CS315 – Programming Languages Project

“Paint++”

Part I

Course Instructor:	Buğra Gedik
Group Number:	2-9
Group Members:	Erin Avllazagaj
	Gülsüm Güdükbay
	Melis Kızıldemir

Table Of Contents

1. Introduction

1.1 It's An Interpreted Language

Paint++ is a scripting language. It doesn't have a compiler that transforms it to machine code. It is executed line by line from the interpreter. It is faster than other scripting languages due to its simple grammar. The only part where it can slow down is when it needs to render the canvas in the screen. The software is assured to run on every decent machine. It will need the user to have g++ compiler or mingw32 compiler for windows. The makefile makes things easier for all the platforms. It will try to compile in Unix environment by default (with g++) and if it gets error it will try for Windows machines. In case of failure the language just won't be able to run.

1.2 Runs sequentially

Since it is an interpreted language the code will run sequentially from top to bottom. Repetition and decision structures are also allowed to make sure that any kind of algorithm can be implemented. The only drawback is that the user cannot define classes or include any new libraries and any new function definition. The language already provides all the necessary functions to draw any kind of shape. It also supports array but only when needed to declare composite shapes.

1.3 Coordinate system orientation

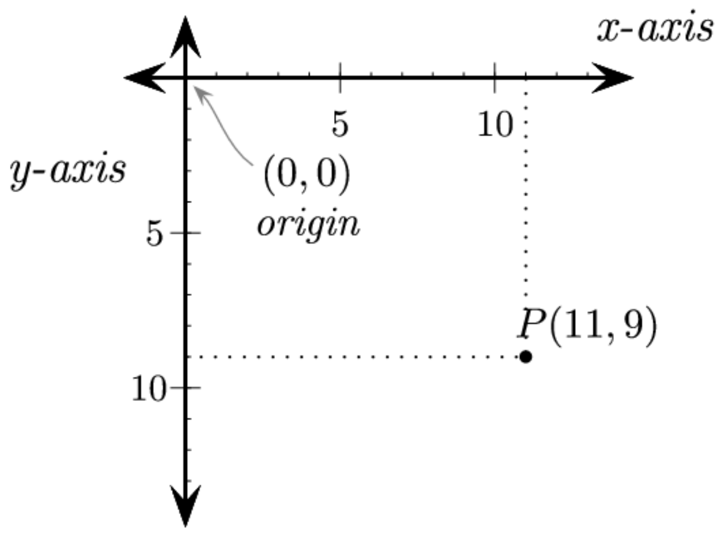


Figure 1 - Paint++ coordinate system

The coordinate system of Paint++ is same as the one used in Java or GTK Python libraries. It has the origin at the top-left corner of the window and increases to the right in x and to the bottom in y. The figure below explains it very well. The units that it's using are pixels, since they are universal to any computer screen. The painting is implemented by Python's PyGUI library that supports Cocoa, Gtk and Win32 UIs.

2. Supported Properties

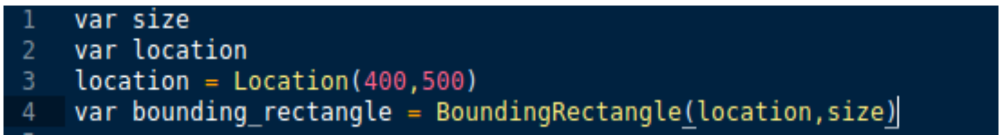
2.1 Entry Point

The developer should be able to declare a variable and make that a Location with specified x and y coordinates. The x and y must be integer types or var. Any float or object will cause error. The binding of the variable declared to Location is done at runtime. But the variable should be a Location before it reaches the point when it is actually used by any function. This rule applies to all the variables, since our language runs sequentially from top to bottom.

After, or even before, declaring a Location, the developer should also declare a Size having width and height of what the bounding rectangle is supposed to have. The user again can declare these variables in any order. It is just a matter of preference. So the language supports many different mindsets and preferences of different developers and it's very easy to read since all the variables need to be initialized before being used.

After declaring these two variables then the developer should be able to declare a third variable which is going to be used for the bounding rectangle. It will have as parameters the Location and Size variables declared and specified above. This is the point where the variables above will be used and an uninitialized variable at this point will give an error at runtime.

An example of the error of initialization is given below at figure 1.



