Criterion C Development

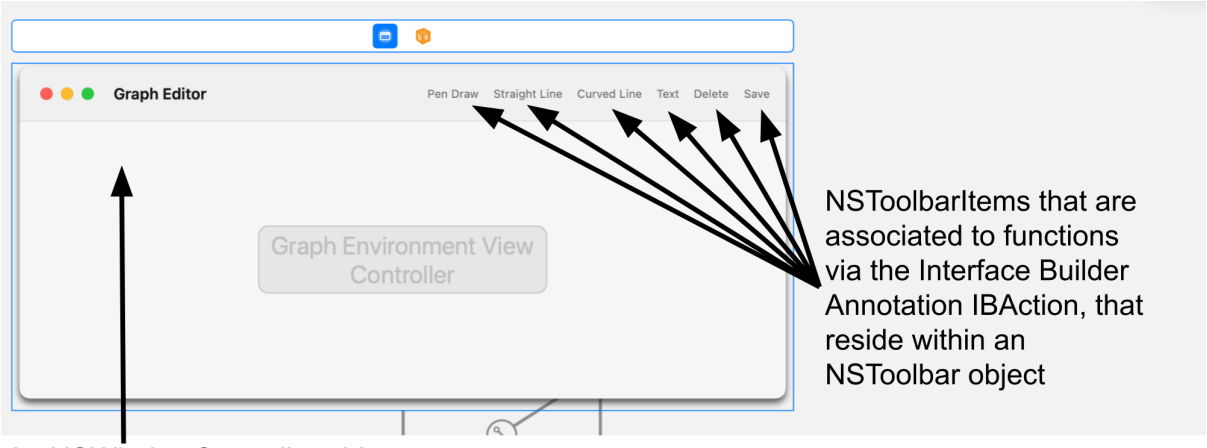
This program utilises Apple's Xcode IDE, version Version 13.0 (13A233), with a version of Swift 5.5, and Apple's Cocoa Libraries and Swift Standard Libraries (see appendix for more information).

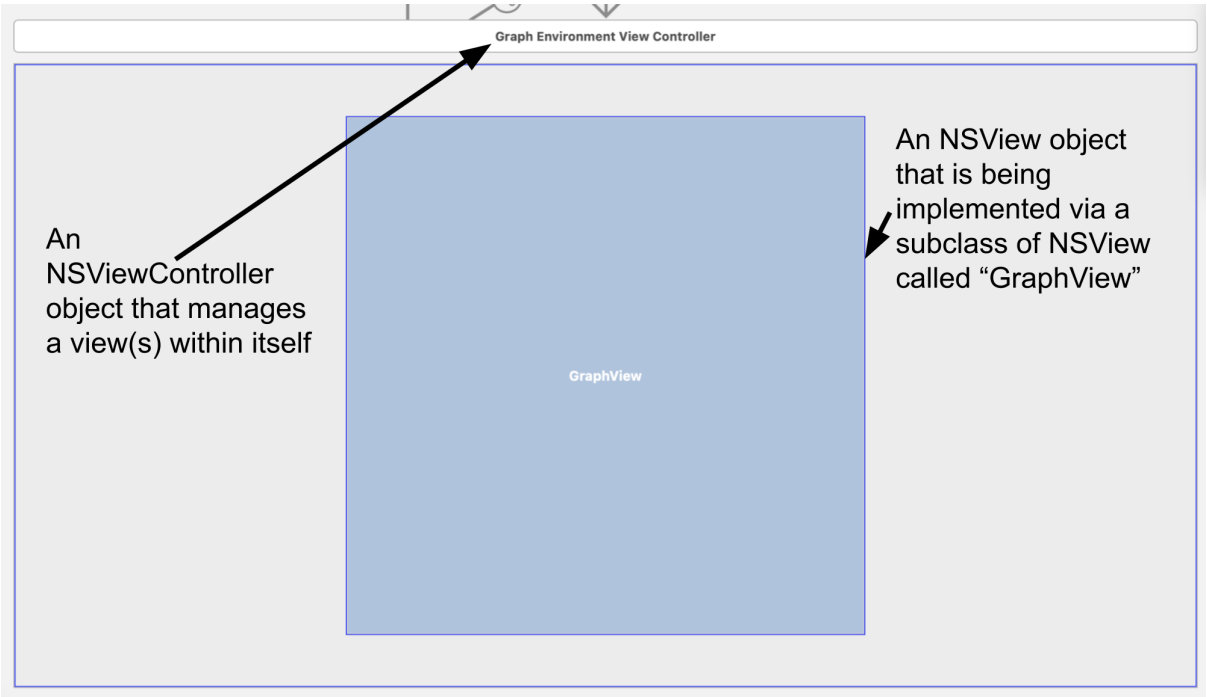
Key Techniques

1. Graphical User Interface
    a. Interface Builder Connections
    b. Drawing lines
    c. Drawing text
2. Programming Paradigm Usage
    a. Object Oriented Programming and Protocol Oriented Programming
        i. Forced Downcasting
        ii. Inheritance
        iii. Polymorphism
    b. Functional Programming
3. Computed Properties
4. Data Structures
5. Error Handling
    a. Optional Handling
6. File Writing
7. User Defined Methods

