Polygon" />
<xs:element ref="Number" />
<xs:element ref="Boolean" />
<xs:element ref="Color" />
<xs:element ref="Text" />
</xs:choice>
</xs:group>

<!-- A list of arbitrary number of object types in arbitrary order -->
<xs:complexType name="objectList">
  <xs:group ref="objectGroup" minOccurs="1" maxOccurs="unbounded"/>
</xs:complexType>

<!-- ODGs can have multiple ODG -->
<xs:element name="ODGs">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ODG" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- An ODG takes 0 to multiple parameters (of primitive types), and
-- 0 to multiple inputs (of object types)
-- to generate one or more outputs (of object types).
-- Rules for interaction, validation, and identical
-- criteria can be provided.
-- An ODG must have a name attribute.
-->
<xs:element name="ODG">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="parameters" type="primitiveList" minOccurs="0"/>
      <xs:element name="inputs" type="objectList" minOccurs="0"/>
      <xs:element name="outputs" type="objectList" />  
      <xs:element name="parameter_update" type="xs:string" minOccurs="0"/>
      <xs:element name="computation" type="xs:string"/>
      <xs:element ref="interaction_rule" minOccurs="0"/>
      <xs:element ref="valid_inputs_rule" minOccurs="0"/>
      <xs:element ref="identical_rule" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<!-- A manipulative element has multiple construct child element -->
<xs:element name="manipulative">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="construct" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- One macros element can have multiple macro elements -->
<xs:element name="macros">
  <xs:complexType>
    <xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
A macro takes 0 to multiple primitives, and 0 to multiple input objects, and apply a sequence of construct.

Below is an example:

```xml
<macro>
  <Point index="0"/>
  <Point index="1"/>
  <construct ODG="CircleWithCenterAndOnePoint">
    <inputs>
      <Point cname="center" ref="0"/>
      <Point cname="point" ref="1"/>
    </inputs>
    <outputs>
      <Circle cname="circle" index="2"/>
    </outputs>
  </construct>
  <construct ODG="Area">
    <inputs>
      <Circle cname="circle" ref="2"/>
    </inputs>
    <outputs>
      <Number cname="result" index="3"/>
    </outputs>
  </construct>
</macro>
```

Construct is one step to apply an ODG.

```xml
<construct name="construct">
  <sequence>
    <parameters type="primitiveList" minOccurs="0"/>
    <inputs type="objectList" minOccurs="0"/>
    <outputs type="objectList"/>
  </sequence>
  <attribute name="ODG" type="xs:string" use="required"/>
</construct>
```

Interaction_rule specifies the rule of how the movement
-- of an object affects its parents.
-- There are three cases defined so far:
-- 1.replace_by_all_parents: moving the intersection point of two circles
--     is equivalent to moving the two circles together
-- 2.replace_by_parent: moving a parallel line, which passes through a given
--     point and is parallel to another line, is equivalent to moving
--     the point
-- 3.replace_by_parents_if_any_parent_selected: this is for those semi-free
--     objects such as a point on a circle. If no parent is selected,
--     they maintain the semi-free behavior. If any one of its parents
--     is selected, it is equivalent to only moving the parents.

<x:s:element name="interaction_rule">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="replace_by_all_parents" />
      <xs:element ref="replace_by_parent" />
      <xs:element ref="replace_by_parents_if_any_parent_selected" />
    </xs:choice>
  </xs:complexType>
</xs:element>

<x:element name="replace_by_all_parents">
  <xs:complexType>
  </xs:complexType>
</xs:element>

<x:element name="replace_by_parent" type="xs:string" /> 

<x:element name="replace_by_parents_if_any_parent_selected">
  <xs:complexType>
  </xs:complexType>
</xs:element>

!-- Validation on the inputs. So far three validation rules are defined.
-- They are not mutual exclusive.
-- allow_duplicate_inputs: whether two inputs can be the same object
-- number_of_inputs_range: minimal and maximal of the number of input
--     objects.
-- even_number_of_inputs: whether the number of input objects should be
--     an even number.
-->
<x:element name="valid_inputs_rule">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="allow_duplicate_inputs" minOccurs="0"/>
      <xs:element ref="number_of_inputs_range" minOccurs="0"/>
      <xs:element ref="even_number_of_inputs" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<x:element name="allow_duplicate_inputs" type="xs:boolean" default="false" />

<x:element name="number_of_inputs_range">
  <xs:complexType>
    <xs:attribute name="min" type="xs:integer"/>
    <xs:attribute name="max" type="xs:integer"/>
Rule to detect whether an existing object should be used as the output object of an ODG instance.

