AniFrame: A Programming Language for 2D Drawing and Frame-Based Animation

Mark Edward M. Gonzales
De La Salle University
Manila, Philippines
mark_gonzales@dlsu.edu.ph

Hans Oswald A. Ibrahim
De La Salle University
Manila, Philippines
hans_oswald_ibrahim@dlsu.edu.ph

Elyssia Barrie H. Ong
De La Salle University
Manila, Philippines
elyssia_ong@dlsu.edu.ph

Ryan Austin Fernandez
De La Salle University
Manila, Philippines
ryan.fernandez@dlsu.edu.ph

ABSTRACT

Creative coding is an experimentation-heavy activity that requires translating high-level visual ideas into code. However, most languages and libraries for creative coding may not be adequately intuitive for beginners. In this paper, we present AniFrame, a domain-specific language for drawing and animation. Designed for novice programmers, it *(i)* features animation-specific data types, operations, and built-in functions to simplify the creation and animation of composite objects, *(ii)* allows for fine-grained control over animation sequences through explicit specification of the target object and the start and end frames, *(iii)* reduces the learning curve through a Python-like syntax, type inferencing, and a minimal set of control structures and keywords that map closely to their semantic intent, and *(iv)* promotes computational expressivity through support for common mathematical operations, built-in trigonometric functions, and user-defined recursion. Our usability test demonstrates AniFrame's potential to enhance readability and writability for multiple creative coding use cases. AniFrame is open-source, and its implementation and reference are available at https://github.com/memgonzales/aniframe-language.

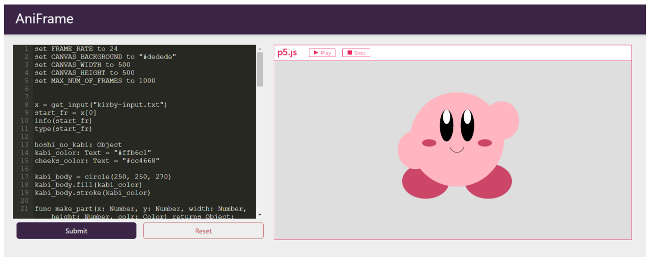
KEYWORDS

Creative coding, programming language design, exploratory programming, domain-specific language, animation

1 INTRODUCTION

Recent years have seen an increased interest in creative coding, or programming with an artistic rather than a functional intent, especially in engaging artistic expression and computational thinking among novice programmers [9, 10]. It has been described as an expressive and exploratory activity [2], with creative coders having to translate high-level visual ideas into code and go through multiple rounds of incremental refining (or even switching to another idea altogether) depending on the resulting output.

The difficulty of repeatedly mapping mental models to computer-compatible code is the crux of programming. Hence, from a human-centered viewpoint, programming languages and, by extension, libraries are not only notations or frameworks for writing code but, more importantly, user interfaces purposely designed to facilitate this mapping and make the programming task easier [17].



```
set FRAME_RATE to 24
set CANVAS_BACKGROUND to "#dedede"
set CANVAS_WIDTH to 500
set CANVAS_HEIGHT to 500
set MAX_NUM_OF_FRAMES to 1000


x = get_input("kirby-input.txt")
start_fr = x[0]
info(start_fr)
type(start_fr)

hoshi_no_kabi: Object
kabi_color: Text = "#ffb6c1"
cheeks_color: Text = "#cc4668"

kabi_body = circle(250, 250, 270)
kabi_body.fill(kabi_color)
kabi_body.stroke(kabi_color)

func make_part(x: Number, y: Number, width: Number,
    height: Number, colr: Color) returns Object:
```

Figure 1: Sample AniFrame Code and Resulting Output. Designed for novice programmers, AniFrame supports a varied set of features that simplify the creation of composite objects and provide fine-grained control over animation, thus facilitating expressivity and exploration for creative coding.

To this end, libraries dedicated to creative coding [12–14, 21, 31] have been developed to provide an ecosystem and set of constructs for drawing and animation. However, using a library presupposes proficiency in the language for which it was created. Some syntactic and semantic aspects may also not be readily intuitive for novice coders. Examples include p5.js' and Cinder's use of braces for block demarcation, which they carry over from JavaScript and C++, respectively [13, 14, 24]; p5.js' mechanics for layering drawn objects, which have been characterized as challenging [27]; and Cinder's interfacing with OpenGL, which can pose a steep learning curve for programmers with no prior experience in graphics APIs [16].

Meanwhile, few domain-specific languages for animation have been designed, with most of them, such as ActionScript 3.0 [29] and Processing [26], following an object-oriented programming paradigm. Although this results in highly structured code, having to create classes and instantiate them even for simple programs impedes rapid prototyping and has been noted to increase the difficulty of learning for beginning programmers [20].