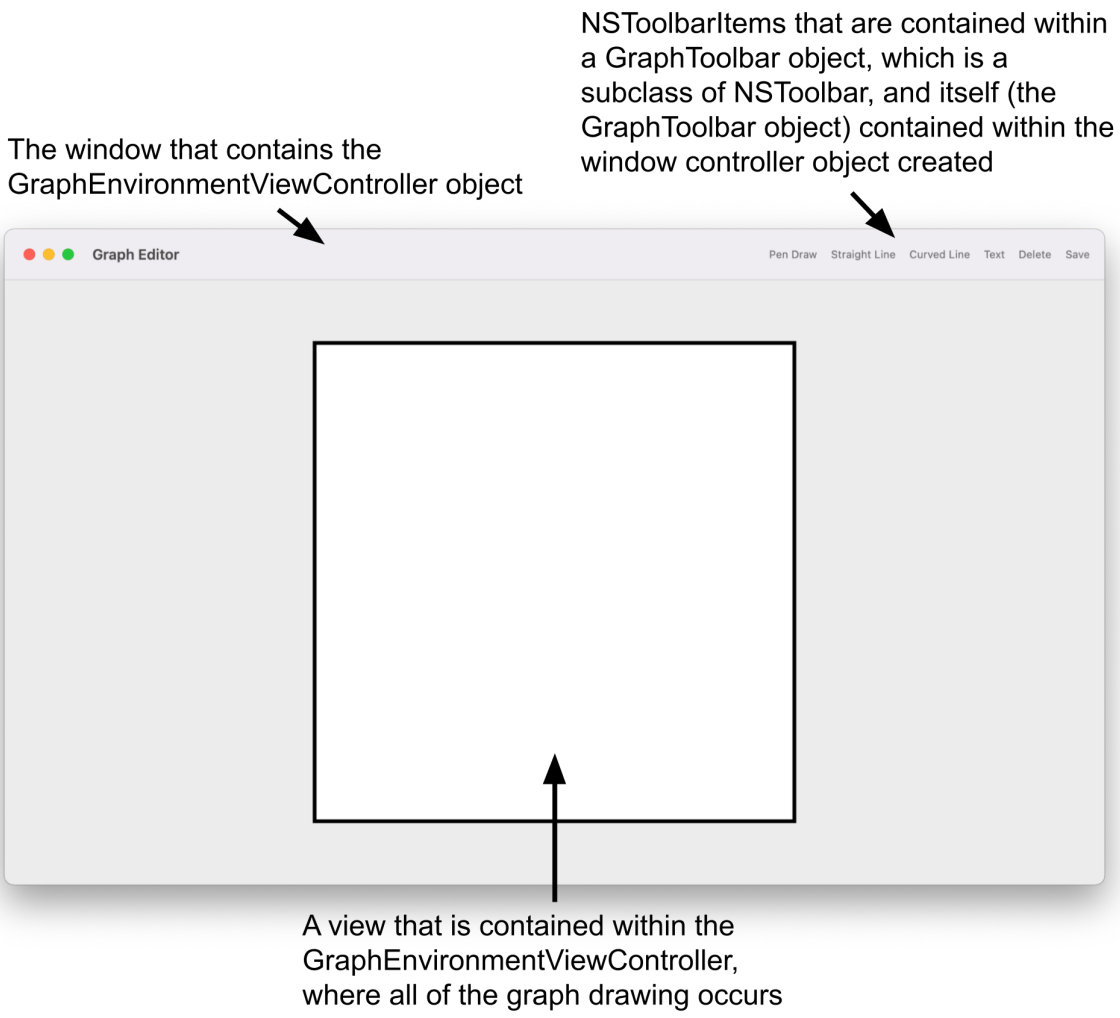
Graphical User Interface

To achieve success criteria 1 (refer to Crit_A_Planning), this application utilises Apple's AppKit framework (see appendix). Excluding the Graph Editor View, all GUI components are implemented with AppKit components (Apple Inc., n.d.). As the client is familiar with AppKit-based applications, the design methodology present across AppKit-based applications was kept in mind.



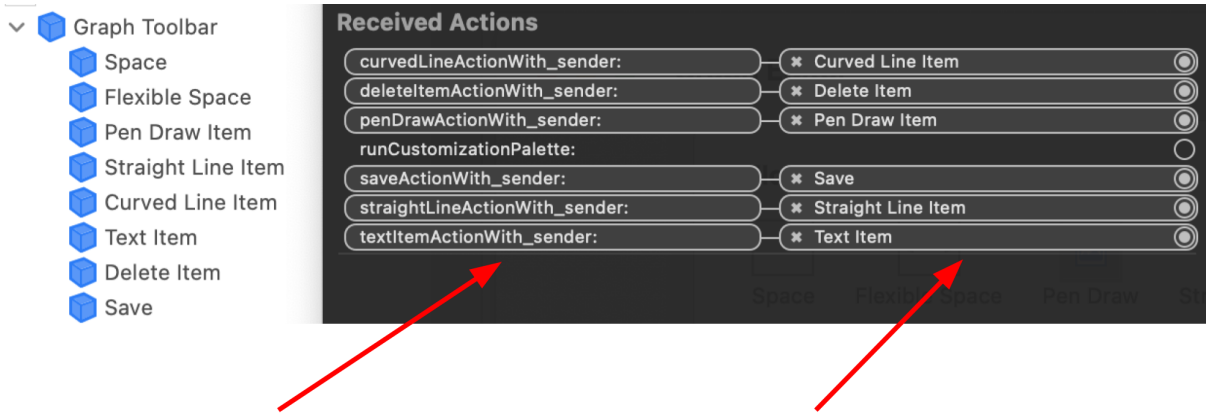


Which the above results in



Interface Builder Connections

Designing was done in Xcode's Interface Builder (IB) (see appendix) to create GUI elements (Apple Inc., n.d.-p). However in order to tell how the GUI must behave, IB Annotations (see appendix) are utilised (Apple Inc., n.d.-k).

