VECTORPAD: A TOOL FOR VISUALIZING VECTOR OPERATIONS

by

JARED BOTT

B.S., Computer Science, University of Central Florida, 2005

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term 2009

Major Professor: Joseph J. LaViola Jr.

ABSTRACT

Visualization of three-dimensional vector operations can be very helpful in understanding vector mathematics. However, creating these visualizations using traditional WIMP interfaces can be a troublesome exercise. In this thesis, we present VectorPad, a pen-based application for three-dimensional vector mathematics visualization. VectorPad allows users to define vectors and perform mathematical operations upon them through the recognition of handwritten mathematics. The VectorPad user interface consists of a sketching area, where the user can write vector definitions and other mathematics, and a 3D graph for visualization. After recognition, vectors are visualized dynamically on the graph, which can be manipulated by the user. A variety of mathematical operations can be performed, such as addition, subtraction, scalar multiplication, and cross product. Animations show how operations work on the vectors. We also performed a short, informal user study evaluating the user interface and visualizations of VectorPad. VectorPad's visualizations were generally well liked; results from the study show a need to provide a more comprehensive set of visualization tools as well as refinement to some of the animations.

ACKNOWLEDGMENTS

I would like to thank my wife, Maria, for her kindness, patience, and love as I slowly made progress on my thesis. I would also like to thank my father Kevin and my mother Barbara for helping me throughout my numerous years of college. Thanks also go out to my brothers and to my friends.

Furthermore, I would like to thank my faculty advisor, Dr. Joseph LaViola, for his guidance and support. I would like to thank the members of my committee, Dr. Charles Hughes, and Dr. Hassan Foroosh for their direction and guidance. Finally, I would like to thank the past and present members of the Interactive Systems and User Experience lab for their help.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

VectorPad is a pen-based application for visualizing three-dimensional vector mathematics. Inputting 3D vector mathematics when using traditional WIMP-based (Window, Icon, Menu, Pointing device) [25] techniques can be confusing and typically requires special knowledge. VectorPad uses handwriting and mathematics recognition technology to allow users to input vectors and vector mathematics easily using a simple stylus. These mathematical equations are then solved using a mathematical engine, and the results are dynamically displayed on a 3D graph. Users can edit their mathematical equations and see how these changes affect the mathematical operations.

SECTION 1.1 Motivation

Current tools for visualizing 3D vector mathematics are often troublesome and unintuitive. Often, they focus on support for one or two mathematical operations and do not allow for complicated mathematical equations. Vector mathematics

operations can be tricky for some people to understand. The ability to visually explore these operations might give users valuable insight into their nature. Our goal is to provide a post-WIMP [9, 31] application that provides the user with a simple way to visualize 3D vector equations for numerous mathematical operations. We wanted to take advantage of tablet PCs and other touch screen technologies, which are becoming increasingly popular and allow for what we consider to be a more natural mathematical interaction than specialized mathematical input languages. We think that a tool for vector visualization, such as VectorPad, might be useful for linear mathematics education, as well as basic 3D graphics education.

SECTION 1.2 Statement of Research Question

In creating VectorPad, we wanted to explore how we could best develop a system that allows users to visualize vector operations. For this reason, we explored different visualizations for some of the more complex mathematical operations present in VectorPad. Through the design of VectorPad, we hope to have gained some insight into this question.

SECTION 1.3 Thesis Overview

This thesis is separated into several chapters. Chapter 2 is a discussion of related work on vector visualization and pen-based mathematical systems. Chapter 3 will cover the user interface elements of VectorPad, and Chapter 4 will cover its system architecture. Chapter 5 will present an informal user study evaluating how users used VectorPad, as well as VectorPad's visualizations and user interface. Finally, Chapter 6 will present a discussion of possible future directions for VectorPad. Finally, examples of MathML code for some equations can be found in the appendix.

CHAPTER 2: RELATED WORK

SECTION 2.1 Handwriting Mathematics

Mathematics and pen-based computing has been explored in a number of ways. First, there are a variety of systems that use a stylus simply as a mathematics input system. In [7], Fateman examines a number of pen-based mathematical input systems. The Freehand Formula Entry System (FFES) is a mathematical formula editor that uses handwriting as its input [28]. FFES also includes the ability to generate LaTeXfrom its input. Commercially, there are systems available such as Microsoft's Equation Editor, which can export to MathML.

Similarly, there are interactive computation systems that take pen-based mathematics as input and solve the mathematics. MathBrush is one such system, and uses a similar system architecture as VectorPad, containing a character recognizer, a structural parser, a typesetting system, and an interface with a computational engine [13, 14]. Microsoft Math [19] is a commercial system that,

among other things, allows the user to input mathematics using handwriting and solve the expressions.

There has been much discussion on the usage of parse trees in mathematical handwriting recognition. One such system that uses parse trees is PenCalc, another mathematical calculation system that uses handwriting as its input [3]. Zeleznik et. al. developed MathPaper [32], a gestural system for pen-based mathematics, using a variety of gestures similar to MathPad^2 [16]. MathPaper supports multiple computational engines, as well as a set of notations for controlling computational assistance. We implement automatic updating in VectorPad, because we consider it to be an important feature in mathematical computation systems, and MathPaper implements this feature as well.

Some papers have examined the issue of portability of mathematics recognition and input systems among applications. MathInk is another a system for pen-based mathematics, built to explore "how to organize pen-based interfaces for mathematical software systems" [27]. One goal in creating MathInk was to provide a mathematical input system for document processing systems and computer algebra systems, like Maple. Furthermore, MathInk uses MathML as its representational format for its computational system. In [26], Smirnova and Watt examine various systems for pen-based mathematical computing in an effort to find a system that can integrate with document processing and computational algebra systems.

Another type of pen-based mathematical system is mathematical sketching. Mathematical sketching is a pen-based, gestural system for mathematics problem solving [15]. In mathematical sketching, a user writes mathematics and sketches supporting figures. A mathematical sketching system must recognize mathematics, solve the mathematics, and then animate the sketched figures. These animations give the user the ability to use their intuition of the mathematics and other domain knowledge to verify that the math is correct. The first mathematical sketching system is MathPad^2 , which solves equations, graphs expressions as well as mathematical functions, and animates physics sketches.

With VectorPad we wanted to explicitly combine pen-computing and vector algebra to explore an area not fully realized in other works.

SECTION 2.2 Vector Visualization

Three-dimensional vectors are generally visualized using a Cartesian coordinate system, particularly in math and physics textbooks. Often times 3D vector visualization is explored in the context of geometry visualization. Construct3D is an augmented reality system designed to aid in geometry and mathematics education [11]. Construct3D explores general techniques for 3D geometric visualization. In [17], Malkowsky discusses some applications of a software program for geometry visualizations and animations, particularly applications in differential

geometry.

Most computer algebra systems provide some sort of graphing ability. Some systems use the graphing abilities of their computational backend to provide vector visualization. MathEdge is a mathematical problem-solving and visualization program that uses Maple as its backend [18]. It provides visualizations for scalar and vector operations, including differential vector calculus. Tront's VectorPad [30] is a pen-based system for vector visualization. It allows the user to draw vectors and perform some mathematics using the vectors.

There are many small applications available on the Internet for basic vector mathematics visualization, particularly in two dimensions. In [5], Crowe and Zand discuss numerous tools available on the Internet for mathematics education, including tools for linear algebra and vector mathematics. Cataloglu performed a study on the usage of open source mathematical tools in vector mathematics education, which used Octave and a number of Java applets related to vector mathematics and physics [2]. Eason and Heath discuss how Java applets can help in learning mathematics and present several Java applets, including one for 2D vector visualization [6]. Kawski presents Vector Field Analyzer II, a Java applet for dynamic vector field visualization [12].

While we ultimately did not include vector fields in VectorPad, it was an area we explored while developing VectorPad. Vector fields are central to a number of

mathematics fields, including fluid dynamics; consequently, vector field visualization has been explored in a number of papers. Hesselink et. al. explore research issues in vector and tensor field visualizations in [10]. Schroeder et. al. present a technique for 3D vector field visualization, the stream polygon, in [24]. In [23], Post and van Wijk explore a number of vector field visualization methods, including arrow plots, streamlines, stream ribbons, and particle motion animation. Telea and van Wijk present a simple method for vector field visualization in [29].

Unlike other most explorations in vector visualizations, which focus upon vector fields, we focused upon the visualization of basic linear algebra operations.

CHAPTER 3: USER INTERFACE

SECTION 3.1 Ink Canvas

VectorPad was designed to have an interaction model similar to pen and paper. We chose this model as it seems to be more natural for mathematics input. To this end, we provide a sketching area for the user to write mathematics as they normally would using pen and paper (Figure 3.1). Using a stylus, the user inputs mathematics through writing and by performing a variety of gestures. Gestures are used to erase writing, group strokes into equations, and perform a variety of other functions. VectorPad uses combinations of three gestures, a lasso gesture, a tap gesture, and a scribble gesture. We chose these gestures due to the ease in recognizing them using heuristic recognition. Since we have a reduced gesture set, combinations of gestures were chosen to have similar actions.

Each stroke the user writes is passed to an online mathematical handwriting recognizer, which recognizes the individual characters and the mathematical

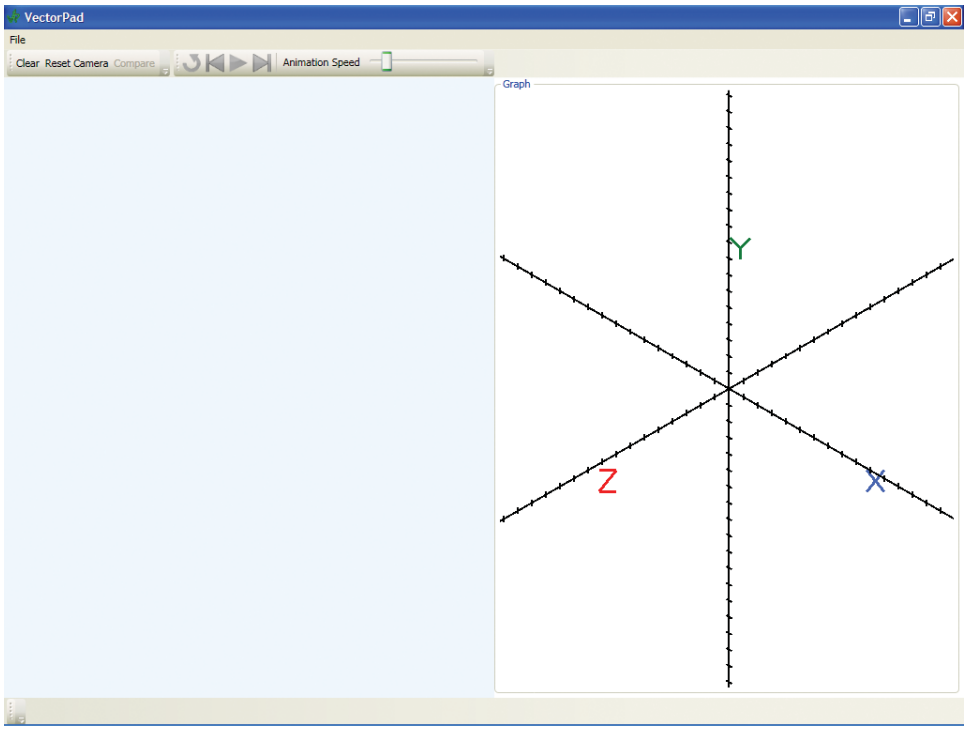

Figure 3.1: VectorPad's main window: on the left is the sketching area, on the right is the graph

structure. The results from the recognizer are displayed below the user's writing in real-time, using a typeset mathematical system (see Figure 3.2). After a character is recognized, alternate recognition results are displayed on an alternates menu. Selecting an alternate result changes the recognizer's recognition for the character. Using a lasso-type gesture around any set of strokes brings up a list of alternates for those strokes. Characters are grouped into equations using a lasso-tap gesture around the strokes, as in [16]. Users can erase using a scribble erase gesture on any strokes.

Once an equation is created, it can be edited by adding more strokes, or erasing

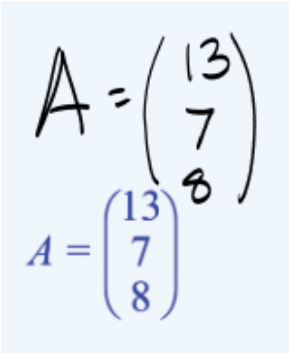

Figure 3.2: Typeset recognition results for a vector

strokes and adding more. The updated results are reflected in the graph. The results from one equation can also be used in another. When one equation depends on another, the dependent equation receives updates from the equation it depends upon.

SECTION 3.2 Graph

The graph in VectorPad is implemented using a 3D viewport with an orthographic camera. We chose to use an orthographic camera because it provides the view that is commonly used in mathematics and physics textbooks. The graph uses a three-dimensional Cartesian coordinate system, with the origin at the center of the graph visually. This system is presented to the user as a set of three number lines with tick marks every one unit of measurement (see Figure 3.3). At the outer edge of the positive side of each number line is a colored text label with the axis's name (X, Y, or Z). We experimented with several different forms of number lines, with

different tick marks and visual styles. We think that this style works best in VectorPad because it was most similar to styles that would be familiar to users.

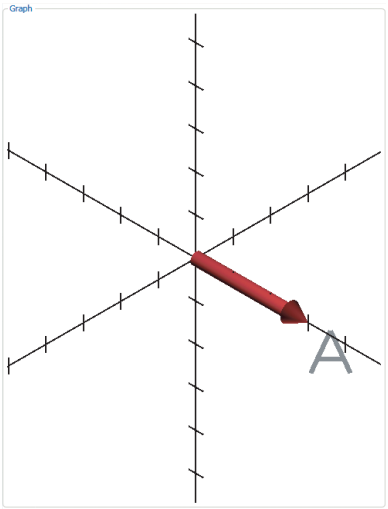

Figure 3.3: One vector has been recognized, A. A is shown on the graph as a red arrow, and VectorPad has automatically zoomed in to maximize its view.

After performing the gesture to recognize an equation, vectors are dynamically illustrated on the graph. Each vector is visually represented by a colored arrow and the equation is highlighted with the arrow's color. This highlight helps the user to understand the bounds of the equation, both for editing the equation and writing other equations. When a mathematical operation is recognized, the operation is animated on the graph. Each operation has its own animation, designed to illustrate the operation in an intuitive manner.

We explored several visual styles for arrows, starting with a basic line that appeared two-dimensional, and a three-dimensional arrowhead. While these lines gave a more sketchy style, the two different styles of the arrowhead and body did not mesh well

together. Next, we tried using arrows that appeared completely flat. However, this flatness did not work well, because it did not provide much of a sense of three-dimensionality, and we moved on. The third style we explored was much like the style we ultimately used; however, the radius of the arrow was larger. After users commented that they felt the arrows were too large, we decreased the radius of the arrows, and arrived at the style used in VectorPad.