In this paper, we present AniFrame, an open-source domain-specific language for two-dimensional drawing and frame-based animation for novice programmers. The language's core principles and features are as follows:

- *Ready Support for Animation-Specific Constructs.* AniFrame features animation-specific data types (e.g., for drawn objects and colors), operations (e.g., for mixing colors and simplifying the layering of objects into composite objects), and built-in functions for shapes and affine transformations.

- *Fine-Grained Control Over Animation.* AniFrame adopts a frame-based strategy where programmers explicitly specify the object to be animated, along with the start and end frames for the animation sequence. Settings such as the frame rate and the total number of frames can also be configured.
- *Reduced Learning Curve.* AniFrame follows a Python-like syntax, limits the number of keywords and control structures to a minimum, and tries to use keywords that are close to their semantic intent (e.g., `Text` instead of `string`). Specifying data types is optional since type inferencing is enforced.
- *Computational Expressivity.* AniFrame supports common mathematical operations, built-in trigonometric functions, and user-defined recursive functions. Their utility is demonstrated in creating self-similar patterns, such as fractals.

2 RELATED WORKS

Similar to traditional programming, creative coding involves translating ideas into a sequence of instructions for a computer to process. However, it differentiates itself in that its goal is the expression of art rather than the development of functional software [9]. It thus opens a variety of functions, ranging from recreational (e.g., making digital sketches and installations [4]) to practical (e.g., introducing coding concepts to novice and non-programmers [10]) intents.

With the increased interest in creative coding in recent years, various tools have been developed to ease the coding process and support creativity among programmers and artists alike [4]. These tools can be broadly categorized into domain-specific languages, desktop-based libraries, and web-based libraries [27].

Among the few domain-specific languages that have been developed, the most widely used are Processing [26] and ActionScript [29]. Processing is a simplification of Java for animation. ActionScript was created to accompany the now-discontinued Adobe Flash, and, with its latest release (version 3.0), it became a superset of ECMAScript. Both follow an object-oriented paradigm (OOP), thus promoting reusability and modularity; however, this may be overly complicated for simple drawing and animation use cases, as well as cognitively demanding for novice programmers [20, 33].

Libraries for creative coding were developed later on, as languages and browsers extended support for more features and improved their portability. Desktop-based libraries include vvvv [5], which supports more sophisticated functionalities such as 3D rendering and machine learning but runs only on .NET ecosystems; C4 [12], which provides a simplified API for mobile-specific features but runs only on iOS; and Cinder [13], which is cross-platform but can be difficult to learn for those without prior experience with OpenGL [27]. Web-based libraries, such as p5.js [14] and Sketch.js [31], are mostly written in JavaScript [27] and, therefore, presuppose knowledge of JavaScript, which may not be suited for beginners due to its global variable-based programming model [6].

While these tools have proved helpful in a variety of use cases, most domain-specific languages for creative coding assume familiarity with OOP, and libraries require experience with their base ecosystem or language, thereby increasing the learning curve for beginners. AniFrame thus borrows design elements from these tools and also attempts to improve accessibility for novice coders, while still maintaining expressivity for more elaborate use cases.

3 LANGUAGE DESIGN

AniFrame's design is grounded in *(i)* ready support for animation-specific constructs, *(ii)* fine-grained control over animation, *(iii)* reduced learning curve, and *(iv)* computational expressivity.

3.1 Animation-Specific Features

The domain-specificity of AniFrame derives from its animation-specific features, as reflected in its data types, operations, built-in functions, and frame-based approach to animation.

3.1.1 *Data Types and Operations.*

AniFrame features three standard data types: *(i)* `Text` for strings, *(ii)* `Number` for floating-point and integer values, and *(iii)* `List` for collections of data that are possibly heterogeneous (i.e., of different data types). Moreover, it supports three domain-specific types: *(i)* `Object` for shapes and composite objects, *(ii)* `Color` for colors (which can be initialized in either hex or RGB), and *(iii)* `Coord` for coordinate pairs.

Selected operations between these domain-specific types are also permitted. For example, adding two objects creates a composite object, where the second operand is layered on top of the first operand. Adding two colors is equivalent to color mixing, while subtracting two colors is equivalent to color subtraction. Operations on coordinates are also defined in a component-wise fashion.

On a syntactic note, the word choices for data types were selected to be as intuitive and close as possible to their semantic intent; for instance, `Text` was chosen instead of `string`, and `Number` instead of `float` or `int`, following the results of the empirical study conducted by Stefik and Siebert [30] among novice programmers.

3.1.2 *Built-In Functions.*

AniFrame provides a varied set of built-in functions for 2D drawing and animation, as listed in Table 1.

The largest class of built-in functions comprises those for drawing, styling, and animating objects. To facilitate rapid prototyping, each animation function has a version that applies the affine transformation only on the x-axis, only on the y-axis, and on both axes. In terms of syntax, their names purposely depart from their formal mathematical terminologies (e.g., `move`, `turn`, and `resize` instead of `translate`, `rotate`, and `scale`) in order to make their semantic intent more readily understood even by programmers without specialized mathematical background.