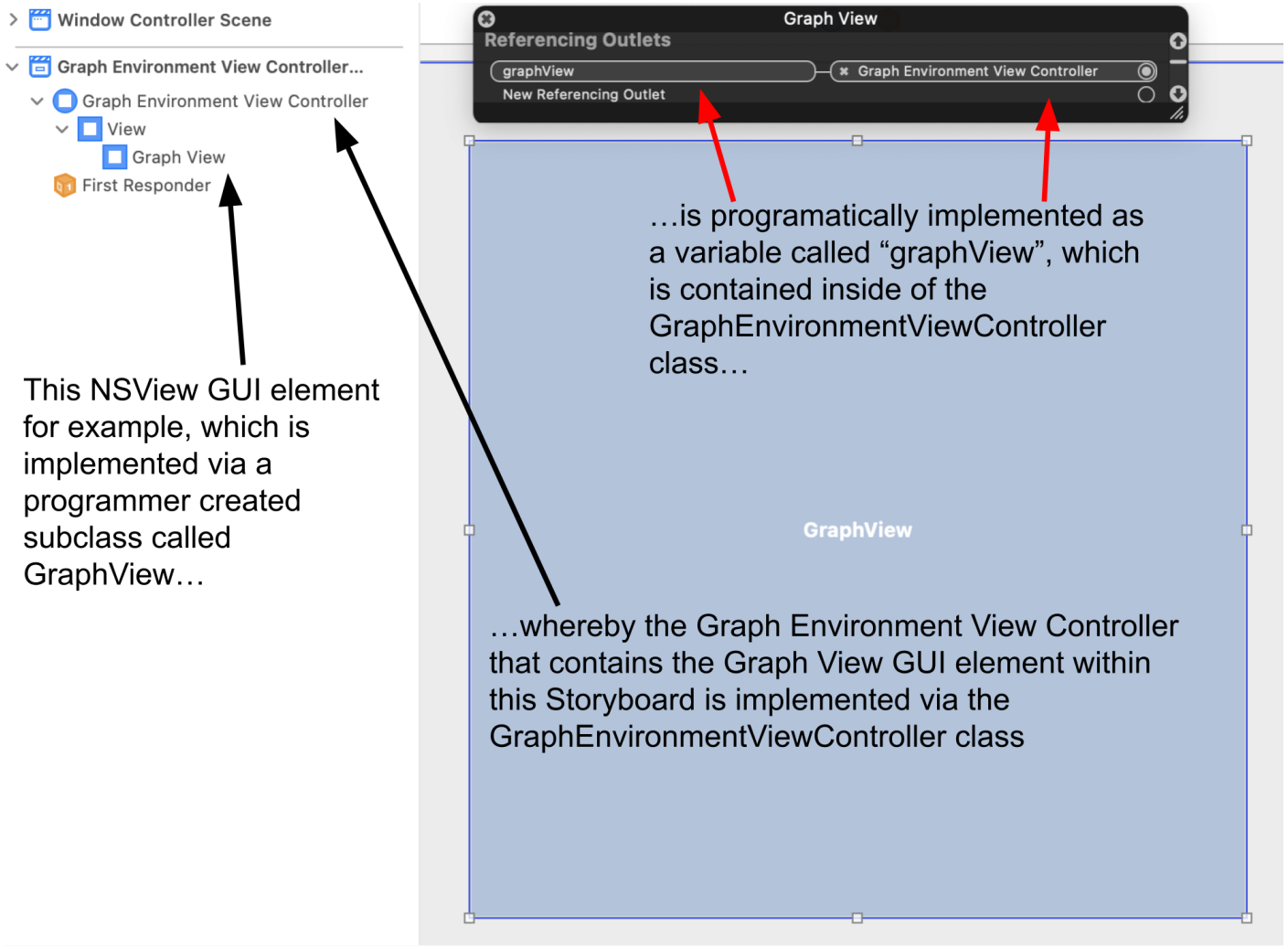


These functions are called… …when these toolbar buttons are pressed

```
12      // A function that is linked to the pen draw toolbar item
13      // "_sender" means that the sender (which can be any data type as the "Any" data type is a wrapper that all
            types, classes, protocols, objects, structs, enums, and other variables and structures in Swift conform to
14      // The use of "_" means that this label does not need to be written when parsing in anything the programmer
            wants into the function
15      // This parameter is standard for functions that are called, whether by an interactive UI element, or by a
            selector
        @IBAction func penDrawAction (_sender: Any) {
            if (GraphView.actionInt == 0) {
                GraphView.actionInt = -1
                GraphEnvironmentViewController.actionInt = -1
            } else {
                GraphView.actionInt = 0
                GraphEnvironmentViewController.actionInt = 0
            }
        }
```

This annotation states that this function can be connected and called by a GUI element made in Interface Builder

This circular indicator states that a GUI element is in fact connected and called by a GUI element made in Interface Builder, and clicking on it will show what element calls it



This circular indicator indicates that a GUI element is in fact connected and programatically represented by this variable, and clicking on it will show what element it programatically represents

```
// The view controller of the graph drawing environment. It is subclassed from NSViewController
class GraphEnvironmentViewController: NSViewController {

    // The toolbar variable, which holds the toolbar within the graph drawing environment
    @IBOutlet var windowToolbar : GraphToolbar!

    // A static variable that holds an 8-bit integer representative of the action to be taken
    // -1 in this program means that not action is to take place
    static var actionInt : Int8 = -1

    // A variable that holds the actual graph view, where drawing takes place
    @IBOutlet weak var graphView : GraphView!
```

This annotation states that this variable is the programmatic representation of a GUI element and changing its attributes will lead to a change in the GUI element that it is connected to

Drawing Lines

Line drawing for the graphs is done with the Core Graphics library (see appendix) (Apple Inc., n.d.-a). As a result, this library allows for drawing that occurs as the user interacts with the graph view itself.