We explored several options for controlling the graph. First, we considered having six buttons, two for each axis, with one to rotate along that axis positively, the other negatively. This idea was discarded because it required the user to press buttons repeatedly to rotate a large amount and still allow the user to be able to make small rotations. Another idea we explored was having the user rotate the graph by moving the stylus in the direction they want to rotate, with the stylus speed determining how far "into" the graph to rotate. Ultimately, we decided to use a trackball model to allow the user to control the graph, where the user moves the stylus throughout the graph to rotate it [4]. This moves the camera around the graph, with the user able to rotate along each axis, although they cannot isolate to a specific axis of rotation (see Figure 3.4). The user can also zoom by pressing the stylus button and moving the stylus up to zoom in, and down to zoom out. After the user adds an equation, the graph is automatically zoomed in or out to fit the various arrows onto the screen, so that they are visible.

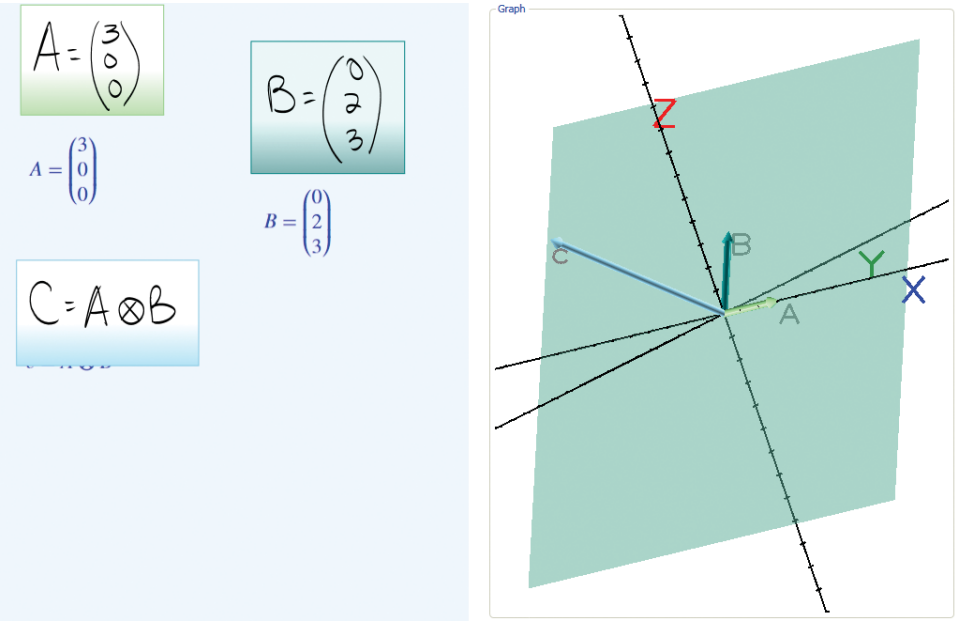

Figure 3.4: Three vectors are shown on the graph, A, B, and C. C is the cross product of A and B. The plane made by A and B is shown as a thin plane colored a greenish-blue hue. C is shown as a purple arrow. The graph has been rotated and zoomed.

A user might want to only see the arrows for a specific equation, so we provide a gesture combination to hide specific arrows. To hide the arrows for an equation, the user performs a lasso-double-tap gesture. The arrows are no longer displayed on the graph and are not used in calculations for automatic zooming. We chose a similar gesture to that used in creating an equation, because we felt that the action of hiding the equation's arrows is in some ways the opposite of creating an equation. We also provide a way for the user to zoom the graph to fit to the arrows for a specific equation. This is accomplished by double-tapping on an equation, which was chosen because double-clicking is often used to zoom to a specific point in applications such as mapping software [8].

Another gesture that VectorPad uses is a single tap on an equation, which highlights all the arrows defined by the equation; tapping outside of an equation removes all highlighting. This highlighting provides another means for the user to identify which arrow belongs to which equation.

We wanted users to be able to see information about the vectors that is not readily apparent from the graph, like the normal of the vector, its length, and its equation. Clicking on an arrow brings up a small visual overlay on the graph that shows its equation, its vector value, length, and normal.

SECTION 3.2.1 Text

Many things are labeled within the graph, for example the number lines. This text is visually two-dimensional, but has a three-dimensional location within the graph. As the graph is rotated by the user, the text rotates to stay facing the camera. As the graph zooms, the thickness of the text is also changed to maintain readability. When zooming in beyond 100%, the text thickness increases so that it keeps a relative thickness in comparison to other objects in the graph. As the user zooms back out, the text decreases to its normal thickness at 100%. While designing the text labels, we considered two other options, using fully 3D text models and 2D text on an overlay to the graph. Since we are rotating the text to always face the camera, 3D text would not have provided any additional benefits. Two-dimensional

text automatically faces the user, but it would occlude three-dimensional objects. For these reasons we chose to use "flat" 3D text.

SECTION 3.3 Visualization Constructs

VectorPad supports several mathematical operations, vector addition and subtraction, scalar multiplication and division, cross product, dot product, and length. When an operation results in a vector answer, a new arrow is added to the graph. Result arrows are colored one of several pink hues to differentiate them from other arrows. For vector-result operations, the operand arrows are moved around the graph and leave behind semi-transparent shadow copies. The animations first typically show the operands moving so that the user can see how the operation affects the two operands. Then, an arrow for the result is added to the graph. This shows the user the operation separate from its final representation and allows them to focus upon the operation before examining the result arrow. The length of an animation can be changed using a slider with values ranging from 0.5 seconds to 10 seconds. This allows the user to examine how the operation works at their own pace. When an animation has finished, any arrows that have moved return to their original positions and the shadow copies are removed.

SECTION 3.3.1 Addition and Subtraction

Addition is an operation that has two operands and results in a vector result. Thus, the addition animation has two component arrows and a result arrow. In vector addition, the components of both vector operands are summed. The animation for addition takes the arrow for the second operand and translates it such that its origin is no longer the origin of the graph, but the endpoint of the first arrow. This places the endpoint of the first arrow where the result's endpoint will be. Once the second arrow has finished translating, a new arrow is created with a normal origin and its endpoint at the point defined by the operation.

Subtraction is similar to addition, except that the second arrow first rotates to its vector's opposite (i.e. its negative vector, see Figure 3.6). We chose this animation for addition and subtraction, because it shows the user the common visualization of these operations in vector mathematics, a triangle made of the operands and the result.

SECTION 3.3.2 Scalar Multiplication and Division

Scalar multiplication and division share the same animation style. This animation has two operands, one a vector and the other a scalar, and the result is a vector. In scalar multiplication, all three components of the vector operand are multiplied by

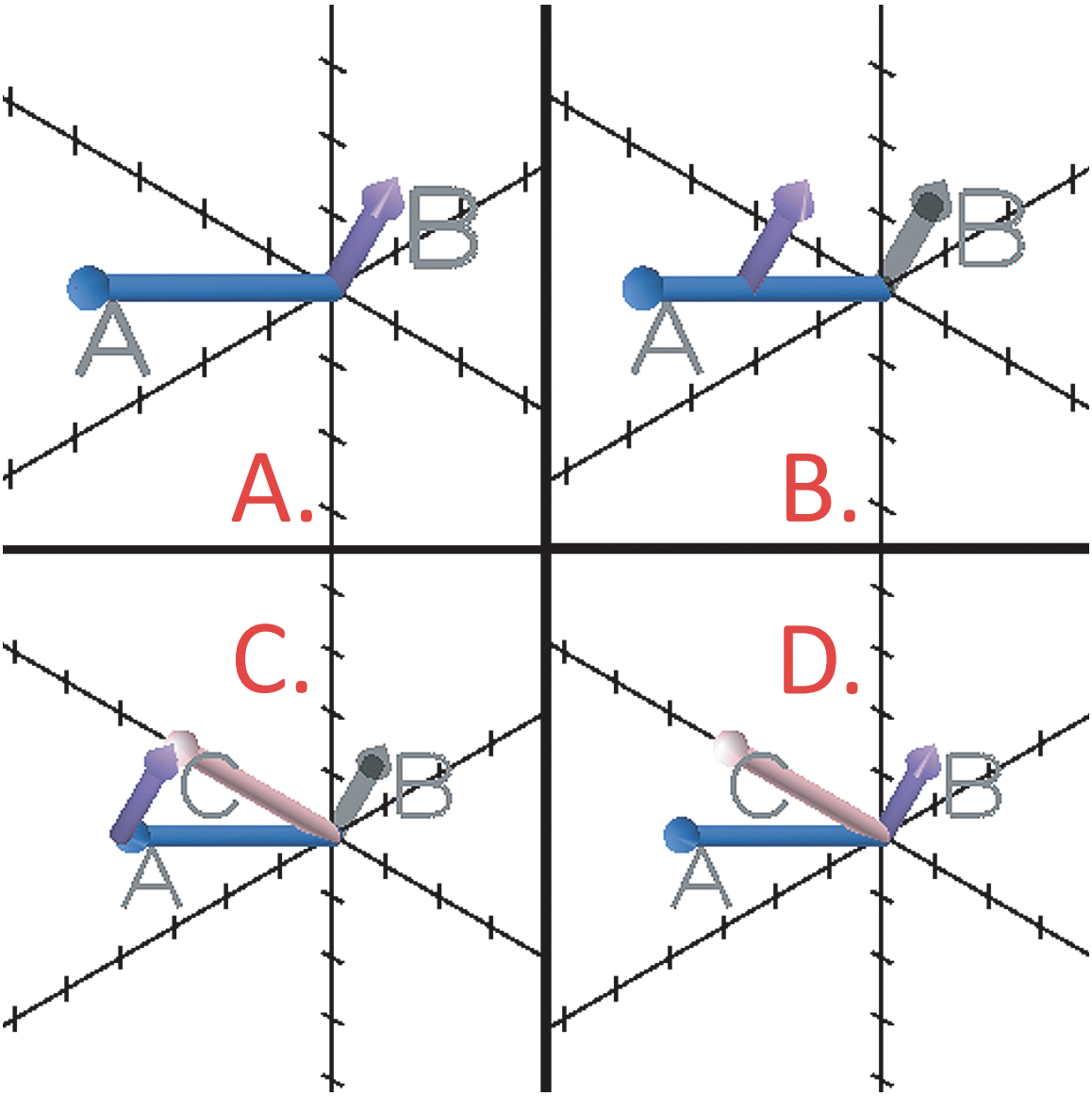

Figure 3.5: Addition animation for A=[3, 5, 7], B=[1, 2, 0], C=A+B. The arrow for A is shown in blue, and the arrow for B in purple. Subfigure A. shows arrows for vectors A and B. Subfigure B. shows an intermediate state as B moves to the end of A. Subfigure C. shows B at the end of A as well as C, making the traditional three-vector triangle. The arrow for C is shown in pink. Finally, Subfigure D. shows A, B, and C, which is equal to [4, 7, 7].

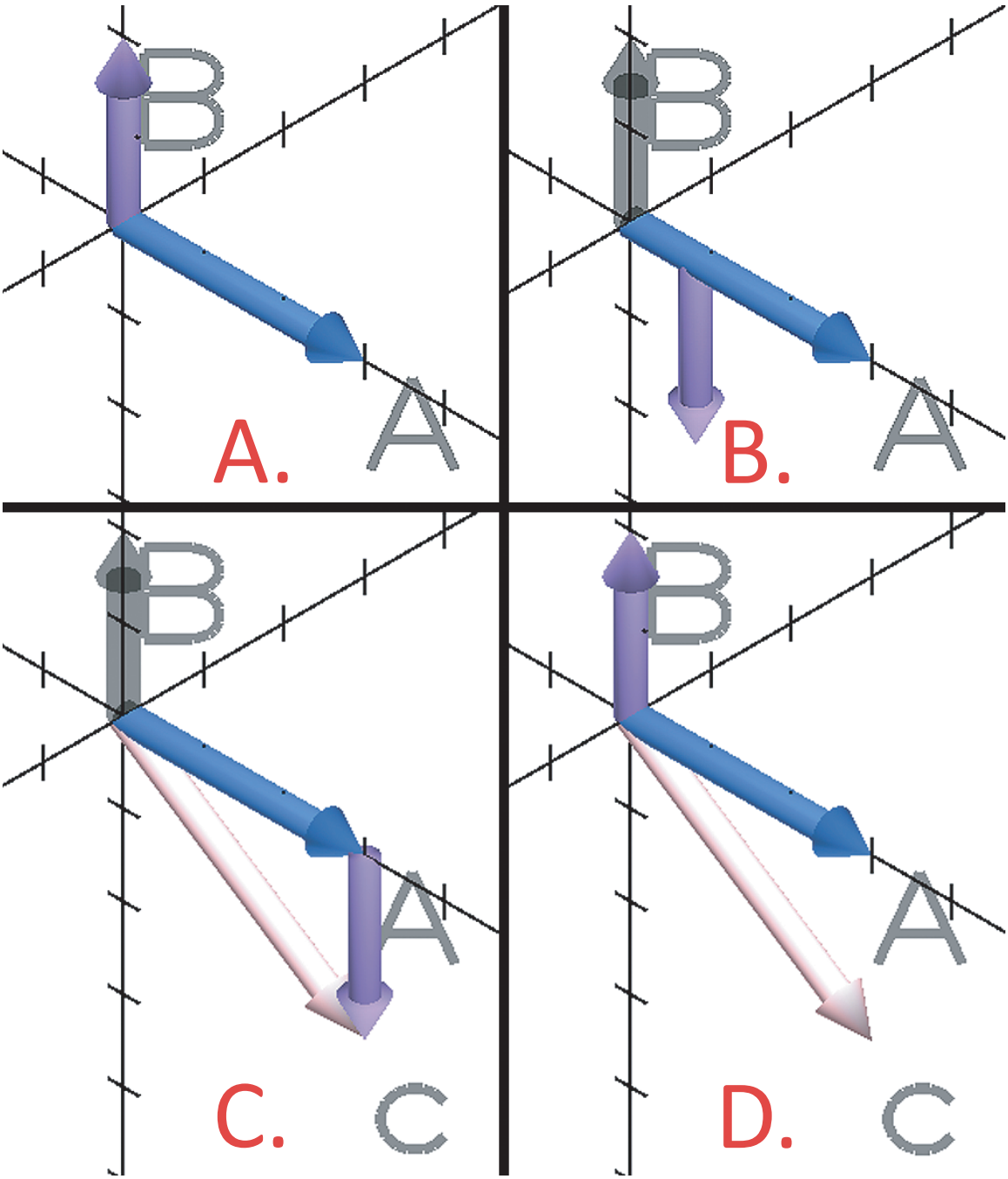

Figure 3.6: Subtraction animation for A=[3, 0, 0], B=[0, 2, 0], C=A-B. The arrow for A is shown in blue, and the arrow for B in purple. Subfigure A. shows arrows for vectors A and B. Subfigure B. shows an intermediate state after B has rotated to [0, -2, 0] and moves toward the end of A. Subfigure C. shows B at the end of A as well as C, making the traditional three-vector triangle. The arrow for C is shown in pink. Finally, Subfigure D. shows A, B, and C, which is equal to [3, -2, 0].

the scalar operand. In scalar division, all the vector components are divided by the scalar operand. Since there is only one vector operand, there is only one component arrow. For scalar multiplication and division, there are two animation variations. In the first animation, the component arrow scales to the resulting size specified by the vector times the scalar (see Figure 3.7). In the second animation, which we termed the "stacking" animation, the component arrow is stacked end over end the number of times specified by the scalar value (Figure 3.8). In both cases, once the components' animations are finished, a new arrow is created with a normal origin and its endpoint as specified by the vector mathematics. Initially, we created one animation for these operations, where the arrow scales to its final length, but we added the second type because we felt that this did not show the relationship between the scalar value and vector value very well.