Common mathematical functions (e.g., square root and pseudo-random number generation) and trigonometric functions, which are important in programmatic animation [22], are also available out of the box for computational expressivity.

Moreover, in order to assist in debugging, the built-in method `info()` can be called to display values or, in the case of objects, their internal representations (discussed in Section 4.3), while `type()` can be invoked to display the data types of variables.

3.1.3 *Frame-Based Animation.*

AniFrame adopts a frame-based approach to animation, where *(i)* all the built-in functions have required parameters for specifying the start and end frames, and *(ii)* an animation is applied only to the target object calling it. For example, to move an object `x` by 3 units to the right starting at frame 10 and ending at frame 20, the code is `x.moveX(3, 10, 20)`.

This approach affords the programmer fine-grained control over animation sequences while still allowing for rapid prototyping since the appearance and position of the object in the in-between frames are automatically computed under the hood. In this regard, AniFrame takes after the idea of keyframes and tweening in Flash.

AniFrame's principle of applying an animation only to the target object calling it attempts to address pain points in the stack-based approach of OpenGL (and later inherited by Processing and p5.js), where an animation is applied to all the objects created subsequent to the animation function call unless reset via `pop()` [14, 15, 26].

3.2 Control Structures

AniFrame is a Turing-complete language in that it has sequential, conditional, and iterative control structures. With regard to conditionals, two-way selection using `if...else` and multiple selection using `else if` are supported. The decision to have only a single multiple selection structure (i.e., not supporting `switch...case` for instance) is deliberate; AniFrame tries to limit the number of keywords and control structures to a minimum in order to maintain a low learning curve for novice programmers.

AniFrame features three classes of iterative control structures: *(i)* precondition-controlled loops using `while`, *(ii)* count-controlled loops using `repeat`, and *(iii)* collection-controlled loops using `for...in`. Including a dedicated count-controlled loop is an attempt to increase readability and writability for use cases such as repeating an object's movement a predefined number of times. A `break` statement is also provided to prematurely escape loops. However, unconditional branching (e.g., via `goto`) is not supported in order to discourage "spaghetti" control flows [32].

3.3 Type System and Scoping Rules

AniFrame follows a static type system with support for both explicit and implicit type declaration (the latter via type inferencing). The motivation for preferring a static over a dynamic type system is to promote increased reliability and code maintainability [11]. Empirical results [23] have also pointed towards the advantage of static type systems for performing tasks that involve working with previously unknown API functions; in principle, this may also apply to AniFrame since a significant part of the initial learning curve entails developing familiarity with its built-in functions.

In terms of scoping, AniFrame follows static (lexical) scoping. Creating user-defined functions is supported, but nested functions are not allowed. Any variable declared outside a function is considered a global variable. A function can read the value of a global variable but cannot modify it, except when it is an object; this exception makes it easier to modularize the creation of composite objects by eliminating the need to pass the base object as a parameter (as demonstrated in Section 5.1). When a local variable inside a function shares the same name as that of a global variable, the global variable cannot be read inside that function. Since the target users of AniFrame are novice programmers, these restrictions are set to simplify the semantics of global scoping and prevent often-unintended side effects when working with global variables [25].

Listing 1: Sample Code Snippet for Demonstrating AniFrame's Implementation

```
nose: Object = rectangle(-650, 250, 800, 15)
nose.fill("#C2B280")
frame = 1
repeat 3:
    nose.moveX(20, frame, frame + 100)
    frame += 101
pinocchio: Object = circle(0, 250, 200)
pinocchio += nose
```

Syntactically, blocks are demarcated via indentations; in this regard, AniFrame takes after Python, which is well regarded for its readability and writability [1].

4 LANGUAGE IMPLEMENTATION

AniFrame is an interpreted language. To maintain a strict separation of concerns, its implementation is divided into three stages: *(i)* lexical analysis, *(ii)* parsing, and *(iii)* semantic analysis. The handling modules, namely the lexer (for lexical analysis), parser (for parsing), and interpreter (for semantic analysis), were written in Python using the language recognition tool ANTLR 4 [18].

4.1 Lexical Analysis

First, the lexer performs lexical analysis, converting the source code into a list of tokens following a longest-match-wins strategy [18] and stripping out comments. For illustration, a sample code snippet and its partial token stream are given in Listing 1 and Table 2, respectively. The patterns for the tokens are defined using regular expressions. In addition, since AniFrame borrows Python's indentation-based block demarcation, special tokens for indentations and outdentations are generated following the stack-based algorithm described in the Python Language Reference [8].

4.2 Parsing

After the lexical analysis, the parser creates a parse tree (Figure 2). This parser is generated via ANTLR's adaptive LL(*) parsing strategy, a predictive approach that involves an arbitrary lookahead and the launching of multiple pseudo-parallel subparsers for efficiency [19]. AniFrame's grammar is expressed using a variant of extended Backus-Naur form.