To allow for graph object manipulation post-creation, the attributes of the graph lines are kept within graph objects, whereby specific line types (linear and nonlinear) are subclassed from a graph object superclass named GraphObject. For example with the class representing straight line objects

```
// A class that describes objects representing straight lines
class StraightLineObject : GraphObject {
    var startCoord : CGPoint
    var endCoord : CGPoint
    var equation : String
    // A computed property, effectively a closure that can return values based on other variables during the point
        of call and be written data
    var gradient : CGFloat {
        // Runs when the variable is used
        get {
            if (self.startCoord != CGPoint(x: -1, y: -1) && self.endCoord != CGPoint(x: -1, y: -1) &&
                (self.startCoord.x - self.endCoord.x) != 0) {
                return (self.startCoord.y - self.endCoord.y) / (self.startCoord.x - self.endCoord.x)
            } else { return CGFloat.infinity }
        // Sets the value of the property
        } set(newGradient) {
            self.gradient = newGradient
        }
    }
    // Another computed property
    var yIntercept : CGFloat {
        get {
            self.startCoord.y - (self.gradient * self.startCoord.x)
        } set (newYIntercept) {
            self.yIntercept = newYIntercept
        }
    }

    init() {
        startCoord = CGPoint(x: -1, y: -1)
        endCoord = CGPoint(x: -1, y: -1)
        equation = ""
        super.init(typeInp: GraphObjectsClass.straightLineObjectID)
    }
}
```

Effectively a struct that contains an x-coordinate and a y-coordinate that also has predefined functions

A placeholder that represents division by zero

CGFloat is a floating point type for Core Graphics classes, and are sized differently based on whether the code runs on a 32 bit system or a 64 bit system

A literal that represents an uninitialised coordinate variable, as validation is done so that only coordinates that are within the graph view are drawn

Drawing Text

As text drawing is significantly optimised in AppKit, AppKit's NSTextView was utilised (Apple Inc., n.d.-c), where text views that would hold text and handle user interactions were added into the Graph Editor View as a subview (Apple Inc., n.d.-d), effectively a view placed on top of the parent view. The NSTextViews are wrapped in a GraphObject for easy storage and referencing.

```
// A class that describes objects representing text objects
class TextObject : GraphObject {
    // NSTextView is a class that describes a text box object
    // It was used as it has predefined methods for drawing
    var textView : NSTextView
    var drawLocation : CGPoint

    init() {
        textView = NSTextView.init()
        drawLocation = CGPoint(x: -1, y: -1)

        super.init(typeInp: GraphObjectsClass.textObjectID)
    }
}
```

```
// A function that runs when the text within a text box changes, so that the size of the text box changes
    accordingly
func textDidChange(_ notification: Notification) {
    // In essence this statement first filters out all TextObject graph objects, and then casts the returned
        array of text objects, as a TextObject array, so that the program only iterates through TextObject
        objects, and is already forced downcasted as a TextObject, so that forced downcasting is not repeated.
    // The use of $0 means that we are using the 0th variable that is parsed into the closure, which in this
        case are the graph objects
    // This function checks to see if all of the text boxes have had changes in their contents, and resizes all
        changed text boxes
    for textObject in (self.graphView.GraphObjects.filter({ $0.objectType == GraphObjectsClass.textObjectID })
        as! [TextObject]) {
        // Code that removes the text box if the user removes all the text within it, thus auto-deleting the
            text box.
        // As this function only runs when the text changes, the text box is able to exist as an empty text box
            when first created, as the text contents have not yet changed
        if (textObject.textView.string.isEmpty) {
            // Removes the text box from the graph view
            textObject.textView.removeFromSuperview()
            // Rewrites the graph objects array so that the deleted text box is taken (or rather filtered) out
            self.graphView.GraphObjects = self.graphView.GraphObjects.filter({ $0 != textObject as GraphObject
                })
            // As the text box is now removed, we can now
            continue
        }
        // NSSize is a struct that contains a width and height variable
        // It holds the longest length of a given line
        var sizeToDraw : NSSize = NSSize.init(width: -1, height: -1)
        // The last character index
        var lastCharIdx : Int = 0
        // The number of lines within the text box
        var height : Int = 0
```

```
// Enumerated means that the program is iterating through both the index of the characters within the
    text box, and the characters within the text box themselves
for (idx, character) in (textObject.textView.string).enumerated() {
    // Runs if the character is a newline character
    // The index check is a sanity check so that if the user accidentally adds a newline character at
        the beginning, the NSRange does not state that the lower bound variable (in this case
        lastCharIdx) is greater than the upper bound variable (in this case idx-1)
    if (character.isNewline && character != " " && idx > 0) {
        // An if statement that compares a given line to the largest line width found
        // NSRange is an object that represents range, from a lower bound to an upper bound
        // It is the way that range literals are stored
        // The attributed string is used at it will take into consideration the width and the height of
            the text line with respect to the glyph (a character with specific characteristics, such as
            font, kerning, size) that will be drawn
        if (NSAttributedString.size(textObject.textView.attributedString().attributedSubstring(from:
            NSRange.init(lastCharIdx...idx-1)))().width
            > sizeToDraw.width) {
            // Updates the new longest width of a line
            sizeToDraw = NSAttributedString.size(
                textObject
                    .textView
                    .attributedString()
                    .attributedSubstring(
                        from: NSRange.init(lastCharIdx...idx-1)
                    )
            )()

            // As the current character is a newline character, the next character of the new line
                should be thhe next character index
            lastCharIdx = idx + 1
        }
        // Height is incremented by one, as a newline represents a new line being created, thus
            increasing the height of the text box
        height += 1
    }
}
```

```
// A sanity check is done at the beginning, so that when the user adds a newline character, the lower
    bound of the NSRange variable does not exceede the length of the string
if (lastCharIdx <= textObject.textView.attributedString().length-1 &&
    NSAttributedString.size(textObject.textView.attributedString().attributedSubstring(
    from: NSRange.init(lastCharIdx...textObject.textView.attributedString().length-1)))().width >
        sizeToDraw.width) {
    sizeToDraw = NSAttributedString.size(
        textObject.textView
            .attributedString()
            .attributedSubstring(
                from: NSRange.init(lastCharIdx...textObject.textView.attributedString().length-1)
            )
    )()
}
// Height is incremented one final time, as the 0th line is not accounted for
height += 1

textObject.textView.layoutManager?.ensureLayout(for: textObject.textView.textContainer!)
// Changes the frame of the text box to its new size
// Frame, is unlike bounds, as its origin is with respect to the superview, which in this case is the
    graph view
textObject.textView.frame = .init(
    origin: textObject.drawLocation,
    size: NSSize.init(
        width: ceil(sizeToDraw.width + 10),
        height: ceil(sizeToDraw.height * CGFloat(height))
    )
)

// Tells the text object that its associated text view must redraw
textObject.textView.viewWillDraw()
```