SECTION 3.3.3 Cross Product

The cross product animation has two operands, both vectors, and the result is also a vector. A cross product of two vectors results in a vector that is perpendicular to both operand vectors. For cross product operations, we created two animations. In the first animation, the first component arrow rotates to the second component arrow and then rotates to the direction specified by the resulting perpendicular vector. Finally, a result arrow is added in the direction of the result vector (see

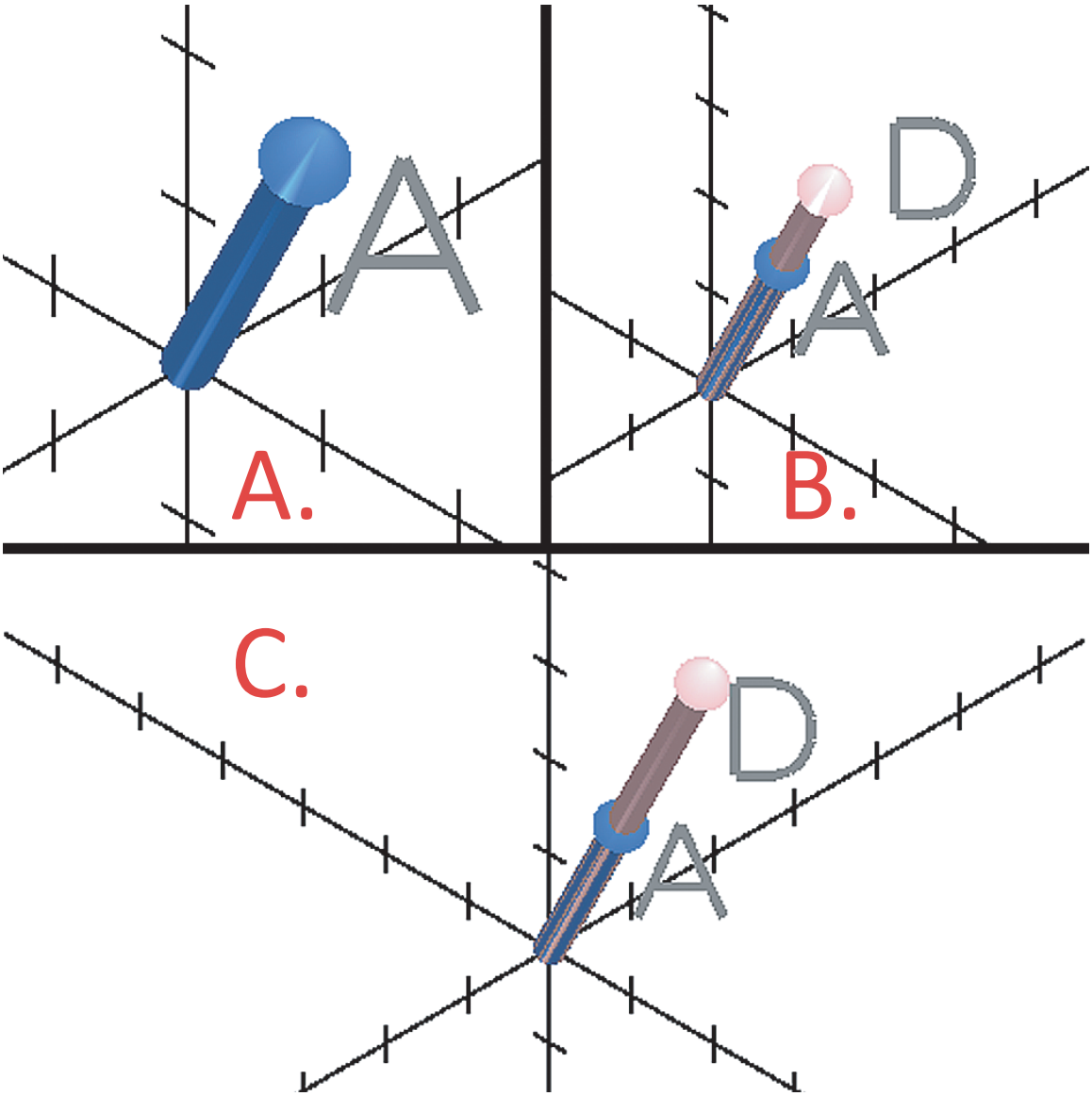

Figure 3.7: The first scalar multiplication animation showing A=[2, 3, 1], B=2, D=A*B. The arrow for A is shown in blue. Subfigure A. shows the arrow for vector A. Subfigure B. shows an intermediate state where D is scaling to its final value. The arrow for D is shown in pink. Finally, Subfigure C. shows D at its full length, [4, 6, 2].

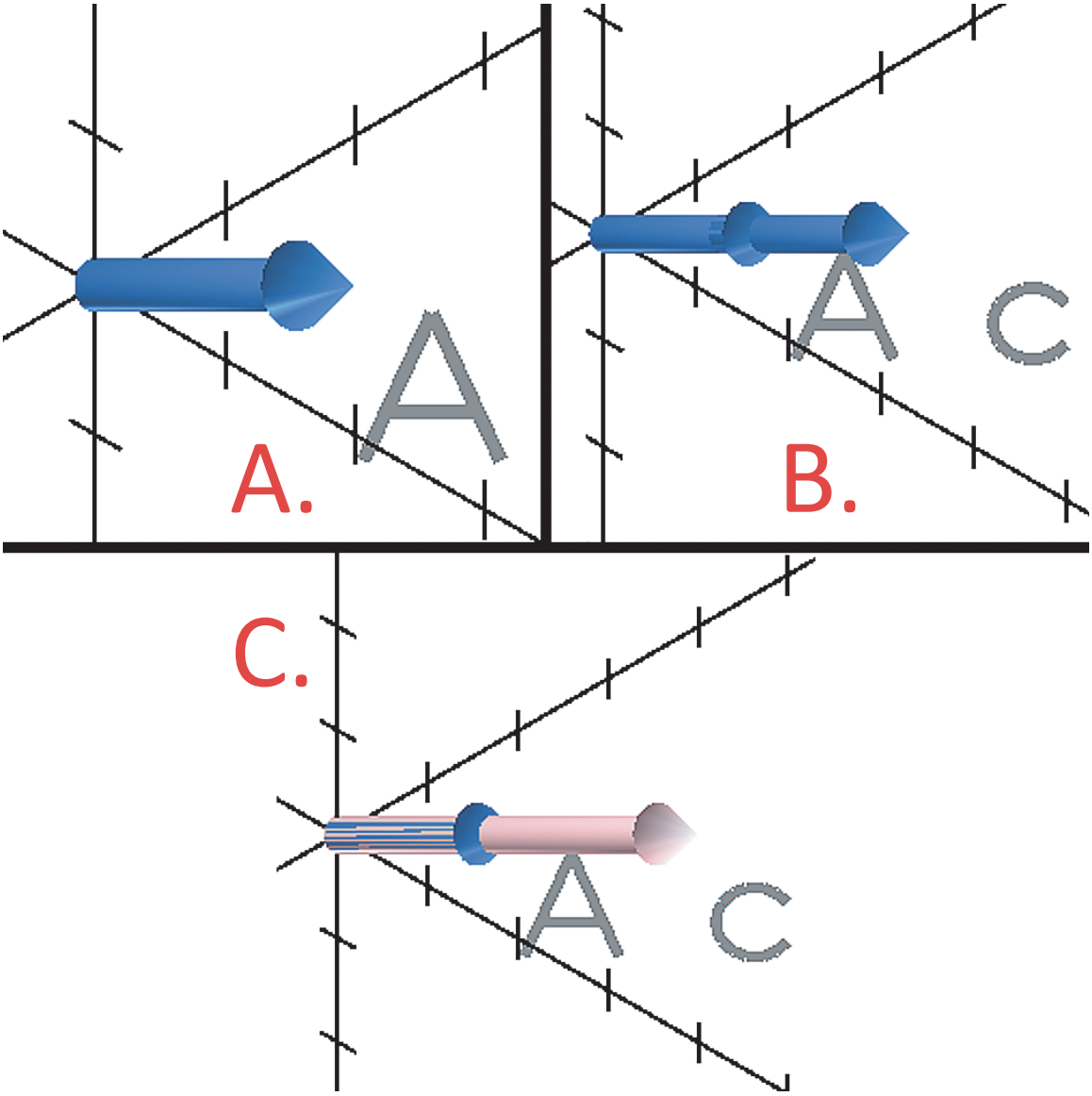

Figure 3.8: The second scalar multiplication animation showing A=[2, 1, 0], B=2, C=A*B. The arrow for A is shown in blue. Subfigure A. shows the arrow for vector A. Subfigure B. shows an intermediate state where the arrows are stacking. A second copy of the arrow for A is moving to the end of the first copy of the arrow for vector A. Finally, Subfigure C. shows C at its full length, [4, 2, 0]. The arrow for C is shown in pink.

Figure 3.9). The second animation is similar to the first except that a semi-transparent plane is added to illustrate the plane made by the two component arrows (see Figure 3.10). We created this animation after some users expressed that they were used to seeing a plane for a cross product.

SECTION 3.3.4 Dot Product

A dot product animation has two operands, again, both vectors, and the result is a scalar value. Dot product, also known as the scalar product, multiplies the like vector components of the two operands and sums them together. We chose to display text results on the graph, since we have a scalar result. Another important aspect of the dot product is its relation to the angle between the two vectors. By taking the length of the first operand and multiplying it by the cosine of the angle between the two vectors, the first vector is scalar projected onto the second vector. This is often illustrated when showing the result of a dot product, so we felt it was beneficial to show it too. We draw a distinctly colored arrow to show this projection. In the case of perpendicular vectors, the result of a dot product operation is zero, and no arrow is shown on the graph, although a text result for the dot product result is shown.

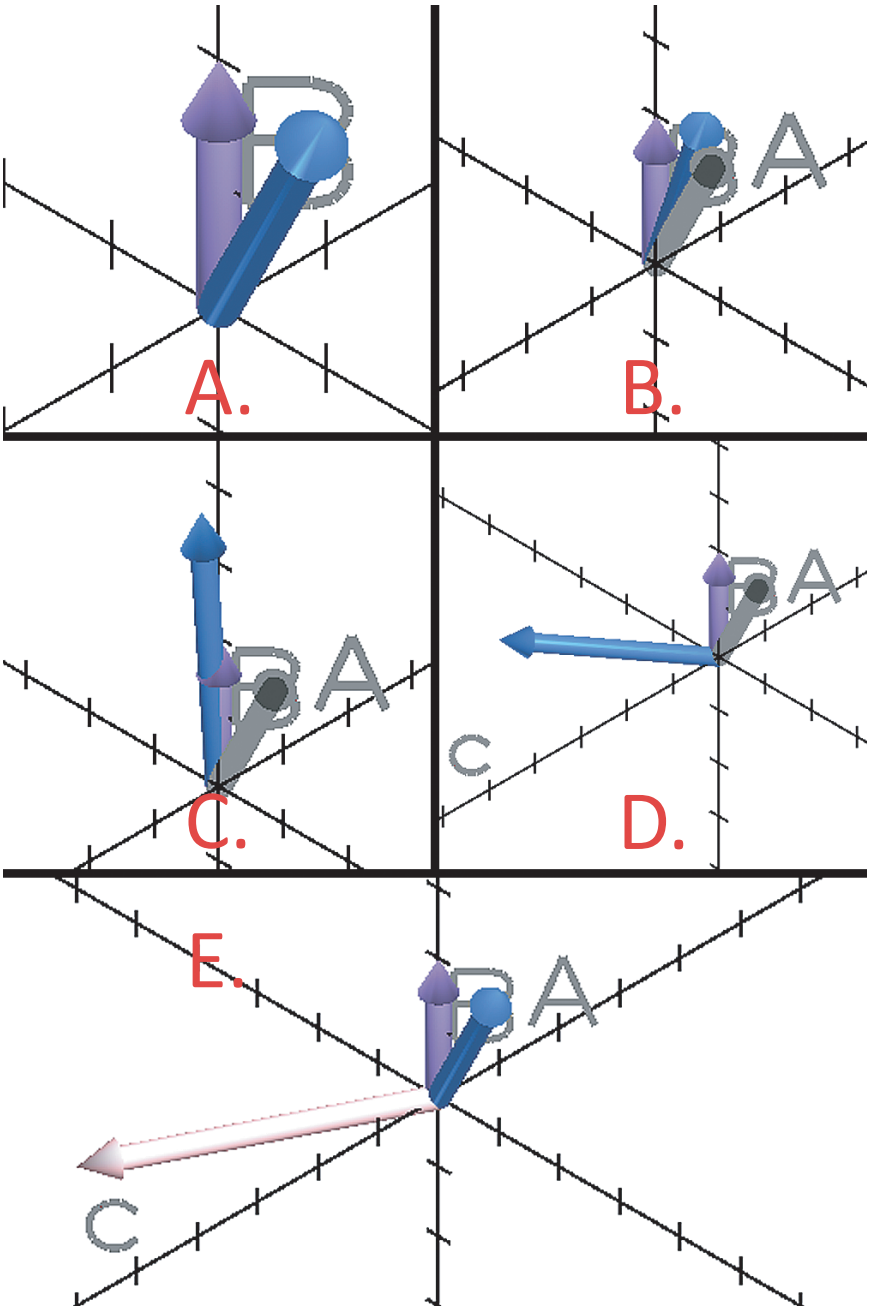

Figure 3.9: The first cross product animation showing A=[2, 3, 1], B=[0, 2, 0], and C=A⊗B. The arrow for A is shown in blue and the arrow for B in purple. Subfigure A. shows the arrows for vectors A and B. Subfigure B. shows an intermediate state where B is rotating to A and B is changing lengths to the length defined by the cross product of A and B. Subfigure C. shows another intermediate state where B has already rotated to A and is now rotating to the position defined by the cross product of A and B. Subfigure D. shows B further along its rotation. Finally, Subfigure E. shows C which is [-2, 0, 4]. The arrow for C is shown in pink.