4.3 Semantic Analysis and Code Generation

After the parse tree is generated, the interpreter traverses it in order to determine the semantic intent of the statements in the source code. This traversal can be performed using either ANTLR's listener or visitor interface [18], but since it is necessary to return the results of visiting certain nodes (e.g., to update the symbol table) and propagate certain values up the parse tree, AniFrame's implementation employs the visitor pattern.

As the parse tree is traversed and expressions are resolved, the interpreter also builds a JSON-like intermediate code that consists

Table 1: Built-In Functions in AniFrame

Category	Functions
Shapes	`point()`, `line()`, `curve()`, `circle()`, `ellipse()`, `triangle()`, `rectangle()`, `quad()` (for quadrilaterals), `polygon()`, `write()` (for text boxes)
Styling	`fill()`, `stroke()`
Translation	`move()`, `moveX()`, `moveY()`
Rotation	`turn()`, `turnX()`, `turnY()`
Scaling	`resize()`, `resizeX()`, `resizeY()`
Shear	`shear()`, `shearX()`, `shearY()`
General Math	`rand_num()`, `rand_int()`, `sqrt()`
Trigonometry	`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, `to_deg()`, `to_rad()`
Miscellaneous	`draw()` (for placing objects on the canvas), `info()` (for displaying values or, in the case of objects, their internal representations), `type()` (for displaying data types)

Table 2: Partial Token Stream for Listing 1

Line #	Column #	Token	Lexeme
1	1	IDENTIFIER	nose
1	5	COLON	:
1	7	OBJECT_TYPE	Object
1	14	ASSIGNMENT_OP	=
1	16	RECTANGLE	rectangle
1	25	OPEN_PARENTHESIS	(
1	26	MINUS_OP	-
1	27	INTEGER_LITERAL	650
...	...	...	...

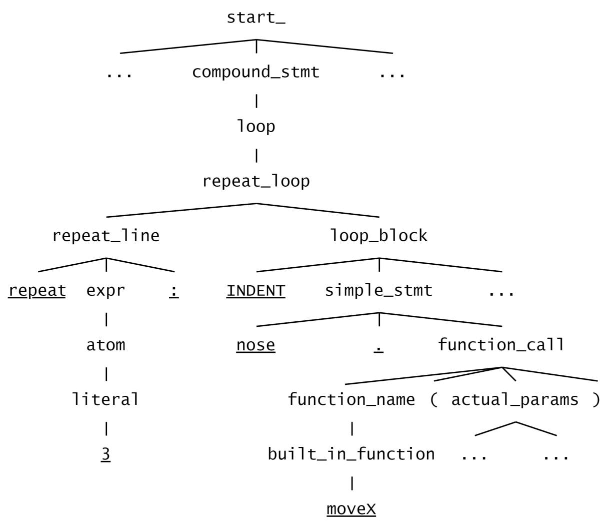


Figure 2: Partial Parse Tree for Listing 1. This figure focuses on the `repeat` loop (Lines 4 to 6). Terminals are underlined.

of two dictionaries: Drawing and Animation (Listing 2). Drawing stores the drawn objects, their constituent shapes, and the styles (i.e., their stroke and fill colors) of each constituent shape. Animation

Listing 2: Intermediate Code Generated for Listing 1

```
"Drawing": {
  "nose": [{"fill": "#C2B280",
    "stroke": "DEFAULT_STROKE", "shape":
    "rectangle(-650, 250, 800, 15)"}],
  "pinocchio": [{"fill": "DEFAULT_FILL",
    "stroke": "DEFAULT_STROKE", "shape":
    "circle(0, 250, 200)"}, {"fill": "#C2B280",
    "stroke": "DEFAULT_STROKE", "shape":
    "rectangle(-650, 250, 800, 15)"}]
}
"Animation": {
  "nose": [{"action": "moveX", "start": 1,
    "end": 101}, {"action": "moveX", "start": 102,
    "end": 202}, {"action": "moveX", "start": 203,
    "end": 303}]
}
```

stores the drawn objects, the animations applied to them, and the start and end frames for each animation.

The rationale for this internal representation is twofold. First, although programmers do not have to be aware of this under-the-hood representation, AniFrame provides a function to view it to facilitate debugging, especially for more experienced users. Hence, a conceptually intuitive and human-readable representation is preferable. Second, since the Drawing and Animation dictionaries are updated as nodes in the parse tree are visited, utilizing a data structure that supports $O(1)$ lookups and updates is critical to speeding up the intermediate code generation.

After the entire parse tree is traversed and the intermediate code is generated, the interpreter converts the intermediate code to the target code. The target code is in JavaScript and follows p5.js' semantics to allow the output to be displayed on web browsers using its player. To this end, each entry in the Drawing dictionary is converted to a class. The entries in the Animation dictionary are processed to yield a series of conditionals, each corresponding to a sequence of frames of continuous, identical animation.