Programming Paradigm Usage

Object Oriented Programming and Protocol Oriented Programming

1. Inheritance

   As all graph objects hold similar characteristics to each other, all graph object classes are subclassed from a common class named “GraphObject”. This also allows all graph objects to be placed into a singular array, with a data type of “GraphObject”, even though individually they have different classes, which simplifies storage of graph objects over the use of multiple arrays.

The identifier of a class in Swift is written in between the “class” keyword and the colon

As classes can only inherit one class, it is standard to write the class that the object is inheriting first. Class inheritance and protocol conformance is shown through writing the class and protocol identifier(s) after the colon

```
// A class that describes objects representing pen drawing
class PenDrawObject : GraphObject {
    // An array of points
    // CGPoint is effectively a struct holding an x-coordinate and a y-coordinate
    var penDrawPoints : [CGPoint]

    // The initialiser of the PenDrawObject
    init() {
        // Writes an empty CGPoint array to the variable
        penDrawPoints =  [CGPoint]()
        // Invokes the initaliser of the superclass, thus initalising the variables of the superclass
        // As this is a parameterised constructor, the type of the object is parsed into the superclass initaliser
        // So that the type of the object is known from initialisation
        super.init(typeInp: GraphObjectsClass.penDrawObjectID)
    }
}
```

The syntax for array creation in Swift is through writing the the data type or object class that the array will contain within square brackets

The syntax for array creation in Swift is through writing the the data type or object class that the array will contain within square brackets, followed by parenthesis, to show that the constructor of the array type is being called

2. Forced Downcasting

As all graph objects are stored under an array with the type of the superclass that they inherit from, the Swift compiler assumes that these objects are purely “GraphObjects”, without the specific variables of each graph object type. Forced downcasting (see appendix) allows these variables to be accessed (Apple Inc., n.d.-n).

Forces the graph object to become a PenDrawObject in order to access the penDrawPoints array

```
case (0):
    // Creates a mutable CGPath object that is effectively an array of points with specialised functions
    let tempVar : CGMutablePath = CGMutablePath.init()
    // Tells the path object to add lines between the points of the pen draw curve
    tempVar.addLines(between: (GraphObject as! PenDrawObject).penDrawPoints)
    // An if statement that checks to see if the point of the user click intersects with the pen draw curve
    if (tempVar.contains(pointInTargetView)) {
        // Filters out the graph object to be deleted, and then returns the rest of the graph objects in a new array that has been
            automatically resized with all elements shifted
        graphView.GraphObjects = graphView.GraphObjects.filter{ $0 != GraphObject }
        // A break statement that exists the loop so that only one object is deleted
        break objectDeleted
    }
    // A break statement that exists the case's code body
    break
```

3. Polymorphism

   Operator overloading allows for significantly more dense code through describing what an operator is doing for a specific data type or object. This can be seen through overloading the equality (==) and inequality (!=) operators to do quick equality checks between objects.

```
// The extension keyword implies that this code body extends the functionality of the CurvedLineObject class
// In this scenario, it is adding functions that allow the CurvedLineObject class to conform to Equatable through overloading the equality and
    inequality operators, which it must as CurvedLineObject inherits this from its superclass, GraphObject
extension CurvedLineObject {
    // Operator overloading the equality operator so that CurvedLineObjects can be compared using the equality operator
    // "left" represents the object on the left side of the operator, and "right" represents the object on the right side of the operator
    // "-> Bool" means that this function will return a boolean variable
    static func ==(left : CurvedLineObject, right : CurvedLineObject) -> Bool {
        return (
            left.height == right.height &&
            left.width == right.width &&
            left.objectType == right.objectType &&
            left.parameters == right.parameters &&
            left.useAdvanced == right.useAdvanced &&
            left.bezierControlPoints == right.bezierControlPoints &&
            left.isCubic == right.isCubic
        )
    }

    // Operator overloading the inequality operator so thhat CurvedLineObjects can be compared using the equality operator
    static func !=(left : CurvedLineObject, right : CurvedLineObject) -> Bool {
        return (
            left.height != right.height ||
            left.width != right.width ||
            left.objectType != right.objectType ||
            left.parameters != right.parameters ||
            left.useAdvanced != right.useAdvanced ||
            left.bezierControlPoints != right.bezierControlPoints ||
            left.isCubic != right.isCubic
        )
    }
}
```

Overrides the equality operator so that when it is used for the CurvedLineObject, the program knows how to handle an equality check

Overrides the inequality operator so that when it is used for the CurvedLineObject, the program knows how to handle an inequality check

4. Delegation

   Some objects also inherit protocols (see appendix) (Apple Inc., n.d.-m) in order to become delegates (see appendix) for certain objects (Apple Inc., n.d.-e), such as NSTextView, in order to detect changes in the text contained within an NSTextView object, and resize the view accordingly.

```
// Assigns the delgate of this text box to be "GraphEnvironmentViewController", which will now handle changes and the user interactions that
    occur on the text box
textObject.textView.delegate = self
```

```
// The extension keyword implies that this code body extends the functionality of the GraphEnvironmentViewController class
// In this scenario, it is inheriting the NSTextViewDelegate protocol, and thus its specific functions and characteristics
extension GraphEnvironmentViewController : NSTextViewDelegate {
    // A function that runs when the text within a text box changes, so that the size of the text box changes accordingly
    func textDidChange(_ notification: Notification) {
```