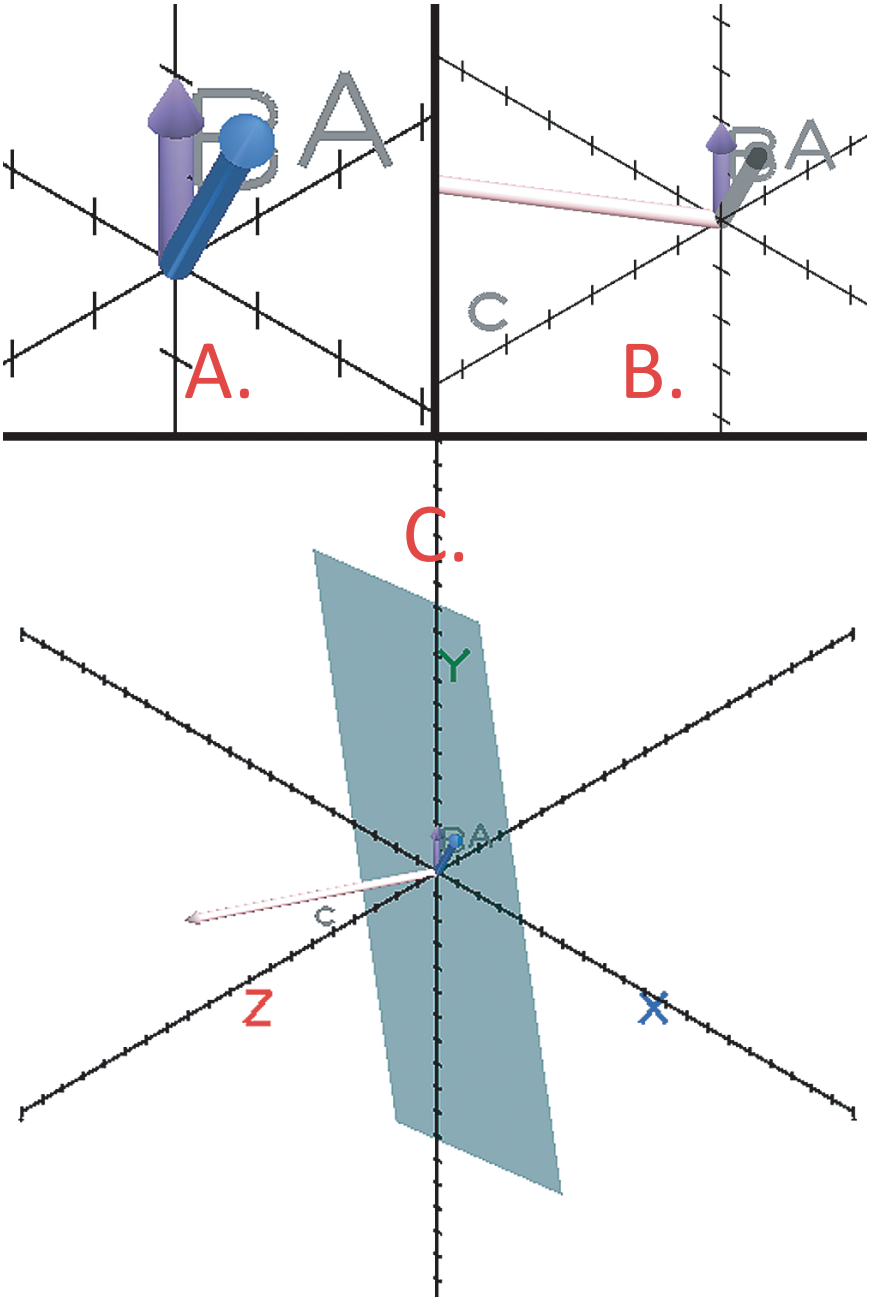

Figure 3.10: The second cross product animation showing A=[2, 3, 1], B=[0, 2, 0], and C=A⊗B. The arrow for A is shown in blue and the arrow for B in purple. Subfigure A. shows the arrows for vectors A and B. Subfigure B. shows an intermediate state where an arrow for C is scaling to its final length. The arrow for C is shown in pink. Finally, Subfigure C. shows the plane made by A and B, and C at its final position [-2, 0, 4].

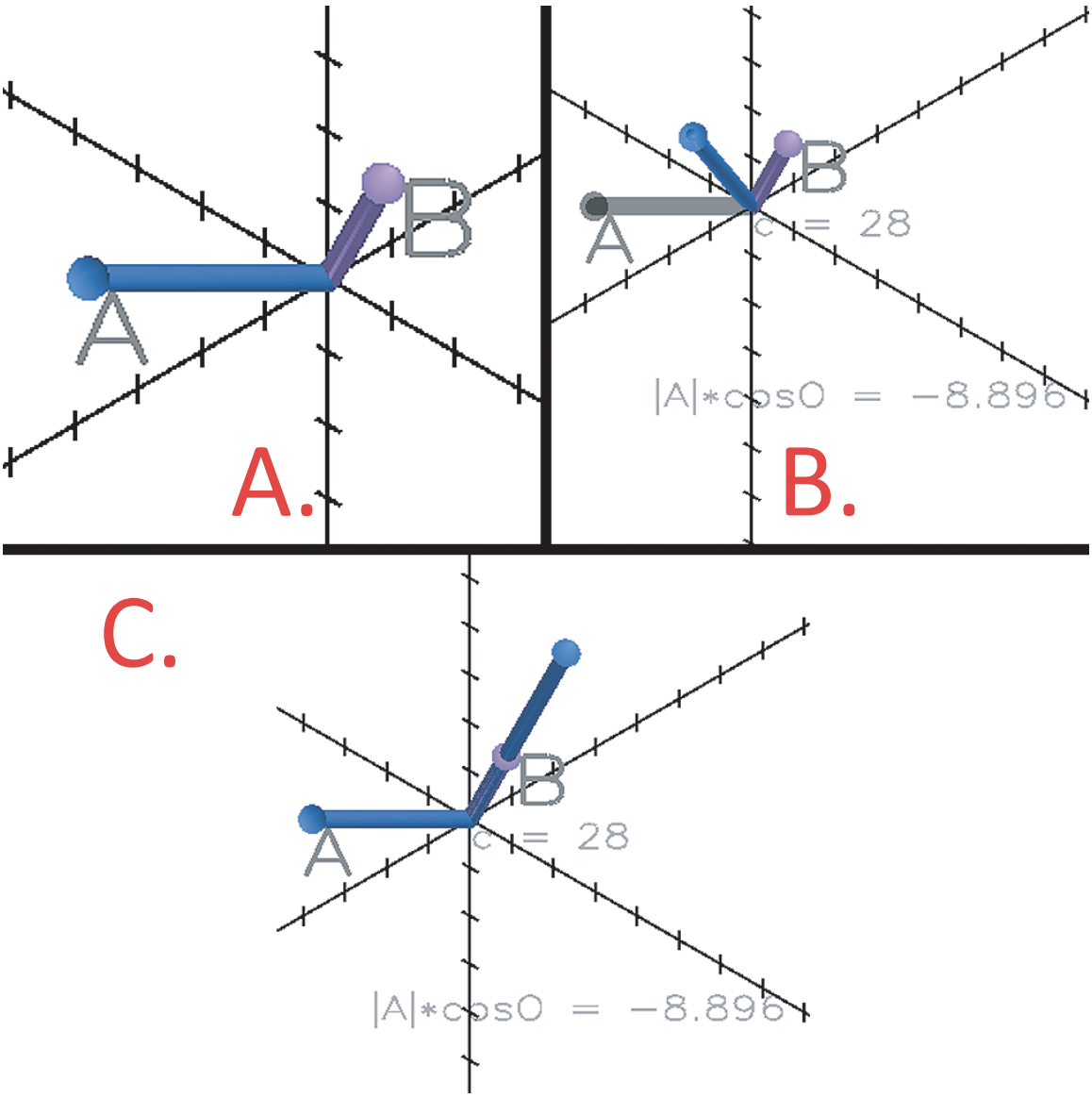

Figure 3.11: The dot product animation showing A=[3, 5, 7], B=[2, 3, 1], C=A·B. The arrow for A is shown in blue and the arrow for B in purple. Subfigure A. shows the arrows for vectors A and B. Subfigure B. shows an intermediate state where A is rotating to show the scalar projection of A onto B. Text shows the value of C at the center of the graph, as well as the value of $|A| * \cos\Theta$. Finally, Subfigure C. shows the scalar projection of A onto B.

SECTION 3.3.5 Complex Operations

VectorPad allows users to create complex equations utilizing multiple operations; for an example see Equation 3.4. To make it easier for users to understand complex equations, we divide complex equations into a series of simple animations. For Equation 3.4, which is defined using Equations 3.1–3.3, VectorPad creates animations for two subequations (see Equations 3.5 and 3.6). First, the animation for Equation 3.5 would play, followed by Equation 3.6. The complex animation for Equation 3.4 is show in Figure 3.12. Each animation for a subequation appears in the animation list as a normal animation, so that a user can repeat it as they like.

$$A = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \tag{3.1}$$

$$B = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} \tag{3.2}$$

$$C = \begin{pmatrix} -1 \\ 2 \\ 2 \end{pmatrix} \tag{3.3}$$

$$D = A \otimes B \otimes C \tag{3.4}$$

$$T1 = B \otimes C \tag{3.5}$$

$$D = A \otimes T1 \tag{3.6}$$

SECTION 3.4 Other Operations

We also chose to implement a length operation, although it does not have an animation. Since the length operation has a scalar result, like the dot product operation, there is not a simple, but meaningful way to animate it as with dot product. We considered showing the arrow rotating along to an axis so that the user could see its length by counting the tick marks on the number line, but we did not feel that this showed the relationship between the length and the vector's component values. Consequently, when a length operation is performed, its result is shown as text on the graph (Figure 3.13). As mentioned in SECTION 3.2, this information is also available by clicking on an arrow, which brings up a small overlay (see Figure 3.14).

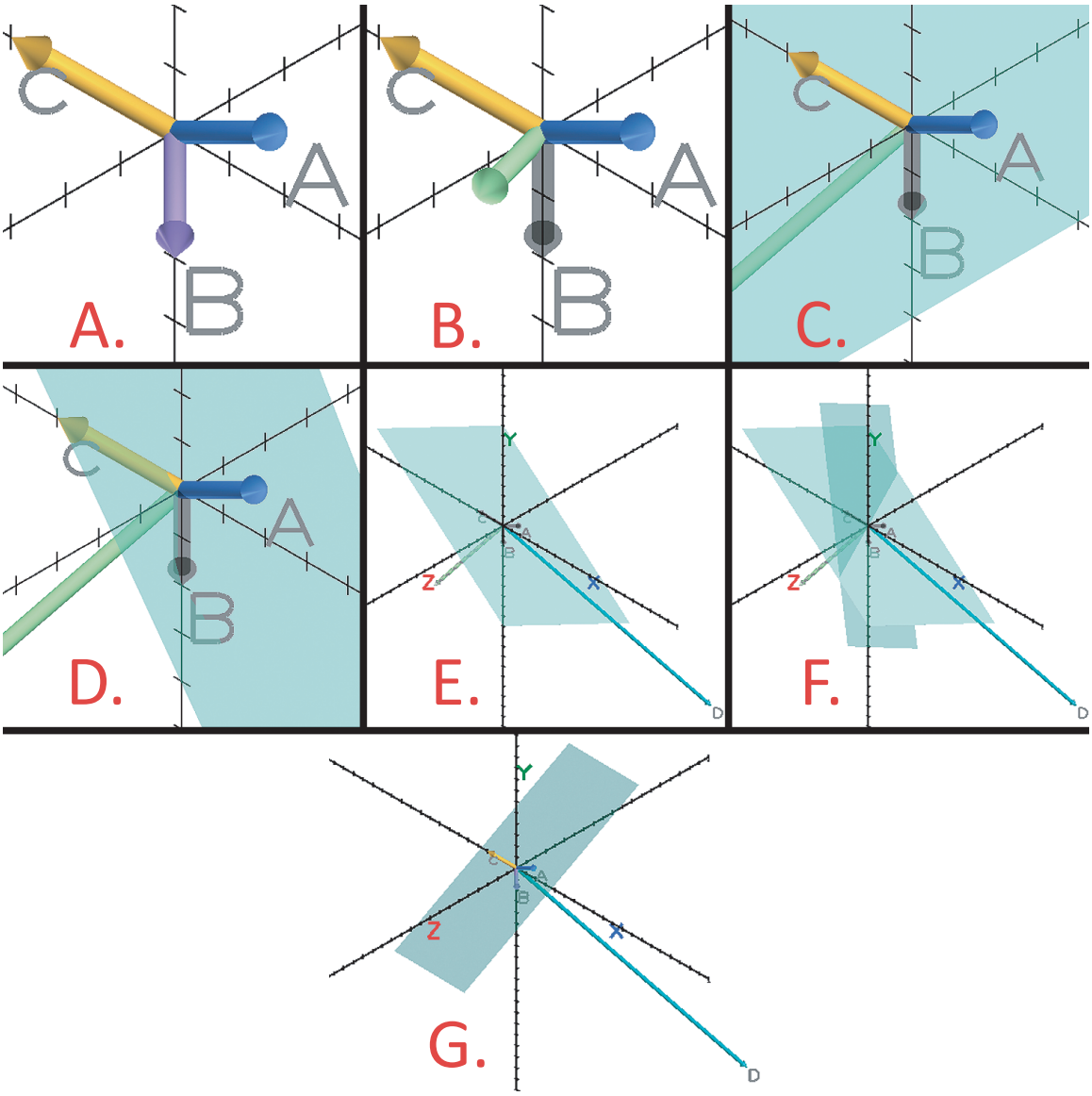

Figure 3.12: Two cross product animations for the complex equation shown in Equation 3.4. The arrows for A, B, and C are shown in Subfigure A. A=[3, 2, 1] and is shown in blue. B=[2, 0, 2] and is shown in pink. C=[-1, 2, 2] and is shown in yellow. In Subfigure B., the arrow for B is rotating to C and has been colored green to represent T1. Subfigure C. shows T1 scaling to its final length and the plane between B and C is rotating into place. Subfigure D. shows another intermediate step right before the plane has finished rotating. In Subfigure E. the plane between B and C is in place and an arrow for D has scaled and rotated to its final position. The arrow for D is colored turquoise. Subfigure F. shows an intermediate state where the plane between A and T1 is rotating to its final position. Subfigure G. shows that plane in its final position.