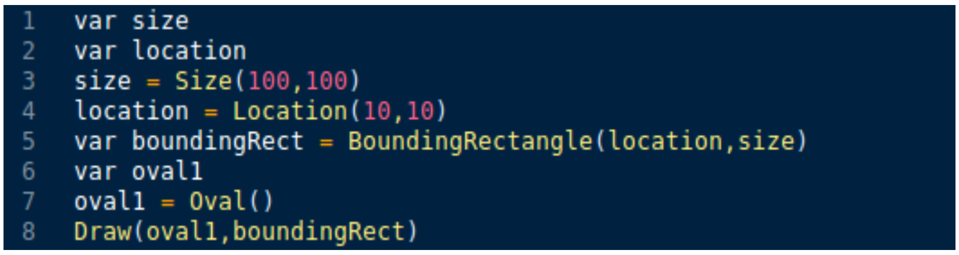
```
1  var size
2  var location
3  location = Location(400,500)
4  var bounding_rectangle = BoundingRectangle(location,size)
```

Figure 2 – Error in runtime "Uninitialized variable"

Any syntax highlighter will accept this code, but when it is supposed to be interpreted it will throw "Uninitialized variable exception" and will point exactly to line 4 in the code so it's easy for the developer to check where that happened. Since a function has limited number of variables then the number of ways to backtrack this is very limited so it's easily debuggable too.

The main feature explained here is the common entry point. To define the bounding rectangle in any way the user can go in two ways, but in the end they have to call the BoundingRectangle function which will be predefined function of the language but not a reserved keyword so user can use it in variable.

Another common entry point is the draw function. This function is overloaded to support different types of drawing, which will be explained in the following sections of this report. In the figure below we show an example of a simple program written in Paint++. It will have a single entry point which is the Draw function.



```
var size
var location
size = Size(100,100)
location = Location(10,10)
var boundingRect = BoundingRectangle(location,size)
var oval1
oval1 = Oval()
Draw(oval1,boundingRect)
```

Figure 3 – The "Draw" entry point

The output of the program below will resemble the figure below. Please note that the output is not actually scaled properly so it might be different to what the developer sees, but the concept demonstrated above will hold for sure.

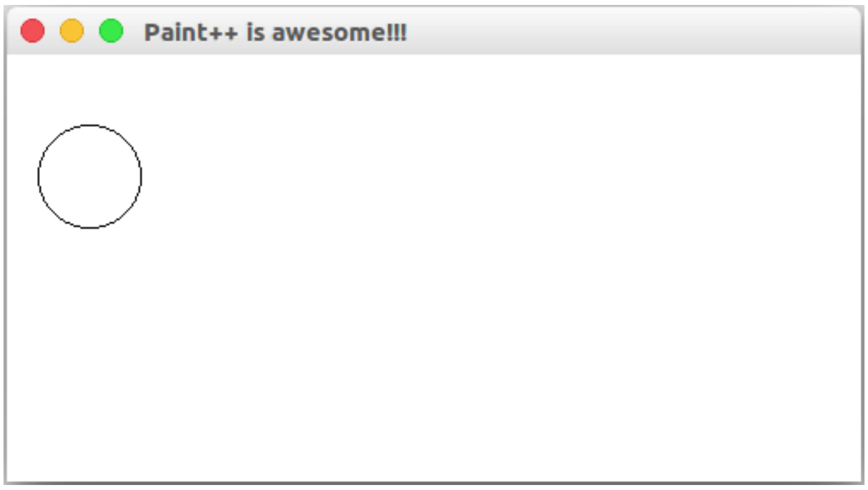
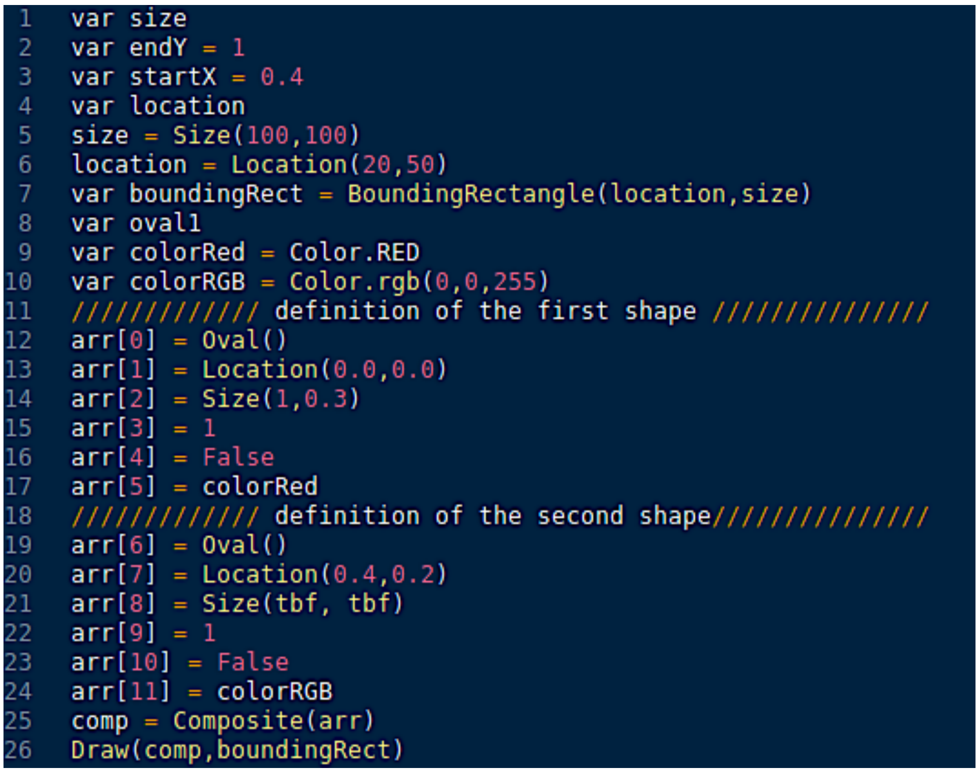