States that the GraphEnvironmentViewController class conforms to the NSTextView protocol, and thus is able to become the delegate of an NSTextView object

A function inherited from conforming to the NSTextViewProtocol, that allows the GraphEnvironmentViewController class to detect when text within an NSTextView has changed, and resize the view accordingly

5. Protocol Conformance

Protocol conformance (see appendix) is used to give certain classes certain behaviours, which can be seen through using protocol conformance to ensure that the GraphObject objects can utilise the equality and inequality operator through conforming to the Equatable protocol

Protocol conformance is often done in extensions to demonstrate that the code within the extension block allows the class to conform to the protocol

States that the GraphObject class conforms to the Equatable protocol, and thus inherits the behaviours that are expected of classes that conform to the Equatable protocol.

```
// The extension keyword implies that this code body extends the functionality of the GraphObject class
// This extension states that GraphObjects conforms to the Equatable protocol, allowing it to use the equality and inequality operator. This protocol inheritance is passed down to
  subclasses of GraphObject, forcing them to conform to the Equatable protocol as well
extension GraphObject : Equatable {
    // Operator overloading the equality operator so that GraphObjects can be compared using the equality operator
    // "left" represents the object on the left side of the operator, and "right" represents the object on the right side of the operator
    // "-> Bool" means that this function will return a boolean variable
    static func ==(left : GraphObject, right : GraphObject) -> Bool {
        // These if statements are meant to call equality functions that do further equality checks on specific attributes of two given graph objects, which is important when the
            graph objects being compared have not been downcasted to their specific graph types
        if (left.objectType == right.objectType && left.objectType == GraphObjectsClass.penDrawObjectID) {
            // As the type of the two graph objects used with the equality operator are the same, they are both force downcasted to the same graph object class so that further
                equality check can occur with the specific variables found within the different graph object, as operator overloading has occured for those graph object types as well,
                and by downcasting both of the objects to the same graph object class, the operator operations associated with the equality operator for that specific graph object can
                occur
            return ((left as! PenDrawObject) == (right as! PenDrawObject))

        } else if (left.objectType == right.objectType && left.objectType == GraphObjectsClass.straightLineObjectID) {
            return ((left as! StraightLineObject) == (right as! StraightLineObject))

        } else if (left.objectType == right.objectType && left.objectType == GraphObjectsClass.curvedLineObjectID) {
            return ((left as! CurvedLineObject) == (right as! CurvedLineObject))

        } else if (left.objectType == right.objectType && left.objectType == GraphObjectsClass.textObjectID) {
            return ((left as! TextObject) == (right as! TextObject))

        } else {
            return (
                left.height == right.height &&
                left.width == right.width &&
                left.objectType == right.objectType &&
                left.parameters == right.parameters &&
                left.useAdvanced == right.useAdvanced
            )
        }
    }
```

Functional Programming

Functional programming (FP) (see appendix) is utilised to extract specific elements from an array, through the filter function, which allows for a subarray of elements with specific characteristics to be created without iterating through the array manually (Apple Inc., n.d.-f).

```
// If "actionInt" is equal to 7, this means that a delete operation has been attempted, which means that it does not matter if the click
    intersects with any text boxes
// If the click is contained within a text box, and the user is attempting to draw something, the program stops that from happening, and
    allows the user to instead edit the text box
// This is done by first creating an array that filters out only text objects, and then another function that checks to ensure that all
    text objects satisfy the condition that their text box does not contain the point that the user has clicked on
if (GraphEnvironmentViewController.actionInt == 7 || graphView.GraphObjects
        .filter { $0.objectType == GraphObjectsClass.textObjectID }
            .allSatisfy{ !($0 as! TextObject).textView.frame.contains(pointInTargetView) }
) {
```

Another feature of functional programming whereby a function checks all elements present within the new array created from the filter function to see if they all satisfy a specific condition, and if so, the function will return true, else it will return false

$0 is the 0th variable that is returned by the allSatisfy function that can be used within the closure to handle the actual equality logic. This is because the allSatisfy function is defined to (and the filter function) returns a closure that contains a variable, and that closure is then expected to return a boolean value, which will then determine if all elements satisfy a certain condition

A feature of functional programming whereby a function filters out elements that meets a specific condition, and then places them into a new array