Example: A segment connecting A and B is the same segment connecting B and A.

There are two rules defined so far:

- identical_if_inputs_in_same_order: example: ray
- identical_if_inputs_in_same_combination: segment

Four primitive types are defined.

- They can be used in <macro>, <ODG> and <construct> elements

- number
- boolean
- text
- color
Common attributes for an object
-- multiplicity is for indefinite number of inputs or outputs; its value
-- is used as the variable name for the size
-- color, label, width, visibility, manipulability can be determined
-- by an expression
-- index is for output object
-- ref is for input object

<xs:complexType name="ObjectType">
  <xs:attribute name="id" type="xs:string"/>
  <xs:attribute name="multiplicity" type="xs:string"/>
  <xs:attribute name="index" type="xs:integer"/>
  <xs:attribute name="ref" type="xs:integer"/>
  <xs:attribute name="cname" type="xs:string"/>
  <xs:attribute name="color" type="xs:string"/>
  <xs:attribute name="label" type="xs:string"/>
  <xs:attribute name="width" type="xs:string"/>
  <xs:attribute name="visibility" type="xs:string"/>
  <xs:attribute name="manipulability" type="xs:string"/>
</xs:complexType>

<xs:element name="Object" type="ObjectType"/>

-- The following geometric object types are defined
-- Point, Circle, Straight, Segment, Line, Ray,
-- Arc, ArcSegment, ArcSector, Polygon

<xs:complexType name="PointType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Point" type="PointType"/>

<xs:complexType name="CircleType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
      <xs:attribute name="r" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Circle" type="CircleType"/>

<xs:complexType name="StraightType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Straight" type="StraightType"/>
<xs:attribute name="x" type="xs:decimal"/>
<xs:attribute name="y" type="xs:decimal"/>
<xs:attribute name="cos_alpha" type="xs:decimal"/>
<xs:attribute name="sin_alpha" type="xs:decimal"/>
<xs:attribute name="minT" type="xs:decimal"/>
<xs:attribute name="maxT" type="xs:decimal"/>
<xs:attribute name="ratioBase" type="xs:decimal"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="Straight" type="StraightType"/>
<xs:element name="Line" type="StraightType"/>
<xs:element name="Ray" type="StraightType"/>
<xs:element name="Segment" type="StraightType"/>

<xs:complexType name="ArcType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="x" type="xs:decimal"/>
      <xs:attribute name="y" type="xs:decimal"/>
      <xs:attribute name="r" type="xs:decimal"/>
      <xs:attribute name="x1" type="xs:decimal"/>
      <xs:attribute name="y1" type="xs:decimal"/>
      <xs:attribute name="x2" type="xs:decimal"/>
      <xs:attribute name="y2" type="xs:decimal"/>
      <xs:attribute name="angle" type="xs:decimal"/>
      <xs:attribute name="posAngle" type="xs:decimal"/>
      <xs:attribute name="posSweep" type="xs:boolean"/>
      <xs:attribute name="largeArc" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Arc" type="ArcType"/>

<xs:complexType name="ArcSectorType">
  <xs:complexContent>
    <xs:extension base="ArcType">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="ArcSector" type="ArcSectorType"/>

<xs:complexType name="ArcSegmentType">
  <xs:complexContent>
    <xs:extension base="ArcType">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="ArcSegment" type="ArcSegmentType"/>

<xs:complexType name="PolygonType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="vX" type="xs:string"/>
      <xs:attribute name="vY" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
The following Non-Geometric Object types are defined:

- Number
- Color
- Boolean
- Text

```xml
<xs:complexType name="NumberType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:decimal"/>
      <xs:attribute name="unit" type="xs:string"/>
      <xs:attribute name="order" type="xs:decimal"/>
      <xs:attribute name="units" type="xs:string"/>
      <xs:attribute name="orders" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Number" type="NumberType"/>

<xs:complexType name="BooleanType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Boolean" type="BooleanType"/>

<xs:complexType name="TextType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="value" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Text" type="TextType"/>

<xs:complexType name="ColorType">
  <xs:complexContent>
    <xs:extension base="ObjectType">
      <xs:attribute name="r" type="xs:decimal"/>
      <xs:attribute name="g" type="xs:decimal"/>
      <xs:attribute name="b" type="xs:decimal"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Color" type="ColorType"/>
```

</xs:schema>
APPENDIX B
HOW EXPRESSIONS ARE PROCESSED

An expression is processed in four steps:

\[\text{findTokens} \rightarrow \text{infixToPostfix} \rightarrow \text{resolveTokensInContext} \rightarrow \text{evaluate}\]

The whole process is provided by the Reusable Implementation Layer (RIL). In the first two steps, the RIL resolves any tokens it recognizes such as “2inches” and “abs”. In step 3, a token resolver is passed in from the context to resolve tokens specific to the context, which can be GeometryEditor’s manipulative authoring environment or the educational page authoring environment.