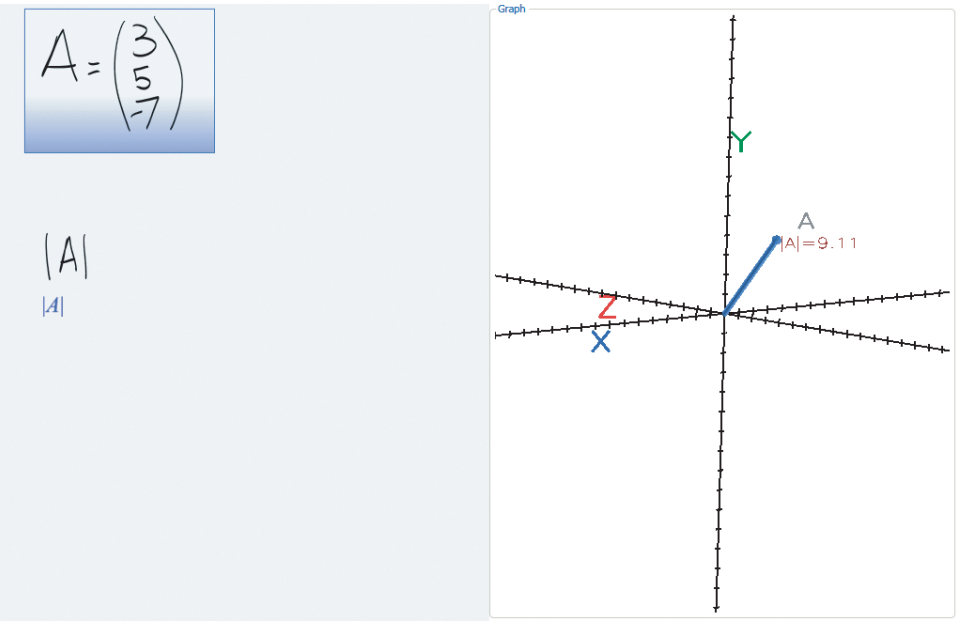

Figure 3.13: The length of a vector can be shown by explicitly writing the operation.

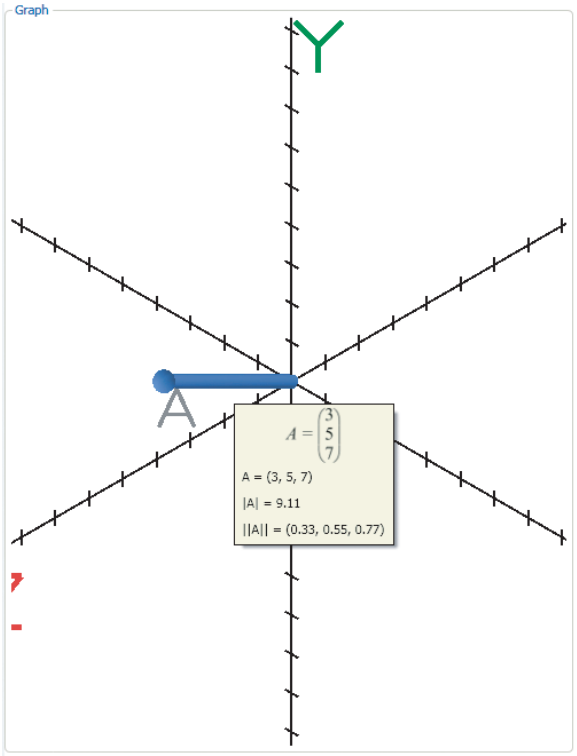

Figure 3.14: The length of a vector can also be shown by tapping on an arrow, which brings up a small display with the vector's definition, value, length, and normal.

SECTION 3.5 User Controls

VectorPad has a set of VCR-like controls to give the user the ability to playback animations. We considered the ability to replay animations important while designing VectorPad, because a user might not understand an animation on their initial viewing. There is a slider control and four VCR buttons, play, backward, forward, and replay. The slider controls the length of time any animation plays for, from 0.5 seconds to 10 seconds. The default value of the slider is 2 seconds, which we found to be a good intermediate speed and gives users sufficient time to examine the operation. A list of animation states is maintained, and the buttons move an index throughout the list. The replay button plays the last animation in the list, giving the user the ability to easily repeat the last animation over and over until they understand it. The backward button decreases the index, while the forward button increments it. The play button plays the animation at the current index.

We have several other buttons in the VectorPad user interface. Users can clear the sketching area, graph, and internal data structures using the clear button. This resets VectorPad to its initial state. Since users can control the graph area by rotating it and zooming in and out, we felt that it would be very useful to give the user a way to reset the camera to its default position and zoom level. This is accomplished through the reset camera button. Finally, the compare button brings up a second window with two graphs to allow the user to compare changes made to

equations.

At the bottom of the window is an alternates toolbar, which is provided by one of the tools we used to construct VectorPad (see SECTION 4.4). Initially, the toolbar is empty, but after any stroke is recognized, alternates for the stroke (or last group of strokes) are populated on the toolbar. By selecting an alternate, the recognizer is instructed on how to parse those strokes. To select an alternate for specific strokes, the user can lasso select the strokes, which will populate the toolbar.

SECTION 3.6 Comparison of States

Through the use of the compare button, users can compare two states of the equations, which brings up a window with two graph areas (Figure 3.15). Each area can be independently controlled with the same stylus-based controls as in the graph on the main window. A new state is created when a new equation is created, or an equation is updated. Under each graph, there are buttons to move forward or backward in the state list for that graph area. Using this comparison window, a user can easily see how changing a vector changes an operation.

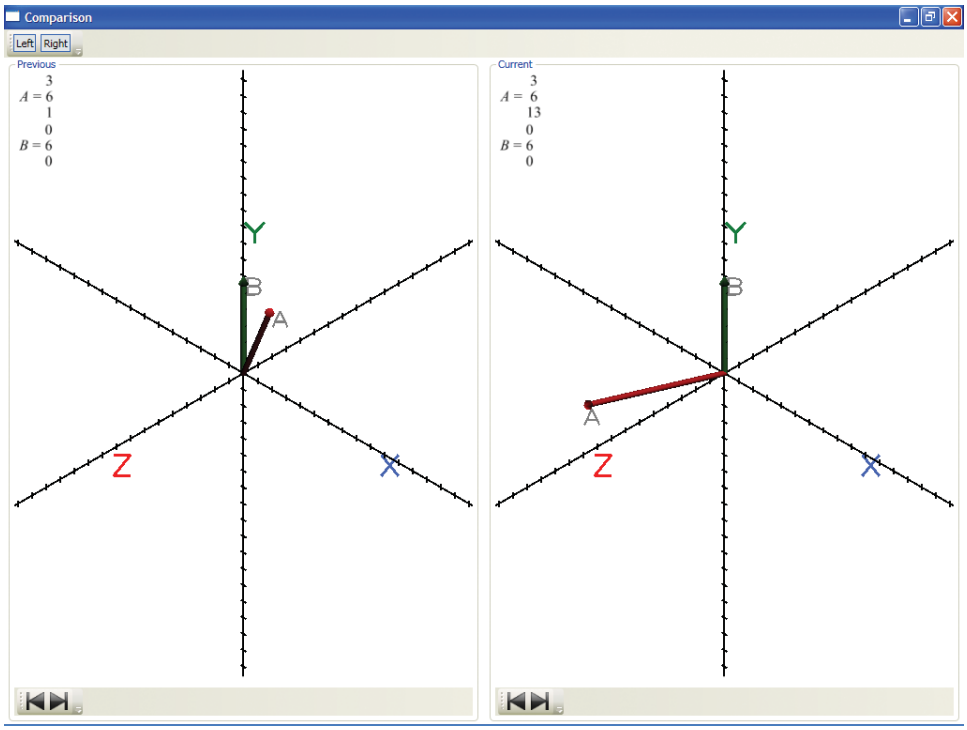

Figure 3.15: Comparison window

CHAPTER 4: ARCHITECTURE

In this chapter we present the software architecture used in VectorPad including its important components, runtime processes, and the tools used to create VectorPad.

SECTION 4.1 Components

The architecture of VectorPad is divided into three primary components, one to handle equations and other mathematics, one to handle the graph, and one to handle the sketching area (see Figure 4.1). The component that deals with equations is deemed the EquationHandler, and as its name implies, it creates, updates, deletes, and performs other operations with equations. The EquationHandler also interfaces with the computational engine. Details about its important subcomponents can be found in SECTIONS 4.1.2–4.1.4. The graph-related component, the GraphicsHandler, creates arrows, planes, animations, and other graphical objects, fits the camera to the models, and prevents text occlusions. The GraphicHandler's subcomponents are described in SECTIONS 4.1.5–4.1.6 and

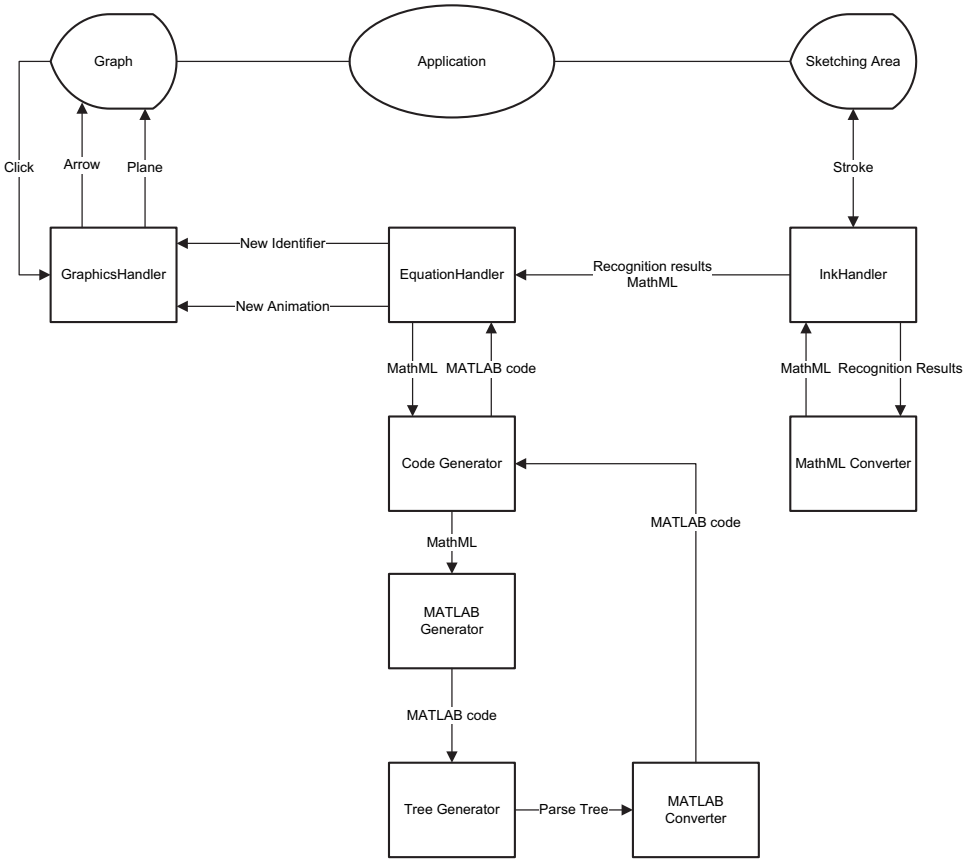

Figure 4.1: The components that make up VectorPad. Arrows show direction of data flow.

SECTION 4.4.1. We call the third component the InkHandler, because it performs gesture recognition, math recognition, and other actions dealing with stylus strokes. Details on the InkHandler can be found in SECTION 4.1.1.

SECTION 4.1.1 InkHandler

Our first component in VectorPad, the InkHandler, deals directly with user input. The InkHandler consists of a mathematics recognizer and a gesture recognizer, and

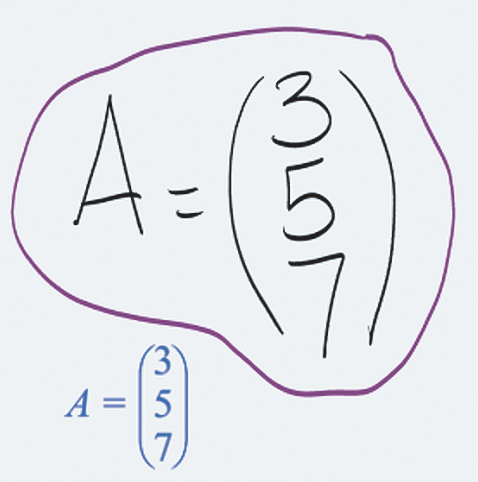

Figure 4.2: A lasso gesture

directly interfaces with the EquationHandler and GraphicsHandler components. This component is used to recognize characters and mathematics, and to display those results. When a scribble gesture is recognized, we send the strokes that have been scribbled upon to the EquationHandler to remove them from any relevant data structures (see SECTION 4.2.8). We also recognized tap gestures for a variety of actions. When a lasso gesture is recognized, followed by a tap gesture, we send any strokes that are enclosed by the lasso gesture to the EquationHandler to create an equation object (SECTION 4.2.1). Tap gestures are also used in hiding graphics, highlighting graphics, and fitting the graph's camera to specific graphics.

SECTION 4.1.2 Identifier

Special components were designed to internally represent important aspects of mathematical equations. The basest component that we use is an Identifier, which

represents a variable in an equation. Equation 4.1 derives from Figure 4.2, which we will be using as an example throughout this chapter. In Equation 4.1, there is one Identifier A, while in Equation 4.2, there are three Identifiers, A, B, and C. The main function of the Identifier component is to store important information about the variable, including its name, type, code related to it, whether the Identifier has already been graphed, and its value. Additionally, there are two types of Identifiers, primary and secondary. Primary Identifiers are those whose value is assigned to (the left hand side of the equation), while secondary Identifiers are other Identifiers used in the equation. Using Equation 4.1 as an example, there is a single Identifier, its name is "A", its type is vector, its value is a vector, $[3, 5, 7]^T$, and it is a primary Identifier. Behind the scenes, there is a secondary Identifier created, MX0, which A references.

$$A = \begin{pmatrix} 3 \\ 5 \\ 7 \end{pmatrix} \tag{4.1}$$

$$C = A + B \tag{4.2}$$

SECTION 4.1.3 Equation

The next most specific component, the equation object, represents a user-input equation. This is the main organizational unit of VectorPad. The primary purpose of an equation object is to be a collection of the Identifiers, code, and other objects that are created when an equation is written by the user. It references any number of Identifiers since the user can write an equation with numerous variables. There are numerous data members in an equation object, including an identifying number, a collection of the strokes composing the equation, a MathML structure, code for the computational engine, a list of Identifiers, and a list of arrows. MathML is an XML-based language designed to represent mathematical structures.