4.4 Error Handling

Separation of concerns is also observed in error handling. Lexical errors (e.g., a token not matching any of the patterns in the lexer grammar) are caught by the lexer during lexical analysis. Syntax errors (e.g., mismatched parentheses) are caught by the parser during syntactic analysis. To complement ANTLR's default error recovery strategy and to output more tailored error messages, the parser grammar also includes special production rules for catching common errors, such as specifying an incorrect number of coordinates in a `Coord` pair. Semantic errors (e.g., incompatible operands) are caught by the interpreter during semantic analysis.

The impact of error messages on the programming experience of novice programmers has been emphasized in human-computer interaction studies [3, 7, 28]. To maximize the helpfulness and utility of error messages, AniFrame attempts to avoid terse, technical phrasing; for example, instead of "unsupported operand type(s) for +: `'int'` and `'str'`", AniFrame displays "+ operator between `Number` and `Text` is not supported." Line numbers of errors are displayed, along with column numbers for lexical and syntactic errors. AniFrame also takes advantage of ANTLR's adaptive LL(*) parsing and error recovery strategy, which are purposely designed to reduce cascading error messages [18, 19].

5 SAMPLE USE CASES

We demonstrate the utility and expressivity of AniFrame as a domain-specific language for creative coding through two sample use cases: animating a composite object and drawing a Sierpiński triangle via recursion.

5.1 Animating a Composite Object

The code for creating and animating a composite object in AniFrame and the output are provided in Listings 3 to 5 and Figure 3, respectively. As discussed in Section 3.1.1, the semantics of the addition operator simplifies the layering of objects into composite objects (Lines 9, 23, and 37 in Listing 3). It is also possible to create reusable object templates via user-defined functions (Listing 4 and Lines 17, 18, 20, and 21 in Listing 3).

The stacking of objects on the canvas is controlled by the order of the `draw()` statements in Lines 43 to 50 in Listing 3; the parameters of these statements pertain to the start and end frames of the objects' appearance on the canvas. Meanwhile, Listing 5 shows how custom animations can be created using AniFrame's built-in animation functions and control structures.

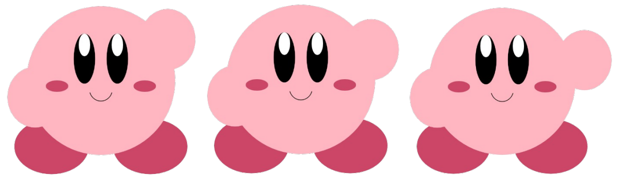
Other language constructs demonstrated in Listing 3 include explicit type declaration (Lines 7 and 8), type inferencing (Lines 1 and 2), and type coercion (Lines 1, 3, and 4; the inferred data type of `kabi_color` is `Text`, but it was implicitly converted to `Color` when passed as an argument to `fill()` and `stroke()`).

5.2 Drawing with Recursion

To show the computational expressivity of AniFrame, we use it to programmatically generate a Sierpiński triangle, a fractal with the overall shape of an equilateral triangle. It is recursively constructed by connecting the midpoints of an equilateral triangle, removing the central triangle formed, and repeating these steps for the remaining (smaller) equilateral triangles.

Listing 3: Sample Code for Creating a Composite Object

```
1  kabi_color = "#ffb6c1"
2  kabi_body = circle(250, 250, 270)
3  kabi_body.fill(kabi_color)
4  kabi_body.stroke(kabi_color)
5
6  cheeks_color = "#cc4668"
7  l_cheek: Object = ellipse(170, 260, 40, 20)
8  r_cheek: Object = ellipse(330, 260, 40, 20)
9  kabi_cheeks: Object = l_cheek + r_cheek
10 kabi_cheeks.fill(cheeks_color)
11 kabi_cheeks.stroke(cheeks_color)
12
13 # NOTE: Insert function definition of make_part(),
     as given in Listing 4
14
15 black = "#000000"
16 white = "#ffffff"
17 l_eye = make_part(220, 210, 38, 90, black)
18 l_eye_shine = make_part(220, 185, 20, 40, white)
19 l_eye += l_eye_shine
20 r_eye = make_part(280, 210, 38, 90, black)
21 r_eye_shine = make_part(280, 185, 20, 40, white)
22 r_eye += r_eye_shine
23 kabi_eyes: Object = l_eye + r_eye
24
25 smile: Object = curve(230, 150, 230, 272, 270,
   272, 270, 150)
26 smile.stroke(black)
27 smile.fill(kabi_color)
28
29 l_foot = make_part(160, 370, 135, 100,
   cheeks_color)
30 r_foot = make_part(340, 370, 135, 100,
   cheeks_color)
31
32 # NOTE: Insert function definition of make_hand(),
     as given in Listing 4
33
34 r_hand = ellipse(130, 280, 100, 110)
35 l_hand = make_hand()
36
37 hoshi_no_kabi: Object = r_hand + kabi_body
38 hoshi_no_kabi.fill(kabi_color)
39 hoshi_no_kabi.stroke(kabi_color)
40
41 # NOTE: Apply animation on l_hand (Listing 5)
42
43 l_foot.draw(1, 1000)
44 r_foot.draw(1, 1000)
45 l_hand.draw(1, 1000)
46 hoshi_no_kabi.draw(1, 1000)
47 l_eye.draw(1, 1000)
48 r_eye.draw(1, 1000)
49 kabi_cheeks.draw(1, 1000)
50 smile.draw(1, 1000)
```



(a) Frame 1 **(b) Frame 9** **(c) Frame 38**

Figure 3: Composite Object Drawn and Animated in AniFrame. The figure (a character sprite waving its left hand) shows selected frames from the output of the code in Listings 3 to 5.