The class Expression is defined as

```javascript
function Expression()
{
    this.expr = null;     // of type RNum/RBoolean/RColor
    this.result = null;
    this.tokens = [];     // infix tokens
    this.nTokens = 0;
    this.pTokens = [];    // prefix tokens
    this.nPTokens = 0;
    this.stack = new STACK();  // MathML namespace
    this.m = "m:"
}

// static fields
Expression.UNARY_NEG = "~";
Expression.ARG_TERMINAL = "@";
Expression.digits = "0123456789";
Expression.ops = {
    "|": 2,
    "&": 3,
    ">": 4,
    ">=": 4,
    "<": 4,
    "<=": 4,
    "<>": 4,
    "==": 4,
    "+": 5,
};
```
"-": 5,
"*": 6,
"\u00B7": 6,
"\u00D7": 6, // times
="/": 6,
"\%": 6,
"\_": 10,
"\!": 10,
"\^": 11
};

// static methods
Expression.SGN = function( op )
{
    return op>0?1:(op<0?-1:0);
};
Expression.IFF = function( op1, op2, op3 )
{
    return op1?op2:op3;
};
Expression.MAX = function( args, nArgs )
{
    var t = args[0], i;
    for ( i=1; i<nArgs; i++ )
        if ( t.getCvtVal() < args[i].getCvtVal() )
            t = args[i];
    return t;
};
Expression.MIN = function( args, nArgs )
{
    var t = args[0], i;
    for ( i=1; i<nArgs; i++ )
        if ( t.getCvtVal() > args[i].getCvtVal() )
            t = args[i];
    return t;
};
Expression.RED = function( op1 )
{
    return op1.r;
};
Expression.GREEN = function( op1 )
{
    return op1.g;
};
Expression.BLUE = function( op1 )
{
    return op1.b;
};
Expression.funcOps = {
    "ABS": Math.abs,
    "SIN": Math.sin,
    "COS": Math.cos,
    "TAN": Math.tan,
    "ASIN": Math.asin,
    "ACOS": Math.acos,
    "ATAN": Math.atan,
    "LN": Math.log,
    "EXP": Math.exp,
    "ROUND": Math.round,
    "FLOOR": Math.floor,
    "CEIL": Math.ceil,
"RANDOM": Math.random,
"SQRT": Math.sqrt,
"SGN": A.SGN,
"IFF": Expression.IFF,
"MAX": Expression.MAX,
"MIN": Expression.MIN,
"RED": Expression.RED,
"GREEN": Expression.GREEN,
"BLUE": Expression.BLUE
};
Expression.constants = { "PI": Math.PI,
    "E": Math.E
};
Expression.compOps = function( op1, op2 )
{
    return Expression.ops[op1] - Expression.ops[op2];
};
Expression.isOperator = function(t)
{
    return Expression.ops[t];
};
Expression.isDigit = function(t)
{
    return Expression.digits.indexOf(t)>=0;
};
Expression.isFunction = function(t)
{
    return Expression.funcOps[t];
};
Expression.isConstant = function(t)
{
    return Expression.constants[t];
};

// instance methods
Expression.prototype.more = function( token )
{
    if ( token.length == 0 )
        return;
    var t, tl, re = /^([-+]?\d*\.?\d+)/,
        re2 = /^(cm|inches|pixels|degrees|radians|\]|\))/;
    if ( re.test( token ) )
    {
        tl = RegExp.$1;
        t = token.substr( tl.length );
        if ( t != "" && !re2.test(t) )
        {
            this.addToken( tl );
            this.addToken( "*" );
            this.addToken( t );
            return;
        }
    }
    this.addToken( token, true );
};
Expression.prototype.addToken = function( token )
An expression is parsed and evaluated in the following steps. We will use the following expression as an example:

\[ len(AB)+2\text{inches} \]

where \( AB \) is a segment.

**findTokens**

findTokens decomposes an expression into tokens. The tokens are stored in an array in the same order as they appear in the expression.