Figure 4 – The Output Of Program in Figure 3

2.2 Location, Size and Color Types

Another feature that the language will support for each shape rather than Location and Size is the Color. It is an optional parameter for all the shapes. By default the color of the shape is black since the canvas Paint++ draws over is white. A good example is changing the color of the two ovals of the code in figure 5. The new code would look like the one in the figure below.



```
var size
var endY = 1
var startX = 0.4
var location
size = Size(100,100)
location = Location(20,50)
var boundingRect = BoundingRectangle(location,size)
var oval1
var colorRed = Color.RED
var colorRGB = Color.rgb(0,0,255)
///////////// definition of the first shape //////////////
arr[0] = Oval()
arr[1] = Location(0.0,0.0)
arr[2] = Size(1,0.3)
arr[3] = 1
arr[4] = False
arr[5] = colorRed
///////////// definition of the second shape//////////////
arr[6] = Oval()
arr[7] = Location(0.4,0.2)
arr[8] = Size(tbf, tbf)
arr[9] = 1
arr[10] = False
arr[11] = colorRGB
comp = Composite(arr)
Draw(comp,boundingRect)
```

Figure 5 – Code explaining the Color optional parameter

The code above is giving exactly the output as the one in figure 5 just the colors change. Paint++ has support of 9 main colors as constants:

1. Color.BLACK
2. Color.BLUE
3. Color.CYAN
4. Color.GREEN
5. Color.ORANGE
6. Color.PURPLE
7. Color.RED
8. Color.YELLOW
9. Color.WHITE

For any other type of color the developer is free to use the rgb color constructor. As in line 10 of the code above Color.rgb() accepts 3 integer type variables and will return a Color object and assign it to the colorRGB variable. If anything else other than exactly three integers or integer type variables are given to the rgb constructor then the program will crush throwing an "undefined type exception". The Oval constructor takes no parameter. The figure below explains the output as what is supposed to be. Note that the figure is not really drawn at a scale and is just a simulation of what the real thing is.

As seen from the line 11 and 18 Paint++ supports the comments but only the inline comment just like Python, however in Paint++ it needs to start with //. The rest is ignored.

This code above allows the user to draw two ovals in the same bounding rectangle and the parameters must be positioned in an array like:

1. The figure is assigned to position 0 in the array.
2. The relative Location is assigned to position 1. It must contain numbers from 0 to 1.
3. The relative Size is assigned at position 2. It must contain numbers from 0 to 1.
4. The stroke width is assigned at position 3. It must be an integer.
5. The fill status is assigned at position 4. It must be a boolean
6. The color is assigned at position 5. It must be a color object.

If there are less relative definitions then the program will crash and throw "unknown type exception". Or if the order is messed up the program will crash the same way. The crash will be seen in the line 25. Possible pitfalls are if the developer uses less or more parameters than what the array needs. In which case the interpreter will crash and notify the developer of "unknown type exception". Or if the developer puts in a string or any object in its 5 parameters. Then again it will crash at runtime throwing "unknown type exception". If the developer decides not to declare anything for the stroke width or anything then they should leave that part of the array empty so for example in this case line 15 and 22 can be deleted and the output will be exactly the same as in figure below.