Listing 4: User-Defined Drawing Functions in Listing 3

```
func make_part(x: Number, y: Number, w: Number,
h: Number, color: Color) returns Object:
    part: Object = ellipse(x, y, w, h)
    part.fill(color)
    part.stroke(color)
    return part

func make_hand() returns Object:
    sugoi = ellipse(360, 160, 100, 110)
    sugoi.fill(kabi_color)
    sugoi.stroke(kabi_color)
    return sugoi
```

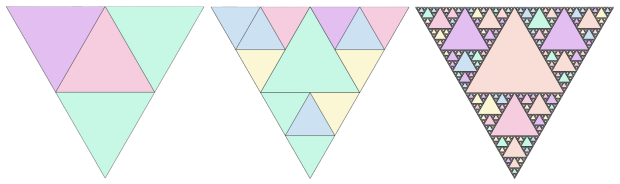
Listing 5: Animating the Left Hand (l_hand) in Listing 3

```
func wave_hand(frame: Number, delta: Number):
    repeat(3):
        l_hand.move(1, 2, frame, frame + delta)

func wave_hand_up(frame: Number, delta: Number):
    repeat(3):
        l_hand.move(-1, -2, frame, frame + delta)

frame = 1
delta = 12
repeat(3):
    wave_hand(frame, delta)
    wave_hand_up(frame + delta, delta)
    frame += 2 * delta
```

The code for creating a Sierpiński triangle in AniFrame and the resulting output are provided in Listing 6 and Figure 4, respectively. Aside from recursion, this code also demonstrates other constructs in the language, including lists (Line 1 in Listing 6), support for common mathematical operations (`rand_int()` in Line 6 and `sqrt()` in Line 14), and global scoping for objects (Lines 7 and 13; global scoping is discussed in Section 3.3).

Listing 6: Sample Code for Creating a Sierpiński Triangle

```
1   colors = ["#CCE1F2", "#C6F8E5", "#FBF7D5",
    "#F9DED7", "#F5CDDE", "#E2BEF1"]
2
3   func sierpinski(x1: Number, y1: Number,
    x2: Number, y2: Number, x3: Number, y3: Number,
    step: Number):
4       if step != 0:
5           shape = triangle(x1, y1, x2, y2, x3, y3)
6           shape.fill(colors[rand_int(0, 5)])
7           triangles += shape
8
9           sierpinski(x1, y1, (x1+x2)/2, (y1+y2)/2,
    (x1+x3)/2, (y1+y3)/2, step-1)
10          sierpinski(x2, y2, (x1+x2)/2, (y1+y2)/2,
    (x2+x3)/2, (y2+y3)/2, step-1)
11          sierpinski(x3, y3, (x3+x2)/2, (y3+y2)/2,
    (x1+x3)/2, (y1+y3)/2, step-1)
12
13  triangles: Object = point(0, 0)
14  sierpinski(0, 0, 600, 0, 300, 300*sqrt(3), 8)
15  triangles.draw(1, 100)
```



(a) Step = 2 **(b) Step = 3** **(c) Step = 8**

Figure 4: Sierpiński Triangle Drawn in AniFrame. The figure was generated programmatically using recursion. The code is given in Listing 6; the step is dictated by the last argument in Line 14.

6 USABILITY TEST

In order to have an initial assessment of AniFrame's readability and writability, we conducted a preliminary usability test. Among the six respondents, three had taken an introductory programming course (covering variables, conditionals, and loops) but have been coding for less than four months; the other three have at least one year of programming experience.

The respondents were asked to complete five programmatic drawing and animation tasks (Table 3) in both AniFrame and p5.js. All six respondents had no previous exposure to AniFrame and p5.js and instead referred to their respective documentation while doing the tasks; in order to mitigate the influence of the documentation, we patterned AniFrame's documentation after that of p5.js. The coding environment was a browser-based text editor without autocomplete. No time limit was set, but the respondents were allowed to give up if they felt that a task was too difficult.

Table 3: Tasks Given in the Usability Test. U1, U2, and U3 had taken an introductory programming course but have been coding for less than four months. U4, U5, and U6 have at least one year of programming experience. A tick mark (✓) indicates that the respondent was able to accomplish the task.