Conceptually, there are two kinds of equation objects: assignment equations, those that simply define an Identifier with some value (Equation 4.1), and mathematical equations, those that define an Identifier with a mathematical operation (Equation 4.2). Equation objects are created with an identifying number which counts up from zero. When a user writes an equation and performs a lasso gesture, an equation object is created, and all the strokes enclosed by the gesture, as well as the lasso stroke, are stored in the equation object. Recognition results from the math recognizer are also stored in the equation object in the form of a MathML document. During the process of converting the MathML to code executable by the computational engine, relevant Identifiers are added to the equation object. Also,

when the equation object's computational engine code is executed, and arrows are added to the graph, references to the arrows are added to the equation object's list.

Continuing our example, Equation 4.1 creates an equation object with a identifying number 0, as it is our first equation object. Equation object 0 has two Identifiers, the primary Identifier A, and a secondary, generated Identifier, MX0.

In Equation 4.2, there are several Identifiers, both user and procedurally generated. The primary Identifier is A, which references a secondary Identifier generated during the conversion to computational engine code. There are also two user defined secondary Identifiers, B and C, and each reference there own backing Identifiers.

SECTION 4.1.4 Code Generator

When an equation object is created, the EquationHandler component interfaces with the mathematical recognizer and stores recognition results. To convert those results, which are stored in the MathML format, into a format that our mathematical engine can execute, we designed a series of components that convert from one format to another. These components make up the Code Generator component. The first subcomponent of the CodeGenerator converts MathML to basic, inline, code for the computational engine. This subcomponent was developed for an earlier project on mathematical imaging. As such, it is not geared specifically toward vectors, and can convert a broader range of MathML to code for our computational engine. Since

```
MX0 = [3;5;7];
  A = MX0;
```

Figure 4.3: Code for the computational engine generated by the Code Generator for Equation 4.1

MathML is stored in a tree structure, we traverse it in a recursive manner, and replace all vectors with secondary Identifiers. Here, we replace a diverse set of mathematical symbols with a smaller set of symbols. For instance, there are several asterisks and multiplication symbols that a user might input; our computational engine will only operate with a normal asterisk character, so this plethora of symbols must be replaced with the standard asterisk. This code will execute, but we want to change the code so that lends itself to easily creating animations.

We want to isolate individual operations, which will make it easier to create animations. The second subcomponent creates a parse tree from the generated code. This tree allows us to easily create lines of code that use a single operation with two operands. The final subcomponent of the Code Generator traverses the parse tree and generates several lines of code for the computational engine. Each line of code has at most one mathematical operation, so that an animation can easily be created from that line of code. Continuing our example, the MathML for Equation 4.1 is available in Listing 1. Once we have that MathML, we create two lines of code (Figure 4.3).

SECTION 4.1.5 Arrow

Variables are represented graphically by 3D arrows, as described in SECTION 3.2, which are implemented in the Arrow component. We implemented the arrow model as a cylindrical tube and a conical tip. The primary data members of the Arrow component are a name and a vector for model orientation and length. Arrow also provides methods to easily scale, rotate, and translate the model.

Scaling the Arrow increases or decreases the length of the cylinder, keeping the conical tip the same size, except when the Arrow is especially small. In such a case, we scale the cone down, too. Rotation changes the location of the endpoint of the model, while keeping the origin of the model at the graph's origin (0,0,0). Finally, translating an Arrow moves its origin, while keeping its scale and rotation. Each of these methods has support for animating the action and firing events upon completion of the animation.

We explored the coloring of Arrows while creating VectorPad. At first, we colored axis-aligned Arrows the same color as the axis's label; Arrows with other alignments were colored purple. For instance, an Arrow that aligned with the X-axis (e.g. $[5, 0, 0]^T$) would be colored blue, and one that aligned with the Y-axis (e.g. $[0, 7, 0]^T$), green. However, this color scheme meant that two Arrows along the same axis would be the same color, as well as two Arrows that were not aligned along a

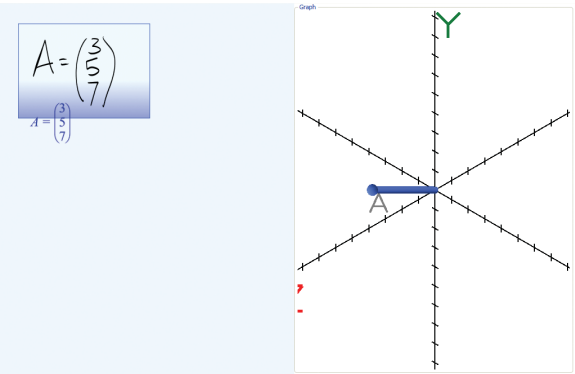

Figure 4.4: An Arrow has been created for Identifier A. This Arrow is colored blue, as is the box showing the bounds of the equation object.

primary axis, so we decided to choose a set of distinct colors from which to color the Arrows. We also maintain a second set of colors, all shades of pink, for Arrows created for primary Identifiers that are from mathematical equation objects.

After an equation object is created, the handwriting is converted to code for the computational engine, and the code is executed, an Arrow is created for the primary Identifier of the equation object. This Identifier gives the Arrow its name, such that an Identifier "A" creates an Arrow with the name "A". In Figure 4.4, we have created a blue Arrow for vector A. As discussed in Chapter 3, animations are created for mathematical operations. These animations move the Arrows for the secondary, user-generated Identifiers in an equation object and usually a new Arrow is created.

SECTION 4.1.6 Plane

The Plane component is a 3D geometric plane model used in one of the Cross Product animations. It is similar to the Arrow component in its available methods. The plane model is centered at the graph's origin and can be translated, rotated, and scaled. The color of the plane can also be changed, however, we always set the plane to be a light blue-green, highly transparent color. We think that this coloration allows the user to see the plane easily, without making it too hard to see the colorations of other models that may be behind the plane.

SECTION 4.2 Process

In this section, we will discuss the processes of VectorPad. The EquationHandler process utilizes its numerous subcomponents to create equation objects once a user has performed the gesture to recognize it. Once an equation object has been created, it is analyzed for mathematical operations, and the GraphicsHandler takes over, creating Arrows and animations as necessary.

SECTION 4.2.1 Creating An Equation Object

First, we will discuss what happens when a user writes upon the sketching area. Each stroke a user writes is tested to see if it is a gesture. Once it has been determined that it is not, the stroke is passed to the math recognizer. When the math recognizer has a result, the result is typeset and displayed on the sketching area. Once a user has written their equation, they lasso-select it and its component strokes are grouped into an equation object. From there, recognition results are pulled from the math recognizer for all the strokes. The recognized expression is then converted into MathML.

SECTION 4.2.2 Converting to Computational Engine Code

Next, the MathML is passed to the Code Generator component, where it is converted into code executable by the computational engine through a series of conversion steps. During these conversion steps, variables are identified and Identifiers are created for them. Some secondary Identifiers are also generated. The first subcomponent of the Code Generator recursively traverses the MathML tree and generates code for the computational engine. In the second subcomponent, this inline format is then converted back to a tree format to identify individual mathematical operations. Each line of code is recursively parsed and split into

```
Node0 = C - D;
A = B + Node0 ;
```

Figure 4.5: Computational engine code for Equation 4.3

operators and operands. These operators and operands form a tree structure, which maintains operator order of precedence, including the use of parentheses. Leaf nodes contain the names of Identifiers, while internal nodes contain mathematical operators.

Finally, the last Code Generator subcomponent recursively traverses the tree and new secondary Identifiers are added for each operation/operand set. This keeps the computational engine code easy to read by humans, and creates lines of code from which it is easy to create animations. For infix operations like addition and subtraction, code is generated by creating a statement in the form `<Identifier.Name>=<left operand><operator><right operand>` (see Figure 4.5). Other statements are represented differently in our computational engine's code format, such as the cross and dot products. Still, creating code for these operations from the tree structure is easy. During this process, a list of new and potential Identifiers is created, which will later be compared against a global list of Identifiers. Any Identifiers that are not already referenced by the equation object are added to its list, and any Identifiers not already in the global list are added to that list.

$$A = B + C - D \tag{4.3}$$

As mentioned in SECTION 4.1.2, the value of an Identifier can be either scalar or vector. The next step is to identify if an Identifier will be scalar or vector based upon any secondary Identifiers it references. This step does not consider any operations that secondary Identifiers might be performing, such as dot product, which references two vectors, but produces a scalar. As such, the value type will be updated after retrieving its calculated value from the computational engine.

Each equation object generally references several Identifiers, and a central list of Identifiers is also maintained in the EquationHandler component. An Identifier in an equation object maintains a list of code for the computational engine relevant to the Identifier and the equation object that contains it. Identifiers in the central Identifier repository contain lines of code relevant to the Identifier and all equation objects. When the user writes a mathematical equation object, VectorPad must query the central Identifier repository for each secondary Identifier. This query returns assignment statements for each secondary Identifier. The code is then ordered to insure proper execution. This ordering places assignment statements for secondary Identifiers at the beginning of the list, so that any Identifier that appears on the right hand side of a line of code has its assignment statement before that appearance.

SECTION 4.2.3 Executing Code

The last step before executing code is determining if an Identifier has previously been graphed. This information is used in adding Arrows to the graph. If the computational engine has a definition for an Identifier (based upon its name), then it is considered to have already been graphed. Once this has been done, the code is executed.

After gathering and ordering all the necessary code, VectorPad determines whether to execute the code or not. When the length operation is specified, VectorPad does not need to ask the computational engine for the answer, since each Identifier already has its value stored. For other operations the code will be executed. Finally, the code is passed through a special component to interface with the computational engine and execute it.

SECTION 4.2.4 Getting Results

Now that the code has been executed, the results must be retrieved from the computational engine. The first step in obtaining results from the computational engine is to determine the primary Identifiers for the equation object. From these Identifiers, we determine which ones the results are not already known for. These results can then be retrieved through the computational engine interface in the form

of an array. The dimensions of this array are examined to determine the Identifier's type, scalar or vector, and its value is stored in the Identifier.

Next, the code for these Identifiers is examined to determine if it involves a simple definition statement or some mathematical operation. In the case of a definition, an Arrow is created upon the graph with its origin at (0,0,0) and its end at the vector value of the Identifier. Each arrow is assigned a color upon creation to help differentiate the arrows. The Arrow is added to the equation object and the object is highlighted with the arrow's color. When an Identifier is the result of a mathematical operation, an animation is created for it. The component arrows of the operation are animated to show the action of the operation, and a result arrow is added to the graph after the animation finishes. Each arrow is also labeled with the name of the Identifier for which it represents.

SECTION 4.2.5 Graphing

For mathematical operations like addition, subtraction, or scalar multiplication, there will be three Identifiers in an equation object, one primary, and two secondary. The Identifiers for each line of code in the equation object are retrieved, and a data structure with information needed to animate the operation is created with these Identifiers, as well as the equation object, and the line of code. This data structure will be used to create Arrows for assignment equation objects and animations for

mathematical equation objects. It is added to a queue of pending graph operations.

Whenever an animation is not currently being executed and there are items in the queue, the head of the queue is serviced. The data members of the animation data structure are examined and the kind of animation is determined to be an assignment animation or a mathematical operation animation.

Graphing Arrows

In an assignment animation, an Arrow is created and added to the graph. Two things are needed to graph an Arrow, an equation object and its primary Identifier. First, we look up if there is an existing color for the Identifier's name in a color lookup hash table. If so, an Arrow is created with the same name as the Identifier and the specified color. If there is no preexisting color for that name, an Arrow is created without a specified color. During the creation of the Arrow, its color is chosen as the next available color in a list of colors. A new entry is added to the color lookup hash table with the Identifier's name as the key and the new color as the value. In both cases, the length and direction of the Arrow is specified by the Identifier's vector value.

The newly created Arrow is then added to a general list of models, to the equation object's list of Arrows, and to the graph. When it is added to the graph, it appears in its final position, with no animations showing its creation. Initially, VectorPad

animated the creation of an Arrow, showing it "growing" and rotating to its final size and rotation, but test users did not generally like this behavior, so it was removed. Once the Arrow has been added to the graph, the graph is zoomed so that the Arrow mostly fills it up (see SECTION 4.2.6). At this time the axes are also extended if the Arrow is longer in any direction than any axis.

Creating Animations

There are several objects needed to create an animation: an equation object, a line of code, its primary Identifier, and its secondary Identifiers. First, we look at the code and find the mathematical operation in it. This operation determines the kind of animation created: addition, subtraction, scalar multiplication, scalar division, cross product, or dot product. In general, animations move the Arrows associated with the secondary Identifiers for a line and leave behind shadow copies to show the user where the Arrows originally were located. Again, the axes are extended to be longer than any Arrow. After the animation has completed, the Arrows reset to their normal position and orientation, and the camera is zoomed to fit all the Arrows on screen.

SECTION 4.2.6 Fitting the Camera

When a new object is added to the graph, or an object is updated, VectorPad zooms the camera to ensure that certain models are visible on the graph and that they are maximized visually. The models that we are concerned with are Arrows, Planes, and text labels that are not used as labels on the axes. Fitting the camera to show all the models on the graph is a complicated procedure. For each model, its 2D coordinates on the graph must be computed, which is simple for Arrows and Planes, but problematic for text labels. For certain objects, one of the libraries we use provides a method for obtaining the 2D coordinates for an object in a 3D viewport.

The text class that we use comes from the Media3D library by Charles Petzold (see SECTION 4.4). Although this class, WireText, is ultimately derived from the same class as Arrow and Plane, this method does not work correctly, so we had to find a different way to do this. First, we tried to use a mathematical approach by using a class found in the another library we used (see SECTION 4.4). This approach takes a viewport and returns a matrix that can be used to transform from 3D coordinates to the viewport's 2D coordinate system. We first find the six corners of the text's 3D bounding box, convert them to 2D, and then create a 2D bounding box out of the points. The results for this were not satisfactory, so another avenue was pursued. The text class has a collection of 3D points that are used to create its component letters. We took this collection, converting all the points to 2D as in our

previous method, and found the top-most, bottom-most, left-most, and right-most points to create the bounding box. This methodology gave us quite satisfactory results.