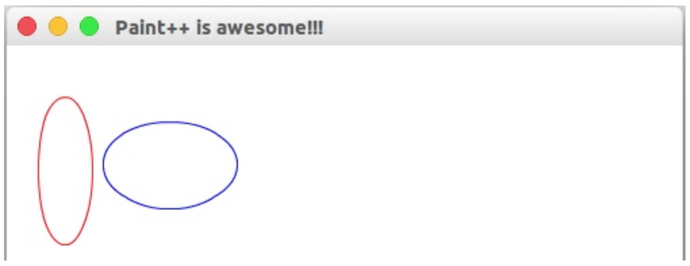
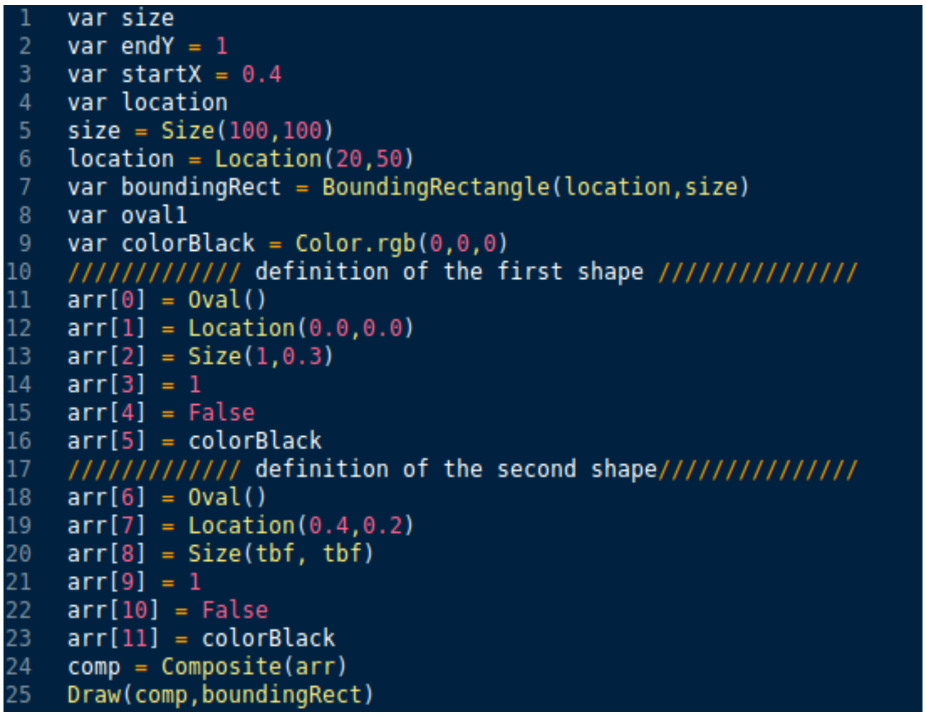


Figure 6 – Output of the code in figure 7

3. Independence From Scale

Another great feature of Paint++ is the introduction of the **relative declarator**. This gives the developer freedom when it comes to drawing. It gives them the ability to draw independent of the scale. The developer for example can draw more than one basic figure in the same bounding rectangle irrespective of the size. This is what a newly defined shape is in Paint++. In that case the developer has to define relative size, which is the ratio of the bounding rectangle that is going to be used for each figure. The bounding rectangle is going to be measured left to right and top to bottom just like the coordinating system that is described above. The code example below best describes this feature of the system.



```
var size
var endY = 1
var startX = 0.4
var location
size = Size(100,100)
location = Location(20,50)
var boundingRect = BoundingRectangle(location,size)
var oval1
var colorBlack = Color.rgb(0,0,0)
///////////// definition of the first shape //////////////
arr[0] = Oval()
arr[1] = Location(0.0,0.0)
arr[2] = Size(1,0.3)
arr[3] = 1
arr[4] = False
arr[5] = colorBlack
///////////// definition of the second shape//////////////
arr[6] = Oval()
arr[7] = Location(0.4,0.2)
arr[8] = Size(tbf, tbf)
arr[9] = 1
arr[10] = False
arr[11] = colorBlack
comp = Composite(arr)
Draw(comp,boundingRect)
```

Figure 7 – Independent Drawing

Another thing to notice is that Paint++ doesn't support direct passing of the parameters like something in the figure below. That is because the interpreter is designed to be a quick and easy step since the rendering of the code will take a lot more time.