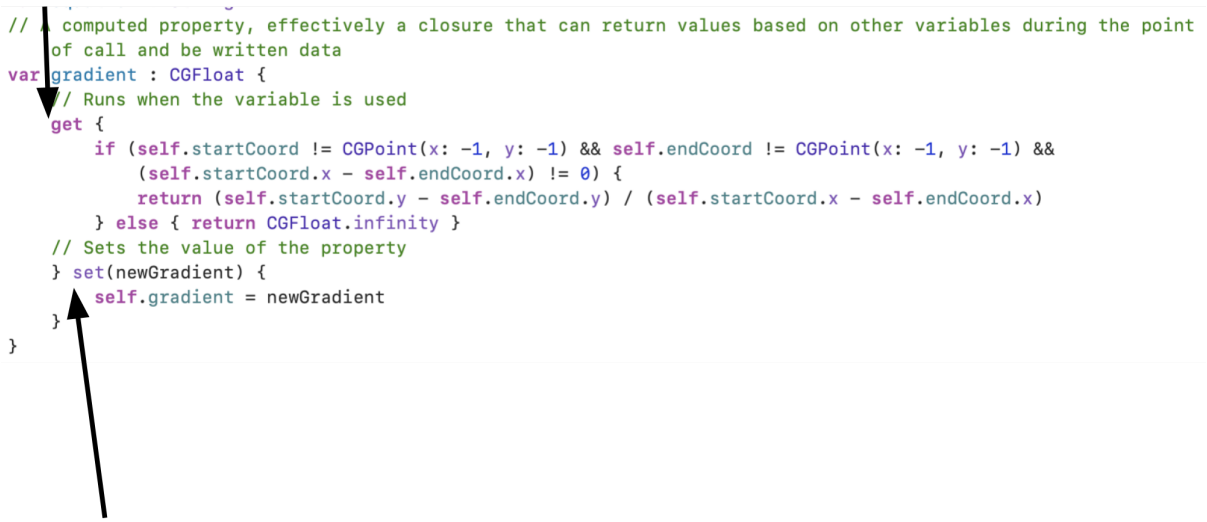
Computed Properties

As some properties are expected to change based on the conditions of the program, computed properties (see appendix) are utilised to allow for variables to have their values at the point they are needed, thus avoiding the use of function, which can make code more readable (Apple Inc., n.d.-l). For example, when the gradient variable is read, as seen below,

```
// As the graph object is a straight line, the y-coordinate can thus be calculated
    via the equation "y = mx + c", where m is the gradient, and c is the y-intercept
// Thus, the points that will be utilised for detecting if the point of the user
    click intersects with the straight line object will be calculated based on the
    equation above
pointsArray.append(CGPoint(x: xValue, y: ((GraphObject as!
    StraightLineObject).gradient * xValue) + (GraphObject as!
    StraightLineObject).yIntercept))
```

the getter code body runs.

A keyword that represents a code block that runs when the program attempts to read the value of “gradient”. Return statements represent what the variable will return when attempted to being read.



A keyword that represents a code block that runs when the program attempts to write to the variable. It exists mainly in case a programmer attempts to write to the variable without knowing that it is a computed property.

Data Structures

Swift by design includes Arrays (equivalent to Java's ArrayList) (Apple Inc., n.d.-g) and Dictionaries (equivalent to Java's HashMaps) (Apple Inc., n.d.-h) which were used to store data. Tuples although built-in, are rarely used throughout this program as they are immutable, and their ability to store multiple data types can be replaced with objects, and this ability means that conversion from tuples to other data structures even when the data types are present within the tuple is the same requires pointers (Apple Inc., n.d.-o).

```
// An array of graph objects for storage
var GraphObjects = [GraphObject]()
```

```
// An identifier that represents a type, which will be used to create a dictionary of functions to draw the
    graph objects
typealias VoidFunction = () -> Void

// A dictionary of functions to draw the graph objects
var funcDict = [String : VoidFunction]()


// Enumerated means that the program is iterating through both the index of the characters within the
    text box, and the characters within the text box themselves
for (idx, character) in (textObject.textView.string).enumerated() {
```

A tuple that is updated after every iteration

Error Handling

A benefit of Swift is its high safety due to strong and static typing (Finn, n.d.), along with constant internal checking of types, and the presence of data. Most error handling is internal for this program as user input is mostly cursor-based.

Validation checks ensure that all data inputted is usable and conforms to the expectations of the program before continuing. Other error handling methods will ensure that actions are not taken that will lead to the program crashing, or undefined behaviour, such as wrong forced downcasting.

With that said, specific checkpoints are placed throughout the program to force the program to return an exception when internal inconsistencies occur, such as mismatching data types, or reading nonexistent variables or values.

These techniques assist in achieving success criteria 5

```
// As graph objects are drawn within "graphView"'s rectangle, with respect to "graphView"'s coordinate
    system, the mouse location that is taken with respect to the coordinate system of the window's primary
    view, is converted to coordinates with respect to the coordinate system of the target view, which in
    this case is "grapView"'s rectangle
// As "graphView.window?.mouseLocationOutsideOfEventStream" may be nil, the nil coalescing operator is used
    so that if there exits no mouse location recorded, despite a mouse click, then the literal "CGPoint(x:
    0, y: 0)" is used so that the program doesn't crash when attempting to operate on a nil value
let pointInTargetView = self.graphView.convert(graphView.window?.mouseLocationOutsideOfEventStream ??
    CGPoint(x: 0, y: 0), from: self.view)

// If the mouse click has occured within the graph view, then the statement is true
if (graphView.bounds.contains(pointInTargetView)) {
```

A form of validation as this ensures that the point clicked is within the graph view, else the object would be drawn out of view, which if occured enough times could lead to many objects occupying memory without them ever being visually seen

Optional Handling

Many Swift data types may be optional (see appendix), meaning that they lack a value and hence are nil, Swift's equivalent of Java's null, (Apple Inc., n.d.-i). As nils cannot be used, optional handling must be done to ensure that when the program "unwraps" (revealing the true value of the optional) a nil value, the branch of execution is changed so that the program does not attempt to operate on nil values.

Throughout earlier examples, the symbols "?", "!", and "??" have been used to handle options. "?" means to unwrap the variable if nil is not present, whereas "!" is the same as "?", but a runtime error is thrown if the variable is nil.

When using "?" to unwrap a variable within a statement that is expecting a non-nil value, the nil coalescing operator, "??", is used to present the statement with a different value should the variable have a nil value.

```
// Data refers to a data type of raw bytes in memory
// As such, it is important to keep track of what type of data is being stored when the Data data type is used
// In this scenario, PDF data is being stored to the variable
// This variable is an optional as the variable may not always have something stored inside of it, as the graph
    view may not have been initialised
static var graphViewToSave : Data?
```

```
// As graph objects are drawn within "graphView"'s rectangle, with respect to "graphView"'s
    coordinate system, the mouse location that is taken with respect to the coordinate system
    of the window's primary view, is converted to coordinates with respect to the coordinate
    system of the target view, which in this case is "grapView"'s rectangle
// As "graphView.window?.mouseLocationOutsideOfEventStream" may be nil, the nil coalescing
    operator is used so that if there exits no mouse location recorded, despite a mouse click,
    then the literal "CGPoint(x: 0, y: 0)" is used so that the program doesn't crash when
    attempting to operate on a nil value
let pointInTargetView =
    self.graphView.convert(graphView.window?.mouseLocationOutsideOfEventStream ?? CGPoint(x: 0,
    y: 0), from: self.view)
```