For expression \( len(AB)+2\text{inches} \), the tokens will be

<table>
<thead>
<tr>
<th>Len</th>
<th>(</th>
<th>AB</th>
<th>)</th>
<th>+</th>
<th>2inches</th>
</tr>
</thead>
</table>

```javascript
Expression.prototype.findTokens = function() {
    var i, n, nBraces=0, c, cNext, token="", prevToken, b=Expression, t=this.tokens, tl;

    var a=this;
    a.nTokens = 0;
    n = a.expr.length;

    for ( i=0; i<n; ) {
        prevToken = "";
        c = a.expr.charAt(i);

        switch (c) {
            case "(";
            case ")":
                // Token is a parenthesis
            case "^":
            case "+":
            case "/":
            case \\":
```
case "]":  
    a.more( token );
    token = "";
    a.addToken( c );
    break;

// consume one more

infixToPostfix

infixToPostfix turns the tokens into postfix order, and also translates numeric, color, Boolean, and text constants into objects of RNum, RColor, RBoolean, and RText classes, which all inherit from Result class. These are object types recognized by the
Reusable Implementation Layer (RIL). Up to this point, we do not translate any geometry-related tokens so that the expression module can be re-used in many contexts.

For expression \( \text{len}(AB)+2\text{inches} \), the tokens in postfix will be

<table>
<thead>
<tr>
<th>@</th>
<th>AB</th>
<th>LEN</th>
<th>RNum object with 2inches as content</th>
<th>+</th>
</tr>
</thead>
</table>

```javascript
Expression.prototype.infixToPostfix = function()
{
  var i, t, s=this.stack, p=this.pTokens, prev, top, r, n=0;
  var re = /^([-+]?\d*\.?\d+)(cm|inches|pixels|degrees|radians)?$/;
  var re2 = /^#[0123456789abcdef]{6}$/;
  s.clear();
  for ( i=0; i<this.nTokens; i++ )
  {
    t = this.tokens[i];
    switch( t )
    {
      case "(":
        r = s.top();
        if ( typeof r == "string" &&
            Expression.isFunction(r.toUpperCase())
        )
          p[n++] = Expression.ARG_TERMINAL;
        s.push(t);
        break;
      case ")":
        while ( !s.empty() )
        {
          t = s.pop();
          if ( t != "(" )
            p[n++] = t;
          else
            break;
        }
        if ( t != "(" )
          throw exception;
        break;
      case ",":
        while ( !s.empty() )
        {
          t = s.pop();
          if ( t != "(" )
            p[n++] = t;
          else
            break;
        }
        if ( t != "(" )
          throw exception;
        break;
      case ":";
      case "+";
      case "-";
      case "*":
        prev = i>0?this.tokens[i-1]:null;
        if ( ( prev == null || Expression.isOperator(prev) || prev == "(" || prev == "," )
        {
```
t = Expression.UNARY_NEG;
}
case "!":
case "^":
case "*":
case "/":
case "<<":
case ">":
case ">=":
case ">=":
case ":=":
case ":=":
case ":=":
case ":=":
case ":=":
case ":=":  
  top = s.empty()?null:s.top();
  if ( s.empty() || top == "(" )
    s.push( t );
  else if (Expression.isOperator(top) &&
      Expression.compOps(t, top ) > 0 )
    s.push( t );
  else
  {  
    while ( !s.empty() )
    {
      top = s.top();
      if ( top == "(" )
        break;
      else if (Expression.isOperator(top) )
      {
        if ( (top == Expression.UNARY_NEG || top == '!' )
            && (t == Expression.UNARY_NEG
                 || t == '!') ) // solves 4---5
          break;
        else if (Expression.compOps( top, t )<0 )
          // for other operators with same
          precedence, pop the one in the stack
          break;
      }
      p[n++] = s.pop();
    }
    s.push( t );
  }
  break;
default:
  if ( Expression.isFunction( t.toUpperCase() ) )
    s.push( t.toUpperCase() );
  else if ( Expression.isConstant( t.toUpperCase() ) )
  {  
    r = new RNum() based on t
    s.push( r );
  }
  else if ( v.isVariable(t) )
    s.push( t );
  else if ( /^(true|false)$/.test(t.toLowerCase()) )
  {  
    r = new RBoolean() based on t;
    s.push( r );
  }
  else
{  
  if ( re.test(t.toLowerCase()) )  
  {  
    r = new RNum() based on t  
    s.push( r );  
  }  
  else if ( re2.test(t.toLowerCase()) )  
  {  
    r = new RColor() based on t  
    s.push( r );  
  }  
  else {  
    // push in any unresolved token  
    s.push( r );  
  }  
  break;  
}  
while ( !s.empty() )  
{  
  p[n++] = s.pop();  
  this.nPTokens = n;  
  return true;  
}

resolveTokensInContext

At this step, different context passes into different tokenResolver. GeometryEditor passes in a resolver that can understand the name of ODGs, and the names of geometric objects or expression objects. Page authoring environment passes in a resolver that can understand tokens referring to expression objects in a manipulative, dynamic texts or inputs.