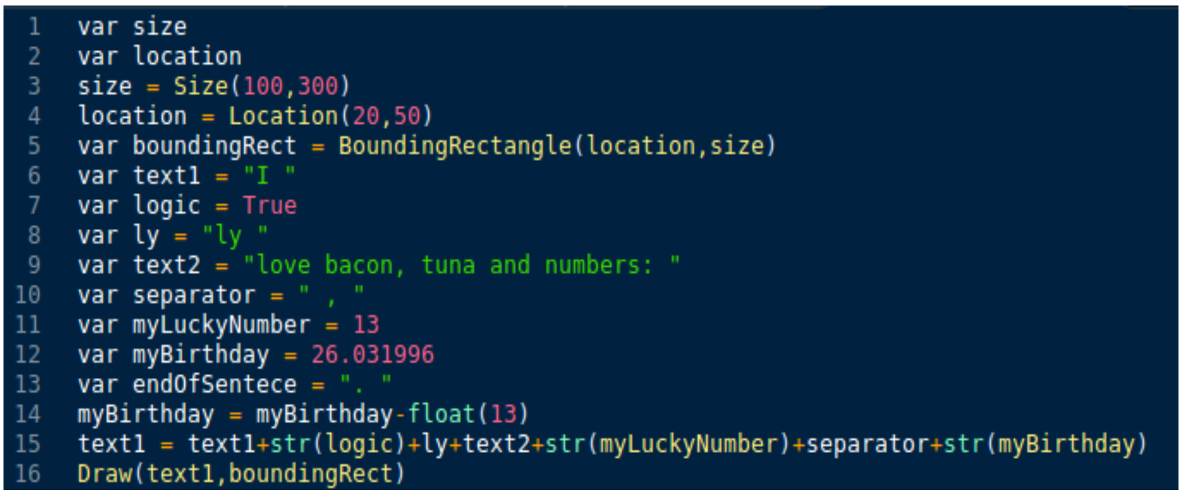
4. Dynamically Typed

Paint++ is a dynamically typed language that means the variable stays abstracts until assigned to any type. It is declared as var whether it is primitive or composite type. The integers and floats support +,-,* and \. Strings support only + and booleans have their logic operators.

1. The integers can turn to float using float() function which takes an integer and returns its float equivalent the opposite is the int() function.
2. The int, float and boolean can turn into string using the str() function.
3. Booleans can become 1(or 0) or 1.0(or 0.0) using int() and float() functions respectively. They can also become strings using str() function and become “True” or “False”.
4. Strings can become float using(float), int using int() or boolean using bool(). The only condition is when converting from string to float or int it must not contain any character other than 0 through 9 and a single dot for floats and no dot for ints.

The functions above support either fixed parameter or any identifier as parameter.

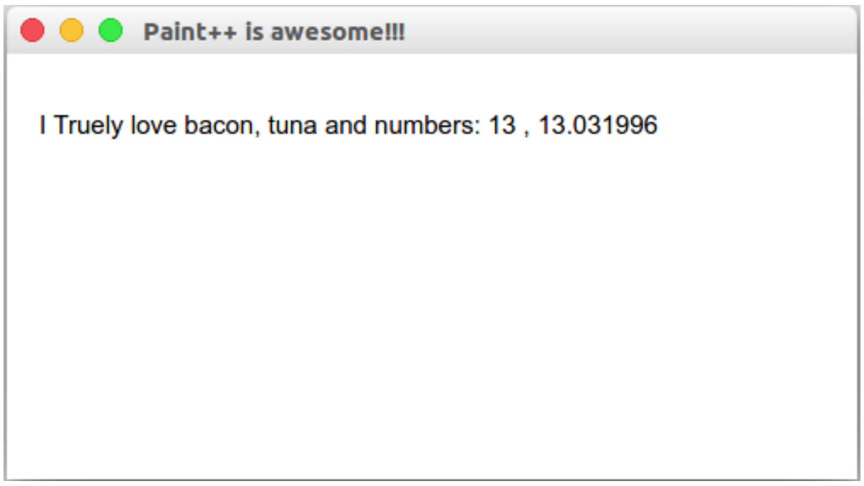
To clearly demonstrate some of the rules above the code in figure 8 is going to help:



```
var size
var location
size = Size(100,300)
location = Location(20,50)
var boundingRect = BoundingRectangle(location,size)
var text1 = "I "
var logic = True
var ly = "ly "
var text2 = "love bacon, tuna and numbers: "
var separator = " , "
var myLuckyNumber = 13
var myBirthday = 26.031996
var endOfSentece = ". "
myBirthday = myBirthday-float(13)
text1 = text1+str(logic)+ly+text2+str(myLuckyNumber)+separator+str(myBirthday)
Draw(text1,boundingRect)
```