Once we have the 2D bounding boxes for all the relevant models, a bounding box containing all the models is created. A fit test is performed in which the bounds of the box are compared with the graph's rendered dimensions to determine if the models are all on screen, and if so, if the camera can zoom in more. If we determine that more zoom is needed, we iteratively zoom in by 10% and perform the same fit test until it is found that we have zoomed in too much. At that point we iteratively zoom out by 1% and perform the fit test until we have zoomed out too much, or we have zoomed out 10 times. Lastly, we zoom out 5% to give a little bit of room around the edges of the graph. If the original fit test tells us that the models do not fit on the screen, the same sort of process is performed, but we first zoom out by 10%, then zoom in by 1%, and again zoom out 5%.

We felt that a user might want to zoom to the models for a specific equation object. When the InkHandler recognizes a double-tap in the bounds of an equation object, the GraphicsHandler starts the camera fitting process, but only with the models used in the equation object. Also, if the user wants to invoke fitting for all the models on the graph, they can double-tap outside of any equation object.

SECTION 4.2.7 Preventing Text Occlusion

With several Arrows in the graph area, text labels are fairly likely to occlude each other. VectorPad prevents this from occurring by moving one of the texts when two texts occlude. One issue that we encountered while trying to work with WireTexts is that their bounding boxes are incorrect. To solve this problem, we first tried to estimate the bounding box of a given text by calculating its approximate length, which is the number of characters times the font size. We also used the font size to calculate its approximate height. The final dimension of the bounding box, depth, was given an arbitrarily small value. Since we know the direction in which the text was written, we could calculate the approximate bounds from these numbers.

At first we tried to prevent occlusions by comparing the 3D bounding boxes for the two WireTexts, and moving one of the texts until the two bounding boxes no longer intersected. However, this still allowed occlusions when one text was in front of the other, but their bounding boxes did not intersect. This could be fixed by giving the bounding boxes an essentially infinite depth, but we did not pursue this because our approximation of the text's bounds was otherwise too poor.

The approach that worked best is the same as in SECTION 4.2.6, taking the text's point collection, converting it to a 2D bounding box, and comparing the bounding boxes. If the boxes intersect, one text is iteratively moved until the bounding boxes

no longer intersect. The direction in which the text is moved is the normal of its original position (Equation 4.4). The text is moved in increments of one-quarter the norm of the direction (Equation 4.5). Each time a text label is added to the graph, each text is checked against each other text. The same occurs every time the graph is rotated. It is important that each text's bounding box is calculated using its original intended position for the first test of each testing session, otherwise texts will start to stray as the graph is rotated.

$$\vec{p}_{norm} = \frac{\vec{p}}{\| \vec{p} \|}. \tag{4.4}$$

$$\vec{d} = \frac{\vec{p}_{norm}}{4}. \tag{4.5}$$

SECTION 4.2.8 Updating An Equation

Updating an equation starts from two actions, erasing an ink stroke, or adding an ink stroke in the bounds of an established equation object. After an update to an equation object, any equation objects that are dependent upon it are updated. First, the graphics objects associated with the dependent equation object are removed from the graph. Next, we gather code from the Identifiers that were updated and add it to the dependent equation object's code. The rest of the steps

are similar to creating an equation object. This update process may have to occur several times when there is a chain of dependent equation objects.

Erasing

Erasing is performed using a scribble erase gesture. When the InkHandler gesture recognizer determines that a scribble was performed, we test to see what strokes it intersects with. Any stroke that intersects with the gesture is marked to be removed from the sketching area and from any equation object that contains it. Once the strokes to be removed have been collected, they are passed to the EquationHandler, where each stroke is checked against each equation object to determine which equation object owns the stroke. Once the owning equation object is determined, the stroke is removed from the object. The last step in updating an equation object when a stroke is deleted is checking if all the strokes from an equation object have been removed. If so, the equation object and its Arrows are removed from VectorPad, and any equation objects that are dependent upon the equation object's primary Identifier are updated.

We consider an equation object to be orphaned when it is a dependent equation object, and the equation object upon which it depends has been erased. An orphaned equation object does not have any arrows on the graph, because its primary Identifier is considered to have an unknown value. When we determine that

there is an orphaned equation object, its strokes are colored a deep red and the user is notified that it cannot be graphed until the missing dependency has been resolved.

Adding

When a new stroke is made on the sketching area, the InkHandler performs a hit test against all the recognized equation objects to determine if the stroke belongs to a new, unrecognized equation object, or to an existing equation object. If the stroke falls 95% within the bounds of an equation object, it is added to the equation object and new recognition results are obtained for the equation object's strokes. The equation object clears its list of Identifiers, since the user can change them with the new stroke. Next, the models (Arrows, text labels, etc.) associated with the equation object are removed from the graph. After that the same process as creating an equation object is performed.

Sometimes a user might want to update an equation object, but not have enough room in the defined bounds for the object. To solve this problem, we allow the user to redefine the bounds of an equation object by performing another lasso-tap gesture. Once the InkHandler has recognized a lasso-tap, the EquationHandler is passed the new bounds and any containing strokes, as well as the equation object to be updated. The equation object's bounds and strokes are then updated, followed by the regular update procedure.

SECTION 4.3 Math Recognizers

We have used two different mathematics recognizers in VectorPad, initially VectorPad used a mathematical recognizer from Microsoft [21], and later we used one in the starPad SDK (see SECTION 4.4). The Microsoft recognizer performed similarly to the math recognizer in starPad in terms of recognition and the symbols which it recognized. MathML could also be obtained from its recognition results. We decided to change to starPad because it provided recognition typesetting and a stronger application framework. Additionally, we are able to edit and change starPad, something we could not do with the Microsoft math recognizer.

SECTION 4.4 Implementation

VectorPad was created using Microsoft's C# language, .NET Platform 3.5, and Windows Presentation Foundation (WPF) 3.5. VectorPad also uses MATLAB as its computational engine through a managed dll. Windows Presentation Foundation was used to create VectorPad because it provides numerous tools for pen-based interfaces and is easy to use to create user interfaces. WPF also has 3D graphics and animation classes. VectorPad uses MATLAB for its math computation due to its link with an earlier project. Since the project used MATLAB for its mathematical computation, we chose to continue using it, even though we can

calculate the vector mathematics quite easily. This also makes it easy to expand upon the mathematical capabilities of VectorPad, only requiring that we can write MATLAB code for any operation.

Several SDKs and libraries were used in VectorPad, including starPad from Brown University [20], Charles Petzold's Media3D library [22], and the open-source 3DTools library [1]. StarPad provides ink-related classes, including character and math recognition, conversion of recognition results to MathML, and typesetting of math results. Our gesture recognition and alternates menu are also provided by starPad. StarPad is programmed using C# and F#. The Media3D library was used for its 3D axes and text classes. The text class was used extensively to provide text labels for VectorPad's graph. Finally, the 3DTools library provides a trackball implementation.

The Arrow and Plane components derive from WPF's ModelVisual3D class. Scaling, rotating, and translating the arrow and plane models are performed using subclasses of WPF's Transform3D (such as ScaleTransform3D and RotateTransform3D). These transformations are applied to the model to get the desired effect and can also be animated.

```
TranslateTransform3D translate = new TranslateTransform3D();
translate.OffsetX = 5;
translate.OffsetY = 0;
translate.OffsetZ = 0;
```

Figure 4.6: An example of a TranslateTransform3D

SECTION 4.4.1 Animations

The animations for mathematical operations are provided by a number of classes, which all derive from a base MathAnimation class. These classes are event driven and use the rotate, translate, and scale methods of the Arrow and Plane classes. The animations in the rotate, translate, and scale methods use WPF's DoubleAnimation class. This class is applied to the various Transform3Ds applied to the models.

Suppose we want to move an Arrow five units in the positive direction along the X axis. This requires that we use a TranslateTransform3D and set it to move five units along the X axis (see Figure 4.6). Next, we create a DoubleAnimation that specifies that the animation will go from the Arrow's current X position to its new X position (Figure 4.7). Finally, we apply the animation to the transform and the transform to the Arrow's model (see Figure 4.8). In this example, we set the animation to have a duration of two seconds, but in VectorPad, the duration's length is specified by a slider, as mentioned in SECTION 3.3.

We wanted to allow the user to pause animations, but this proved to be problematic. DoubleAnimation (and its related classes) cannot be paused or

```
DoubleAnimation xAnimation = new DoubleAnimation();
xAnimation.From = origin.X;
xAnimation.To = origin.X + 5;
xAnimation.Duration = new Duration( TimeSpan.FromSeconds( 2 ) ) );
```

Figure 4.7: An example of a DoubleAnimation

```
translate.BeginAnimation( TranslateTransform3D.OffsetXProperty,
xAnimation );
this.Content.Transform = translate;
```

Figure 4.8: Applying a DoubleAnimation to a TranslateTransform3D

stopped once started when invoked using the BeginAnimation method. WPF provides other forms of animation, such as AnimationClocks and Storyboards, but we were not able to make either of these methods work correctly.

CHAPTER 5: INFORMAL USER STUDY

SECTION 5.1 Informal User Study

We conducted an informal, preliminary user study of VectorPad to obtain some feedback on its design and user interface, with six participants. Participants were generally versed in vector mathematics and linear algebra. First, we demonstrated the use of VectorPad to the participants, showing them what mathematics VectorPad recognizes and how to use the various gestures. Participants were then free to play with VectorPad, although we asked them to use all the different animations. Once participants were done using VectorPad, we asked them questions about their experience, such as which cross product animation they preferred, whether they liked each animation, and their comments on various aspects of the user interface.

SECTION 5.1.1 Results

We asked participants whether they found the different arrow colors useful. While most participants said yes, one participant replied that he did not use the colors to identify the different arrows, but rather the labels. However, he did say that he liked having the arrows different colors. Another participant said that she "liked the shaded rectangles with the same color as the arrow," because they helped to distinguish which arrow represented which Identifier. Similarly, one participant responded that she would like the labels to be the same color as the arrow, which she said would make it easier to identify which label was for which arrow.

Participants liked how the graph zoomed in on arrows and animations, as it gave them a better view of the models. Most participants never turned this feature off, even though they were made aware of it during the demonstration and during their usage of VectorPad. We think that this shows that most users would not turn off the feature, at least initially; in other words, such a feature would have to be designed so as not to be problematic, since users won't turn it off, even when it causes them problems.

Five of the six participants responded that the animation for the addition operation made sense and was clear. The last participant responded that he understood it, but that it did not fit his mental model for vector addition. While the participants

felt that the addition animation was clear, there were some problems with the subtraction animation. The primary issue with the subtraction animation was with the second arrow's rotation to its negative position. This rotation is animated while the arrow is translating to the end of the first arrow and consequently three participants felt that it was less than clear what was occurring. One participant commented that it could be made clearer by translating and rotating at different times, while another said that rotating the arrow was not needed.

One participant who had a minimal background in vector mathematics found the second scalar multiplication animation, the stacking animation, easier to understand, as it better conveyed how the scalar operand affected the vector operand. Participants did not display much of a preference for one scalar multiplication animation over the other, but several did express that they felt that the stacking animation would be better for novices who did not understand the scalar multiplication operation.

For cross product, most of the participants preferred the animation that showed the plane made by the two operands. One participant said that it "makes sense" and shows the way people usually visualize it. However, one participant felt that the plane was not needed most of the time and that it should be optional. Most participants seemed ambivalent about the dot product animation. Showing the scalar projection was not the most natural idea for the participants. Comments

mainly focused upon the text results, with one participant commenting that the "text could make it cluttered quickly."

Participants grouped into two categories based upon their use of the sketching area. Some participants used the entire sketching area, writing many equations before clearing. The other group would only write a few equations before clearing. With the first group, some participants felt that the sketching area was too small and wanted a way to expand the sketching area. Another interesting trend that we noticed was that some participants wrote several equations before going back and lasso-tapping each one. This effectively causes the animations to form a queue, which can be distracting for the user.

We found that the participants often had problems when writing a dot product, because the gesture recognizer would recognize the "$\cdot$" as a tap gesture. Another issue with the dot product was that the dot would be incorrectly recognized and the proper symbol was not provided as an alternate (this was also true for "$\otimes$"). The combination of these two problems made it very difficult for participants to write mathematics using the dot product. One participant suggested that we provide a different symbol for the dot product, but we feel that other approaches would be better. First, we want the experience of writing mathematics to be as familiar as pen and paper as possible, and introducing a different symbol would be antithetical to that goal. Second, we would like to explore automatic grouping of strokes into

equations (see SECTION 6.1).

Another gesture-related issue came from the symbol for the cross product operation, "⊗." Writing this symbol as three strokes, two for a "×" and then a "◯" results in the recognition of a "×" followed by the lasso gesture. While most participants wrote the "⊗" symbol as a "◯" followed by a "×," this was a problem for one participant. In order to keep problems like this from occurring, we must be mindful of the symbols and gestures used.

Another issue we found during the study was that models would clip during animations, which we did not take into account when we fit the camera to the models. Several participants said that this made it hard to tell what was occurring when large portions of the models were clipped from view. Similarly, when an arrow is directly pointing into the camera, it is invisible, which was confusing to some participants. We noticed that the participants rarely wrote vectors that were entirely or predominately negative and usually they kept the graph orientation such that arrows were either perpendicular to the camera or pointing towards it.

We had several suggestions for VectorPad from the participants. Some participants suggested that we show the answer to each equation on the sketching area in text form. Currently, to determine the value of a vector, a user must click on its arrow to see the information box that pops open. We explored having the vector values on the sketching area in early versions of VectorPad, but removed it after negative user

feedback.

Another suggestion was to allow the user to create multiple equations with a single gesture. Similarly, one participant suggested adding a mechanism to load a set of vectors from a file to lessen the amount of writing needed.

From the user study we found that we should have provided a separate means for the user to identify the positive and negative directions on the number lines. While each axis is labeled on its positive side with the axis's name, this was not immediately clear to the participants. Additionally, when the graph was zoomed in far enough, the labels were not visible, so users would have no way to identify the positive and negative directions for the axes. Similarly, it can be hard to visually identify or verify the length of any arrow along any axis, particularly for vectors with non-zero components along all three axes. One participant suggested that showing the X, Y, and Z components for each arrow would help users to perform such visual identification.

Clutter was a major problem for many of the participants. Moving the text results for the dot product and length operations out of the 3D space of the graph and onto a transparent overlay that can be toggled on or off seems like a good way of removing some of the current clutter. Additionally, other text information, like number line scale, direction, and axis name would best be added in a way that allows the user to change its visibility.

CHAPTER 6: FUTURE WORK AND CONCLUSION

SECTION 6.1 Future Work

There are numerous areas in which we could improve and extend upon VectorPad. These include support for row vectors and matrices; support for more operations, such as vector calculus, vector fields, and matrix-vector operations; and providing a way to batch load equations. The user study also indicates that we need to refine our tap recognition or find a gesture more suitable to vector mathematics.