As the "mouseLocationOutsideOfEventStream" variable may be nil, the use of the question mark emphasises that the variables after it may be nil

The nil coalescing operator is used as the function requires a non-nil value parsed into it, else the program will fail. By using this operator, if the variable preceding it returns nil, then the non-nil value after it will be used, which in this case is a CGPoint literal

```
// A closure that returns nothing. The use of the exclamation mark means that if the closure is nil, or in
    other words not initialised, it will invoke a runtime error
var penDraw : VoidFunction!

var straightLineDraw : VoidFunction!

var curvedLineDraw : VoidFunction!

var textDraw : VoidFunction!
```

The use of the exclamation mark means that if the variable is nil when used, the program will create an exception as a runtime error. This is because if these closures are not written to, this means that the graph object type that the closure is used for is empty, then the program cannot continue regardless. Thus it is necessity that they are written to, hence why they cannot be nil when read

File Writing

Swift's file writing utilises the "Data" data type, which contains byte data (Apple Inc., n.d.-j). In order to write an image representing the view of graphView, the view is first converted to PDF data, and then converted to a TIFF representation, wrapped as an NSImage object, before converted to bytes for writing. This is to achieve success criteria 4.

```
// Data refers to a data type of raw bytes in memory
// As such, it is important to keep track of what type of data is being stored when the Data data type is used
// In this scenario, PDF data is being stored to the variable
// This variable is an optional as the variable may not always have something stored inside of it, as the graph
    view may not have been initialised
static var graphViewToSave : Data?
```

```
// A function that is linked to the save toolbar item
@IBAction func saveAction (_sender: Any) {

    // The panel is first saved to a variable, to allow for changes in attributes, and later invocation
    let savePanel = NSOpenPanel()
    savePanel.canCreateDirectories = true
    savePanel.allowsMultipleSelection = false
    savePanel.canChooseDirectories = true
    savePanel.canChooseFiles = false

    // The panel's function is invoked as a closure, whereby "result" is the variable that holds the results of the user interactions. As functions are meant to have return types,
        the use of -> symbolises that this closure returns nothing after finished
    savePanel.begin { (result) -> Void in
        // An if statement that is true if the user clicks on the OK button within the panel
        if result.rawValue == NSApplication.ModalResponse.OK.rawValue {
            // The directory that the user plans to save in
            let saveLocationURL = savePanel.directoryURL

            // The directory and the file name combined. The line below will yield the directory + "Untitled.tiff", as the user has not and cannot enter a name for the file into
                the name field, as Open Panels do not have that
            // ".tiff" is appended to the end of the file name to ensure that no matter happens, the file will always save as a TIFF file image
            let fileName = saveLocationURL?.appendingPathComponent(savePanel.nameFieldStringValue + ".tiff")

            // The byte data is created based on PDF data from "graphViewToSave", and then converted to a TIFF image file representation, before it is converted back to bytes for
                later writing
            let TIFFImage = Data((NSImage.init(data: GraphToolbar.graphViewToSave!)?.tiffRepresentation)!)

            // A structure that can handle errors thrown from statements that can throw errors
            do {
                // Attempts to write the TIFF image file to the desired directory
                try TIFFImage.write(to: fileName!)
            } catch {
                // For internal debugging. It is not seen by the end user
                print(error)

            }
            return
        // An if statement that is true if the user clicks on the Cancel button within the panel
        } else if result.rawValue == NSApplication.ModalResponse.cancel.rawValue {
            return
        }
    }
}
```

[1058 words]

Bibliography

Apple Inc. (n.d.-a). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/appkit
Apple Inc. (n.d.-b). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/coregraphics
Apple Inc. (n.d.-c). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/appkit/nstextview
Apple Inc. (n.d.-d). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/appkit/nsview/1483539-subviews
Apple Inc. (n.d.-e). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/swift/using-delegates-to-customize-object-behavior
Apple Inc. (n.d.-f). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/swift/sequence/filter(_:)
Apple Inc. (n.d.-g). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/swift/array
Apple Inc. (n.d.-h). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/swift/dictionary
Apple Inc. (n.d.-i). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/swift/optional
Apple Inc. (n.d.-j). Apple Developer Documentation. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/documentation/foundation/data
Apple Inc. (n.d.-k). Attributes — The Swift Programming Language (Swift 5.7). Retrieved June 15, 2022, from https://docs.swift.org/swift-book/ReferenceManual/Attributes.html
Apple Inc. (n.d.-l). Properties — The Swift Programming Language (Swift 5.7). Retrieved June 15, 2022, from https://docs.swift.org/swift-book/LanguageGuide/Properties.html
Apple Inc. (n.d.-m). Protocols — The Swift Programming Language (Swift 5.7). Retrieved June 15, 2022, from https://docs.swift.org/swift-book/LanguageGuide/Protocols.html
Apple Inc. (n.d.-n). Type Casting — The Swift Programming Language (Swift 5.7). Retrieved June 15, 2022, from https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html
Apple Inc. (n.d.-o). Types — The Swift Programming Language (Swift 5.7). Retrieved June 15, 2022, from https://docs.swift.org/swift-book/ReferenceManual/Types.html
Apple Inc. (n.d.-p). Xcode 14 Overview - Apple Developer. Retrieved June 15, 2022, from Apple Developer website: https://developer.apple.com/xcode/
Finn, A. (n.d.). Chapter 5 Static Typing and Type Inference | Learn Swift. Retrieved June 15, 2022, from - /aidanf/ website: https://www.aidanf.net/learn-swift/types_and_type_inference