Figure 8 – Variable Types and The Operations

As we can see, the Draw function can take a string type parameter but only shapes and string type. It doesn't take any other parameter because it will crash. Strings are then drawn in the bounding rectangle. The output for the code above is in the figure below. As we can see the boolean value got translated into the "True" string and in line 14 the fixed parameter we passed to the float() function made the integer 13 become 13.0 and be accepted by the operation. In the line 15 we just concatenate all the strings and the ones that are converted to strings. Note that the y of boundingRect is increased to 300px to support the length of the whole text string. In line 15 we can still store that in the variable we used in concatenation (text1). Another thing to note is that the declaration of all the variables is var and there is no difference from any of them at first. But when it runs the interpreter will create the types for each one of them. In the float(), int(), str(), bool() we can only put primitive types (string is considered primitive type) Shape, Location, BoundingRectangle, or a relative declarator is not supported and will throw an error at runtime.



```
I Truely love bacon, tuna and numbers: 13 , 13.031996
```

Figure 9 – Output of the Code in Figure 8

5. Paint++ Supports Built in Shapes

5.1 Line, Rectangle, and Oval.

Paint++ supports line, rectangle and oval as built in shapes. All of these shapes have different properties, however these properties are not given when the shapes are instantiated.

5.2 Common Syntax For Instantiating Shapes

Because of the fact that none of the properties are given when the shapes are instantiated, there exists a common syntax for instantiating all of the shapes. Also, since this programming language is dynamically typed, each shape is instantiating without specifying the type. The type is detected at runtime according to the drawing method parameters. Therefore each instance is a “var”. This syntax is as follows:

```
var rect1 = Rectangle(1)
var compositeshape =  Composite(arr)
var line = Line(NW, 2, 1) //1: dir, 2:arrow, 3:ratio of arrow to s.w.
var oval1 = Oval()
var str = “Hello!”
```

The variable rect1 will be a Rectangle shape in the when it is instantiated. The number and the type of parameters when the shapes are instantiated, helps the interpreter to understand which type it is. So the interpreter understands that a var is a Composite type according to the parameter’s type, which is an array. The interpreter understands that rect1 is a Rectangle, since, even though there is one parameter (like compositeshape), the parameter is of type Boolean.

5.2.1 Parameters

There are several optional and mandatory parameters while instantiating shapes. These parameters are for determining the properties of shape instances that cannot be specified in the future while drawing. However it would be beneficial to note that these parameters cannot specify the location and the size of the instantiated object. The parameters are as follows:

- Direction: The direction parameter is specified if the programmer wants to draw a line. There are 8 directions: North (N), South (S), East (E), West (W), North East (NE), North West (NW), South East (SE), and South West (SW). The programmer should specify the direction via the abbreviations while drawing a line.
- Arrows: This parameter is specified while drawing a line too. It indicates the start and end of a line to distinguish which end of the line will have the arrow. If this parameter is given 0, the arrow is in the starting end, if it's given 1, the arrow is in the ending end and if it is given 2, the arrow is in the both ends.
- Arrow Size: This parameter is specified while drawing a line too. It indicates the arrow size relative to the line stroke width. It is given as a ratio in decimal.
- Rounded Corners: This parameter is specified while drawing a rectangle. It specifies whether if the drawn rectangle has rounded corners or not. It is either TRUE (1) if the programmer wants rounded corners, or FALSE (0) if the programmer does not want rounded corners.

5.2.2 Scale – Free Shape Instances

The shape instances are scale – free, that is, the programmer does not specify the length, width, radius, x – coordinate nor y – coordinate while instantiating a shape. These properties are given in the future while drawing the shape via the draw method.

5.2.3 Bounding Rectangle

There exists a bounding rectangle for every shape instance when they are drawn. The BoundingRectangle takes two parameters when it is initialized. The first parameter is the Location object and the second one is the Size:

```
var sze = Size(width1, length1)
var loc = Location(xcoor, ycoor)
var br =  BoundingRectangle(loc, sze)
```

The BoundingRectangle instances are used while drawing the objects. They are put into the Draw function as a parameter.

6. The Language Should Support Basic Drawing Statements

This language supports a draw function that has varying parameters, to draw all the shapes. The type of the drawn shape is determined in the runtime, since Paint++ is a dynamically typed programming language. As a general rule of thumb, the Draw method takes the shape instance that is to be drawn as the first parameter.