SECTION 6.1.1 Finding A Good Camera Position

We would like to explore a way to determine an appropriate camera direction and position for a vector or set of vectors. As we mentioned in SECTION 5.1, sometimes the models could be invisible due to their position relative to the camera, so we need to move the camera when we determine that a model is directly facing it. Other times an arrow or animation does not present well from the current camera position

and we would do well to move the camera to give the user the best view possible. Additionally, throughout the course of an animation, the best angle of view will change. One possible way to determine an appropriate viewing position would be to calculate the position from which the target arrows have the largest two-dimensional bounding box. To calculate the appropriate camera location, we could split the relevant vectors into two groups, sum each group, find the cross product for the two sums, and move the camera to an appropriate point along that vector.

Animating the camera as a visualization animation occurs, particularly with multiple animations for complex equations, could be quite beneficial for the user, as it would help them to see the animations in their entirety. At the beginning of an animation, the user would benefit from seeing the arrows up close; for large result vectors we might want to zoom out to show the result as it is formed. Extreme care needs to be taken so that the camera does not shake or move strangely as the arrows animate. Additionally, the user needs to be able to disable automatic camera movements.

SECTION 6.1.2 Scaling Issues

While animating the graph's camera will help the user to see the whole of an animation, it does not help the user to see the details that are visible while zoomed in and while zoomed out. With both long and short arrows on the screen at the

same time, a user cannot easily examine both without zooming in and out repeatedly. The comparison window can help the user with situations like these, since they can position one graph to examine the smaller arrows, and the other for the longer arrows. However, this does not help with animations, since the comparison window does not support them. One solution would be to add animations to the comparison window and change the comparison window so that it can show the same graph state on each graph.

Another solution is add "picture-in-picture" to the graph. A smaller graph could appear in one corner of the main graph. This subgraph could be controlled independently of the main graph, or the two could be linked. Having "picture-in-picture" would allow the user to see an animation from multiple views without having to open a new window, as with the comparison window.

SECTION 6.1.3 Automatic Equation Segmentation

We are interested in whether VectorPad could support automatic segmentation of strokes into equations. The mathematics recognizer in starPad already supports grouping into "ranges," which we might be able leverage into equation segmentation. We would have to maintain support for a grouping gesture so that users could correct errors in segmentation.

One of the most serious problems that arises with automatic equation recognition is

determining when to create a visualization for an equation. While writing an equation, there may be several times when the currently written strokes form a valid equation. Creating visualizations at each of these points would be quite distracting for most users. To accomplish automatic equation recognition, we would need to add a component to verify that the mathematics involved in an equation was valid and relevant to linear algebra, to reduce needless computations and visualizations.

A partial solution to this problem can be introduced through the use of a timer. After the user has paused writing for a sufficient amount of time, we would assume that the user has finished writing their equation and an equation would be created. Similarly, when the mathematics recognizer decides that a new range of writing has been created, this could be a signal that we should create a visualization. Combining a timer, a mathematics verifier, and detector of new ranges could conceivably create a reasonable automatic equation recognition system.

In another approach to automatic equation recognition, we could reduce the number of gestures a user has to perform, as discussed in SECTION 5.1. We could allow the user to make a single gesture to recognize multiple equations and perform automatic segmentation upon all the writing on the sketching area. This introduces at least one problem; the user will now have a potentially long chain of animations to watch and try to understand. Introducing a user-configurable pause between animations might help to alleviate this problem, as would a pause button.

SECTION 6.1.4 Equation Dictionary

We like the idea of an equation dictionary or storage area, where a user can drag their equations to clear up space on the sketching area. This would solve the sketching area space issue, and help lessen the writing burden on the user when working with large numbers of vectors. We envision each dictionary entry as having a visual representation of the equation using a thumbnail-like static graph and information about the vectors involved. A user could drag equations from the sketching area to the graph to free up sketching space, and back to the graph for editing.

SECTION 6.1.5 Obtaining Mathematics from Visualizations

A potentially interesting direction for VectorPad is to allow the user to manipulate arrows and for VectorPad to create equations for the arrows. A user could move an arrow and VectorPad would look at the position in which it was left to determine some sort of mathematical definition for its vector. This sort of mathematical determination system can be quite complex, depending upon the movements allowed and the system constraints. Obtaining a definition for a single arrow with its origin at the graph's origin is a simple task. Determining how that definition relates to other arrows on the screen is a considerably more complex problem. A

single vector might be defined by a large number of possible equations using different mathematical operations. A system of constraints can help to reduce the complexity of determining equations and reduce the set of possible equations to a small number of equations that a user can reasonably examine. Constraining the system to operations involving only the vectors on the graph reduces the complexity of the problem, but still leaves a large number of possible equations. We would not advocate having a system in VectorPad that does not constrain to using only already defined vectors.

One way in which this might be a tenable system would be to allow the user to constrain the type of math which defines the vectors. For instance, when considering only existing vectors as operands, constraining the system to only look at addition and subtraction greatly reduces the number of possible equations, though it does not reduce it to only one possible equation. Reduction to addition and subtraction falls apart when axis-aligned unit vectors (particularly all three) appear upon the graph, as any vector can be produced in any number of ways. When scalar multiplication/division is allowed, along with addition and subtraction, things become much more complex, since each vector can be multiplied by an arbitrary scalar. One way to simplify this problem is to introduce a constraint on the scalars involved. A user could define that they only want scalars that are rational numbers, integers, or even positive integers. This would force the mathematical determination system to produce a smaller set of results.

Another constraint that must be incorporated into the system in order for its results to be meaningful is forcing the system to only produce "simple" results. This means that VectorPad would not give results, which for instance, differed only in being multiplied by some number. A simple result is also one that cannot be further simplified. For example, Equation 6.1 can be further simplified, so it would not be returned as a possible equation.

$$C = A + A + B \tag{6.1}$$

Another possible constraint for the mathematical determination system would be on the number of operands in an equation. With only three operands allowed in an equation, and only a relatively small number of operands, it would be quite simple to test all the possible combinations of operands, if we had to do so. More likely than not, we think that a user would be looking for a simpler answer or an answer using specific operands. This brings us to another constraint, which operands the system can use to find equations. In our minds, a user will often be looking for an answer that comes from a few specific vectors, vectors that might not be the only ones upon the graph. A system to specify those vectors would help to reduce the equation set.

We could also incorporate different levels of constraints for different types of users. A user who is learning linear algebra could have built in constraints, defined to produce results that are simple and easy to understand. Constraint sets could even

be created for specific learning exercises, for instance, allowing only equations that use the cross products when the user is learning about the cross product operation. More advanced users could enable or specify their own constraints.

As computer systems increase in computational power, we can determine a larger number of mathematical equations for any given arrow in a short period of time. We could then cull these equations using the constraint system to present the user with a small list of possible equations. This could be problematic as the system might not produce viable results without taking the constraints into account when generating equations. A system that works with the constraints while generating possible equations would be a better approach.

SECTION 6.2 Conclusions

We have presented VectorPad, a tool for dynamic visualization of 3D vector mathematics. VectorPad provides support for a number of mathematical operations using vectors, including addition, scalar multiplication, and cross product, and visualizes these operations using a 3D graph. Vectors are represented on the graph by arrows and operations are animated to illustrate their effects. Users input vectors using pen-based techniques and VectorPad recognizes the user's handwriting using character and mathematics recognition. Vectors are visualized using three-dimensional arrows and mathematical operations are animated using these

arrows. Users are able to control the graph by rotating it and zooming in and out. VectorPad also has controls to allow the user to replay animations, as well as play them at different speeds.

We also conducted an informal user study, where we found that the animations we implemented were generally liked by the participants. We also found that the participants wanted tools to help them to further understand and view the visualizations, such as automatic camera rotation. We identified a number of areas in which VectorPad could improve its visualizations, including the subtraction and dot product visualizations. We think that our results show that users want to be able to explore vector visualizations both during and after the period of animation. Finally, we presented a variety of directions that we could explore in future work.

APPENDIX: MATHML EXAMPLES

```
<math>
    <mrow>
        <mi>A</mi>
        <mo>=</mo>
        <mrow>
            <mo>(</mo>
            <mtable>
                <mtr>
                    <mtd>
                        <mn>3</mn>
                    </mtd>
                </mtr>
                <mtr>
                    <mtd>
                        <mn>5</mn>
                    </mtd>
                </mtr>
                <mtr>
                    <mtd>
                        <mn>7</mn>
                    </mtd>
                </mtr>
            </mtable>
            <mo>)</mo>
        </mrow>
    </mrow>
</math>
```

Listing 1: MathML for Equation 4.1

```
<math>
  <mrow>
    <mi>A</mi>
    <mo>=</mo>
    <mrow>
      <mi>B</mi>
      <mo>+</mo>
      <mi>C</mi>
      <mo>−</mo>
      <mi>D</mi>
    </mrow>
  </mrow>
</math>
```

Listing 2: MathML for Equation 4.3

LIST OF REFERENCES

[1] 3d tools for the windows presentation foundation, July 2009. `http://www.codeplex.com/3DTools`.

[2] Erdat Cataloglu. Open source software in teaching physics: A case study on vector algebra and visual representations. *The Turkish Online Journal of Educational Technology - TOJET*, 5(1), January 2006.

[3] Kam-Fai Chan and Dit-Yan Yeung. Pencalc: a novel application of on-line mathematical expression recognition technology. In *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, pages 774–778, 2001.

[4] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-d rotation using 2-d control devices. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 121–129, New York, NY, USA, 1988. ACM.

[5] David Crowe and Hossein Zand. Computers and undergraduate mathematics 3: Internet resources. *Computers & Education*, 35(2):123–147, 2000.

[6] Ray Eason and Garrett Heath. Paintbrush of discovery: Using java applets to enhancemathematics education. *Primus*, 14(1):79–95, 2004.

[7] Richard Fateman. Handwriting + speech for computer entry of mathematics. `http://www.eecs.berkeley.edu/~fateman/papers/voice+hand.pdf`, 2004.

[8] A. Gomi, R. Miyazaki, T. Itoh, and Jia Li. Cat: A hierarchical image browser using a rectangle packing technique. In *Information Visualisation, 2008. IV '08. 12th International Conference*, pages 82–87, July 2008.

[9] Mark Green and Robert Jacob. Siggraph '90 workshop report: software architectures and metaphors for non-wimp user interfaces. *SIGGRAPH Comput. Graph.*, 25(3):229–235, 1991.

[10] L. Hesselink, F.H. Post, and J.J. van Wijk. Research issues in vector and tensor field visualization. *Computer Graphics and Applications, IEEE*, 14(2):76–79, Mar 1994.

[11] Hannes Kaufmann, Dieter Schmalstieg, and Michael Wagner. Construct3d: A virtual reality application for mathematics and geometry education. *Education and Information Technologies*, 5(4):263–276, 2000.

[12] Matthias Kawski. Dynamic visualization in advanced undergraduate courses. In *6th Southern Hemisphere Symposium on Undergraduate Mathematics Teaching*, pages 91–100, 2007.

[13] George Labahn, Edward Lank, Scott MacLean, Mirette Marzouk, and David Tausky. Mathbrush: A system for doing math on pen-based devices. In *DAS '08: Proceedings of the 2008 The Eighth IAPR International Workshop on Document Analysis Systems*, pages 599–606, Washington, DC, USA, 2008. IEEE Computer Society.

[14] George Labahn, Scott MacLean, Mirette Marzouk, Ian Rutherford, and David Tausky. Mathbrush: An experimental pen-based math system. In W. Decker, M. Dewar, E. Kaltofen, and S. Watt, editors, *Dagstuhl Seminar Proceedings, Challenges in Symbolic Computation Software*, October 2006.

[15] Joseph J. Laviola, Jr. *Mathematical sketching: a new approach to creating and exploring dynamic illustrations*. PhD thesis, Brown University, Providence, RI, USA, 2005.

[16] Joseph J. LaViola, Jr. and Robert C. Zeleznik. Mathpad^2: a system for the creation and exploration of mathematical sketches. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 432–440, New York, NY, USA, 2004. ACM.

[17] Eberhard Malkowsky. Visualization and animation in mathematics and physics. In *Proceedings of Institute of Mathematics of NAS of Ukraine*, volume 50, 2004.

[18] Michael McCabe and Jenny Watson. From mathedge to mathwise: The cutting 'edge of interactive learning and assessment in mathematics. In *3rd International Conference on Technology in Mathematics Teaching*, October 1997.

[19] Microsoft: Math. Computer program, July 2009. `http://www.microsoft.com/math`.

[20] Timothy Miller. Microsoft center for research on pen-centric computing: starpad sdk, July 2009. `http://pen.cs.brown.edu/starpad.html`.

[21] Marko Panic. *Math Handwriting Recognition in Windows 7 and Its Benefits*, pages 29–30. Lecture Notes in Computer Science. Springer Berlin, 2009.

[22] Charles Petzold. *3d programming for windows®: three-dimensional graphics programming for the windows presentation foundation.* Microsoft Press, Redmond, WA, USA, 2007.

[23] Frits H. Post and Jarke J. Van Wijk Y. Visual representation of vector fields: Recent developments and research directions. In *Scientific Visualization – Advances and Challenges*, pages 367–390. Academic Press, 1994.

[24] W.J. Schroeder, C.R. Volpe, and W.E. Lorensen. The stream polygon-a technique for 3d vector field visualization. In *Visualization, 1991. Visualization '91, Proceedings., IEEE Conference on*, pages 126–132, 417, Oct 1991.

[25] Ben Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[26] Elena Smirnova and Stephen M. Watt. A context for pen-based mathematical computing. In *in Proceedings of the 2005 Maple Summer Conference*, pages 17–21, 2005.

[27] Elena Smirnova and Stephen M. Watt. Communicating mathematics via pen-based computer interfaces. In *Proc. 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, (SYNASC 2008)*, pages 9–18. IEEE Computer Society, September 2008.

[28] Steve Smithies, Kevin Novins, and James Arvo. Equation entry and editing via handwriting and gesture recognition. *Behaviour and Information Technology*, 20(1):53–67, January 2001.

[29] Alexandru Telea and Jarke J. van Wijk. Simplified representation of vector fields. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 35–42, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[30] Joseph Tront. Vectorpad. Computer program, July 2009. `http://filebox.ece.vt.edu/~jgtront/tabletpc/vector_pad.html`.

[31] Andries van Dam. Post-wimp user interfaces. *Commun. ACM*, 40(2):63–67, 1997.

[32] Robert Zeleznik, Timothy Miller, Chuanjun Li, and Joseph J. Laviola, Jr. Mathpaper: Mathematical sketching with fluid support for interactive computation. In *SG '08: Proceedings of the 9th international symposium on Smart Graphics*, pages 20–32, Berlin, Heidelberg, 2008. Springer-Verlag.