#	Task	Intent Tested	AniFrame						p5.js					
			U1	U2	U3	U4	U5	U6	U1	U2	U3	U4	U5	U6
1	Draw a circle and make it move from left to right	Drawing a shape and applying a single transformation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	Display the text "Hello world" on a blue canvas	Working with `Text`, drawing a text box, and setting canvas-level configurations	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
3	Draw a triangle and make it move up and down thrice at 5 frames per second	Drawing a simple shape and applying multiple transformations	✓	✓		✓	✓	✓					✓	✓
4	Draw a traffic light by creating red, green, and yellow signal lights and enclosing them in a rectangular border	Working with `Color` and drawing a composite object	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
5	Draw a simple face with eyes, nose, and mouth, and make the eyes (and only the eyes) move from left to right	Drawing a composite object and applying a transformation to only a component of it	✓	✓		✓	✓	✓			✓	✓	✓	✓

As seen in Table 3, all six respondents were able to complete Tasks 1 and 2 in both AniFrame and p5.js. Task 4 was completed by all six respondents in AniFrame and by five respondents in p5.js. On the other hand, Task 3 was accomplished by five respondents in AniFrame but only by two respondents in p5.js (both of which have at least one year of programming experience). Lastly, Task 5 was accomplished by five respondents in AniFrame and by four respondents in p5.js.

Task 3 involved making a triangle move up and down thrice. All six respondents first tried searching for a "move" function in the documentation but did not obtain any hits in p5.js since it uses "translate". In relation to this, U2, U4, and U5 mentioned that they appreciate AniFrame's use of the less technical term "move" since it is more accessible for beginners.

Coding in p5.js, U2 and U4 were able to move the triangle unidirectionally via `translate()` and `Vector` but were unable to implement the required change in direction. U5 did not use these constructs and instead opted for a frame-by-frame approach, manually adjusting the y-coordinate of the triangle per frame (Listing 7). U3 followed a similar approach but was unable to limit the motion to three times since they coded a loop for the counter whereas the semantics of p5.js require a conditional (Line 8 of Listing 7).

Meanwhile, coding in AniFrame, all six respondents utilized the built-in `moveY()` function (Listing 8). The respondents found the task to be easier in AniFrame (U1, U2, U4, U5, and U6), noting that it led to "significantly more readable and fewer lines of code" (U1) and required "less mental gymnastics since the `repeat` loop and `moveY()` work as expected compared to p5" (U2). However, one of the respondents (U3) was unable to finish the task since they did not adjust the `frame` parameter of the `moveY()` statement that corresponds to the change in direction.

Listing 7: p5.js Code of U5 for Task 3 of Usability Test (Moving a Triangle Up and Down Thrice)

```
1  function setup() {
2      createCanvas(720, 400);   frameRate(5);
3  }
4
5  y = 75;   delta = 5;   ctr = 3;
6
7  function draw() {
8      if (y <= 200 && y >= 0 && ctr + 1 >= 0) {
9          background(200);
10         triangle(30, y, 58, y-55, 86, y);
11         y += delta;
12     } else {
13         delta *= -1;   y += delta;   ctr--;
14     }
15 }
```

Task 5 involved drawing a simple face (with eyes, nose, and mouth) and animating only the eyes. Coding in p5.js, U2 was able to draw all the required facial features but, since they drew the eyes before the other features, the animation intended for the eyes cascaded down to the other features as a result of the stack-based semantics of p5.js (Section 3.1.3). This pain point was not observed in AniFrame since it was purposely designed such that an animation is applied only to the target object calling it. Moreover, the AniFrame code was described to be easier to write and trace since the target object is clearly specified when performing the animation function call (U2, U4, and U5).

Listing 8: AniFrame Code of U5 for Task 3 of Usability Test (Moving a Triangle Up and Down Thrice)

```
set FRAME_RATE to 5
i=0
repeat(3):
    shape = triangle(30, 75, 58, 20, 86, 75)
    shape.moveY(20, 1+i, 80+i)
    shape.moveY(-20, 81+i, 160+i)
    i += 160
shape.draw(1, 1000)
```

From the post-task semi-structured interview and the respondents' impressions, all six respondents cited AniFrame's readability as a major contributing factor to its suitability for beginning programmers. Its smaller set of functions also made it "less intimidating and a good starting point for animation" (U6).

With regard to the points for improvement, U3 and U5 noted that, although the frame-based approach allowed for fine-grained control, having to specify the start and end frames for every animation function call may not be readily intuitive for novice programmers and for those unfamiliar with the concept of frames in the first place. U2 suggested augmenting the coding environment with a visual representation of the frames vis-à-vis the animation timeline. Another recommendation was to allow for the canvas dimensions, canvas background, and frame rate to be accessible as special variables.