6.1 Parameterization

The parameters of the draw functions are different for every shape, since every shape has different information to be provided before drawing. The Draw function's usages are given below for Rectangle, Oval, Line, String and Composite types:

- Rectangle:
`Draw(shape, boundingRectangle, strokeWidth, fillState, fillColor)`
- Oval:
`Draw(shape, boundingRectangle, strokeWidth, fillState, fillColor)`
- Line:
`Draw(shape, boundingRectangle, strokeWidth, fillColor)`
- Composite:
`Draw(shapeArr, boundingRectangle, strokeWidth, fillState, fillColor)`

For all the Rectangle, Oval and Composite types, if the fillState is given FALSE (0), the fillColor is ignored. Otherwise, for the Rectangle and Oval, all of the parameters are mandatory. For the Line and String, all of the parameters are mandatory. For the Composite type, strokeWidth, fillState and fillColor are optional parameters, because, if the programmer doesn't specify one or more of them, the interpreter will understand that it should take the colors or fill state or stroke width of the individual unit shapes that make up the composite shape. In all of the Draw functions, shape parameter is mandatory.

The draw function understands which shape to draw according to the shape parameter's type, which is determined via the different number of parameters when they are instantiated (i.e. one oval has no parameters, line has three parameters, rectangle has one parameter, composite has one parameter, which is an array). If the user wants to draw a Line with the Rectangle Draw method's parameter number, the parser raises an error message saying that the parameters are incompatible.

6.2 Drawing Strings

As well as drawing primitive shapes and composite shapes, drawing strings are also supported in this language. Strings are instantiated and drawn as follows:

```
var sze = Size(width1, length1)
var loc = Location(xcoor, ycoor)
var br =  BoundingRectangle(loc, sze)

var str = “Paint++ is awesome!”
Draw(str, boundingRectangle, fillColor)
```

Whereas the boundingRectangle is defined as the previous examples with the location and size, str is the string to be drawn and fillColor is the color of the string that is drawn. The str and boundingRectangle parameters are mandatory, however fillColor is optional. If fillColor is not specified, the default printed string will be in black.

7. The Language Supports Loops (while)

This language supports while loops to handle repetitive tasks. In this language the word while is reserved, meaning that it cannot be used as an identifier name, and when the word while is seen it is understood that it is a control flow statement and the statements between opening and closing curly brackets will be repeated while the condition inside opening and closing parenthesis that come after while is satisfied.

Syntax of while loops:

```
while(logical_expr)
{
        program
}
```

Here's an example about the usage of while loops in this program:

```
var size = Size(tbf,tbf)
var loc = Location(tbf,tbf)
var boundingRect = BoundingRectangle(loc, size)

var strokeWidth = tbf
var rect = Rect(1)

var comp
var arr

var z = 0

while( z < 24)
{
        while(z < 24)
        {
                arr[z] = Rect(1)
                z = z + 1

                arr[z] = Location(tbf+z,tbf+1)
                z = z + 1

                arr[z] = Size(tbf, tbf)
                z = z + 1
                arr[z] = tbf
                z = z + 1

                arr[z] = True
                z = z + 1

                arr[z] = Color(tbf, tbf, tbf)
            }

            comp = Composite(arr)
            Draw(comp, boundingRect, strokeWidth)
            z = z + 1
        }
```

This language allows nested "while" statements and in that case "while" block is understood as from the open brackets to the first unmatched closing brackets. So the second while loop covers adding elements to the array "arr" and the first "while" loop covers the second while loop and creating a new composite object and drawing it.

In the context of this language, this piece of code creates composite objects that consists of increasing number of rectangles and draws these objects.

8. The Language Supports Conditionals (if and if/else)

Our language supports if and if else statements. In our language the words if and else are reserved, meaning that it cannot be used as an identifier name, and when the words if and else are seen it is understood that it is a control flow statement and the statements between the opening and closing curly brackets of if or the else block will be executed depending on the logical expression that comes after the if word.

Syntax of if statements:

```
if(logical_expr)
{
        program
}
```

Syntax of if else statements:

```
if(logical_expr)
{
        program
}
else
{
        program
}
```

Here's an example about the usage of if else statements in our program:

```
var  isRect
var isLine
var shape

if(isRect)
{
        if(!isLİne)
        {
                shape = Rect(1)
        }
}
```

```
else
{
        if(isLine && !isRect)
        {
                shape = Line(NW,0,0)
        }
        else
        {
                shape = Oval
        }
}

var loc = Location(tbf,tbf)
var size = Size(tbf,tbf)
var boundingRect = BoundingRect(loc, size)

var strokeWidth = 1
var fillStatus = True
var color = Color(tbf,tbf,tbf)

Draw(shape, boundingRect, strokeWidth, fillStatus, color)
```

Our language allows nested if and if else statements. In these cases else blocks belong to the nearest if statements in the same scope. For example the last else block is in the first else blocks scope, and belongs to the if statement in the same scope and first else statement is in the main programs scope and belongs to the first if statement which is also in the same scope.