A token resolver returns a TokenInfo object. It can tell whether this token is an operator or an operand.

When resolveTokensInContext goes through all the tokens, it deals with only the types of operands instead of their values. It validates whether types of inputs match an operator’s requirement.
For expression \( \text{len}(AB)+2\text{inches} \), the tokens after running \( \text{resolveTokensInContext} \) will be

| @ | TokenInfo object with 1.a field pointing to segment object AB 2.a flag indicating this is an operand | TokenInfo object with 1.a field pointing to the correct ODG named LEN 2.a flag indicating this is an operator (function) | RNum object with 2\text{inches} as content | + |

```
Expression.prototype.resolveTokensInContext = function( tokenResolver )
{
    var i, t, s=this.stack;
    var tokens = this.pTokens;
    var n = this.nPTokens;

    for ( i=0; i<n; i++ )
    {
        t = tokens[i];
        if ( t belongs to Expression.ops or Expression.funcOps ) {
            pop the correct number of operands from the stack
            push in the result type
        } else if ( t instanceof RNum/RColor/RBoolean/RText ) {
            push in the result type
        } else {
            ask tokenResolver for information of t;
            TokenInfo tokenInfo = tokenResolver(t);
            // replace the token string by TokenInfo object
            tokens[i] = tokenInfo;

            if ( tokenInfo.isFunction() ) {
                pop the correct number of operands from the stack
                // TokenInfo.dryExecute finds the correct ODG based on
                // the types of the operands if multiple ODGs share the
                // same name
                call tokenInfo.dryExecute();
                push in the result type
            } else {  // tokenInfo is an operand
                push the type tokenInfo.getType() into the stack
            }
        }
    }
}
```
r = s.pop();    // the type of the result of the expression
return r;
}

**Evaluate**

*Evaluate* has very similar algorithm as *resolveTokensInContext*. The only difference is *evaluate* deals with real values while *resolveTokensInContext* deals with types.

The previous three steps are called only once during the processing of an expression. *Evaluate* is called repeatedly during a user interaction.

```javascript
Expression.prototype.evaluate = function()
{
    var i, t, s=this.stack;
    var tokens = this.pTokens;
    var n = this.nPTokens;

    for ( i=0; i<n; i++ )
    {
        t = tokens[i];
        if ( t belongs to Expression.ops or Expression.funcOps ) {
            pop the correct number of operands from the stack
            push in the result object
        }
        else if ( t instanceof RNum/RColor/RBoolean/RText ) {
            push in the result object
        }
        else {  // t is an operand
            TokenInfo tokenInfo = t;
            if ( tokenInfo.isFunction() ) {
                pop the correct number of operands from the stack
                call tokenInfo.execute();
                push in the result object
            }
            else {  // tokenInfo is an operand
                push the type tokenInfo.gerResult() into the stack
            }
        }
    }

    r = s.pop();    // the type of the result of the expression
```
    return r;
};
REFERENCES

[1] Algebra Tiles manipulative,
   http://www.coe.tamu.edu/~strader/Mathematics/Algebra/AlgebraTiles/AlgebraTiles1.html.


http://www.ct4me.net/math_manipulatives.htm.


[22] The Geometer’s Sketchpad, [Computer software],


    187.

[26] A. P. Jahn, "Locus" and "Trace" in Cabri-géomètre: relationships between geometric
    and functional aspects in a study of transformations, International Reviews on


[28] Major Differences between JavaSketchpad and Desktop Sketchpad,

[29] Joyce, D.E, The Geometry Applet
    http://aleph0.clarku.edu/~djoyce/java/Geometry/Geometry.html.


[41] Logo Philosophy and Implementation,


[44] Mathmetical Markup Language (MathML) 1.01 Specification,


[64] XBL-Extensible Binding Language 1.0 Specification,

[65] XML User Language (XUL) 1.0 Specification,