7 CONCLUSION

In this paper, we present AniFrame, an open-source domain-specific language for two-dimensional drawing and frame-based animation for novice programmers. Its design can be characterized as follows:

First, it features animation-specific data types, operations, and built-in functions for rapid creation and animation of composite objects. Second, it allows for fine-grained control over animation sequences through explicit specification of the target object, alongside the start and end frames. Third, it attempts to reduce the learning curve by adopting a Python-like syntax, supporting type inferencing, and using keywords that map closely to their semantic intent. Fourth, it promotes computational expressivity through built-in mathematical functions and support for recursion.

Our usability test points to AniFrame's potential to facilitate increased readability and writability for creative coding applications. Future directions include exploring syntactic and semantic improvements to enhance the intuitiveness of the frame-based approach, as well as providing support for three-dimensional graphics.

REFERENCES

[1] Zahin Ahmed, Farishta Jayas Kinjol, and Ishrat Jahan Ananya. 2021. Comparative Analysis of Six Programming Languages Based on Readability, Writability, and Reliability. In *2021 24th International Conference on Computer and Information Technology (ICCIT)*. 1–6.

[2] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Article 100.

[3] Brett A. Becker, Paul Denny, James Prather, Raymond Pettit, Robert Nix, and Catherine Mooney. 2021. Towards Assessing the Readability of Programming Error Messages. In *Proceedings of the 23rd Australasian Computing Education Conference*. 181–188.

[4] A. Benedetti, T. Elli, and M. Mauri. 2020. "Drawing with Code": The experience of teaching creative coding as a skill for communication designers *(12th International Conference on Education and New Learning Technologies)*. 3478–3488.

[5] Andrey Bo. 2012. Linear Spread: Illustrated Guide to vvvv for Newbies in Computer Arts. https://visualprogramming.net/.

[6] Douglas Crockford. 2008. *JavaScript: The Good Parts*. O'Reilly Media, Inc.

[7] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Article 55, 15 pages.

[8] Python Software Foundation. [n. d.]. The Python Language Reference - 2. Lexical analysis. https://docs.python.org/3/reference/lexical_analysis.html.

[9] Ira Greenberg, Deepak Kumar, and Dianna Xu. 2012. Computational art and creative coding: teaching CS1 with Processing. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (North Carolina, USA). 660.

[10] Ira Greenberg, Deepak Kumar, and Dianna Xu. 2012. Creative coding and visual portfolios for CS1. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. 247–252.

[11] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Softw. Engg.* 19, 5 (Oct 2014), 1335–1382.

[12] Travis Kirton. 2013. C4: creative coding for iOS. In *International Conference on Tangible, Embedded, and Embodied Interaction*.

[13] Rui Madeira and Dawid Gorny. 2013. *Cinder Creative Coding Cookbook*. Packt.

[14] L. McCarthy, C. Reas, and B. Fry. 2015. *Getting Started with p5.js: Making Interactive Graphics in JavaScript and Processing*. Make Community, LLC.

[15] Microsoft. [n. d.]. OpenGL - glPopMatrix function. https://learn.microsoft.com/en-us/windows/win32/opengl/glpopmatrix.

[16] James R. Miller. 2014. Using a software framework to enhance online teaching of shader-based OpenGL. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. 603–608.

[17] Leonel Vinicio Morales Díaz. 2010. Programming languages as user interfaces. In *Proceedings of the 3rd Mexican Workshop on Human Computer Interaction* (San Luis Potosí, Mexico) *(MexIHC '10)*. 68–76.

[18] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.

[19] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. *SIGPLAN Not.* 49, 10 (Oct 2014), 579–598.

[20] Nicolás Passerini and Carlos Lombardi. 2020. Postponing the Concept of Class When Introducing OOP. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway). 152–158.

[21] Denis Perevalov. 2013. *Mastering openFrameworks: Creative Coding Demystified*. Packt Publishing.

[22] Keith Peters. 2006. *Trigonometry for Animation*. Apress, Berkeley, CA, 41–68.

[23] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. 2014. An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. 212–222.

[24] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct 2017).

[25] Anthony Ralston. 2003. *Side effect*. John Wiley and Sons Ltd., GBR, 1573–1574.

[26] C. Reas and B. Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. NBER.

[27] Emil Sandberg. 2019. Creative Coding on the Web in p5.js: A Library Where JavaScript Meets Processing.

[28] Eddie Antonio Santos. 2022. What makes a programming error message good?. In *Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education Research (UKICER '22)*. Article 21, 1 pages.

[29] Rich Shupe and Zevan Rosser. 2010. *Learning ActionScript 3.0*. O'Reilly Media.

[30] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (Nov 2013), 40 pages.

[31] Justin Windle. [n. d.]. Sketch.js. https://soulwire.github.io/sketch.js/.

[32] W. A. Wulf. 1979. *A case against the GOTO*. Yourdon Press, USA, 83–98.

[33] Stelios Xinogalos. 2015. Object-Oriented Design and Programming: An Investigation of Novices' Conceptions on Objects and Classes. *ACM Trans. Comput. Educ.* 15, 3, Article 13 (Jul 2015), 21 pages.