One thing to be noticed is that if and else blocks are always interpreted as the program that is enclosed by the opening curly brackets that comes after the if or else statement and the first unmatched closing curly brackets that will come after the opening curly brackets. So no matter the program to be written inside the statements is one line or not, it should always be in the opening and closing parenthesis, unlike many other programs.

In the context of our language, this block of code decides if the shape is a rectangle, a line or a circle and draws it. If the boolean variable isRect is true and and isLine is false, then the shape becomes a Rectangle if the isLine is true and isRect is false then the shape becomes a Line and if bot of them are true or false the shape becomes a circle.

9. The Language Supports Modularity At the Level Of Functions

Our language supports 3 types of functions:

1. A function to draw the shapes
2. Functions to create primitive type of shapes
3. A function to create composite type of shapes

In order to draw a shape first the shape must be created. The functions that create the shape can take optional parameters such as direction for the line, arrows and arrow size and a Boolean parameter for the rectangle that defines if the corners of the rectangle rounded or not. Composite function which takes an array as a parameter takes which shapes will be in the composite shape, their relative sizes and locations, their relative stroke width and their fill states and colors as parameters.

Draw function takes the shape and size, location and stroke width of the shape and draws the shape accordingly. If no parameters for the size and location are specified, it fits the shape to the window. If the shape is not composite it takes stroke width, fill state and fill color as well.

It is important to note that when a shape is created, it is independent of scale, meaning that it does not have a size and location, and these are specified in the draw function. Even when creating composite shapes, only relative sizes and locations of the shapes are determined, so the exact size and location of the composite shape is still unknown.

To sum, functions that create primitive type of shapes only handles the structure of the shape, and draw function for these shapes handles size and location of the shape as well as stroke width, fill state and fill color. For the composite shapes, only size, location and stroke width is handled by the draw function, because fill state and fill color of the primitive types that make up the composite object is specified while creating the composite object.

10. Extension of Shapes

Our language will have a function that is Composite(arr) for drawing composite shapes. Composite function takes an array as a parameter, which specifies the properties of the primitive types that will make up the composite shape. These parameters are:

1. Type of shape (ex. Rectangle, Circle or Line)
2. Relative location of the shape in the composite shape
3. Relative size of the shape
4. Relative stroke width of the shape
5. Fill status of the shape
6. Color of the shape

According to these parameters, Composite function creates the composite object.

Draw function takes different parameters for the composite objects compared to primitive objects. Since fill status and color of the primitive shapes are declared when creating the composite object, unless otherwise stated, these parameters are used when drawing the composite object. However if fill status and color is specified while calling the draw function, these will be applied to the whole object, overriding the values of the primitive types that make up the composite.

An example of creating and drawing composite objects is as follows:

```
var arr

arr[0] = Rectangle(1)
arr[1] = Location(tbf, tbf)
arr[2] = Size(tbf,tbf)
arr[3] = tbf
arr[4] = True
arr[5] = Color(tbf,tbf,tbf)

arr[6] = Circle
arr[7] = Location(tbf, tbf)
arr[8] = Size(tbf,tbf)
arr[9] = tbf
arr[10] = False
arr[11] = Color(tbf,tbf,tbf)
```

```
arr[12] = Line(NW,3,tbf)
arr[13] = Location(tbf, tbf)
arr[14] = Size(tbf,1)
arr[15] = tbf
arr[16] = Color(tbf,tbf,tbf)

var comp = Composite(arr)
var loc = Location(tbf,tbf)
var size = Size(tbf,tbf)
var boundingRect = BoundingRectangle(loc,size)

var strokeWidth = tbf

draw(comp, boundingRect, strokeWidth)
```

The code segment above first creates an array and then adds parameters of the composite object to it. One thing to be noticed is that line shapes does not have fill status so it is not added to the array. Then, Composite object "comp" is created by calling the Composite function with arr parameter. Finally "comp" is drawn by calling the draw method with parameters comp, location, size and stroke width. In this case fill status and color is not specified as parameters for the draw function, so fill status and color of the primitive objects are taken into consideration while drawing the shape.

11. Resources

[1]http://programarcadegames.com/index.php?chapter=introduction_to_graphics