THAPL
Theaterical Programming Language

Matan I. Peled
Department of Computer Science
Technion—Israel Institute of Technology
mip@cs.technion.ac.il

Research Proposal
Advised by Prof. Yossi Gil

Abstract

We propose to develop THAPL, a programming language for generating animations in presentations. THAPL draws inspiration from the literary format of a theater play, for instance William Shakespeare's [19] plays.

THAPL focuses on animations of the kind usually found in slideshows, where elements appear, disappear, move around, and so on.

1 Introduction

The purpose of this research is to explore an innovative approach to the declarative/imperative paradigm of programming languages. To demonstrate this approach, we propose to develop a prototype for domain-specific lanuage inspired by the scripts of theatrical plays (hence dubbed THAPL), which is intended to be used in the context of generating animations for use in presentations ("slide-shows"). The Theater play metaphor encompasses the concept of a classical theater play script (e.g. Shakespearian play), enumerating the 'dramatis personæ', scenery, text, and actions (e.g. 'exit chased by a bear').

THAPL attempts to expand the declarative/imperative paradigm by introducing an actor/action model, and specialized constructs to describe actions occuring simultaneously.

Examples of domains where THAPL may be useful include academic courses (especially those that deal with graphs and algorithms), new product presentations, fiscal reviews, and in general and domain where it is helpful to visualize concepts by moving and transforming objects on screen.

For a first look at a THAPL program, please examine Fig. 1. We see an implementation for a basic presentation, which is comprised of elements appearing in sequence.

Vision Briefly: Take the play metaphor and play with it in the context of generating presentations ("slide-shows").

The "Chicken Chicken" presentation in Fig. 1, used here as a "lorem ipsum", is a very simple one, with limited transitions. Nonetheless it does contain a few slides with figures that have elements that appear in sequence. The code shown models this sequence.

Fig. 1 THAPL implementation of the Chicken Chicken [24] presentation.

```
Atomic actions:
  show.
  hide.
  color.

Act Chicken:
  Dramatis Personae:
    ChickenChickenChickenBox:
      ChickenDB.
    ChickenQuestion.
  Scenery:
    ChickenFlowChart.

  Action:
    show chickenDB then show chickenChickenChickenBox.
    # "over" is an example of an elaborator.
    show chickenQuestion over chickenFlowChart.

Act ChickenChicken:
  Dramatis Personae:
    Chickens2.
    Chickens4.
    Chickens8.
    Chickens16.

  Action:
    # "||" is a short form of "meanwhile"
    show chickens2 || color chickens2: red.
    show chickens4 || color chickens4: green.
    show chickens8 || color chickens8: blue.
    show chickens16 || color chickens 16: purple.
```

Here and henceforth, our listings use the following conventions:

1. Keywords, such as **meanwhile**, are typed in blue boldface.
2. Comments, for instance "*# note this*", are typed in green slanted lettering.
3. Other parts of code, such as identifiers, are rendered in **sepia boldface**.

The **Atomic actions** clause is declaring the actions that are used later on, and can be ignored for the purpose of this introduction. The code is divided into acts denoted by the **Act** keyword, where each act specifies an animation. The acts shown here are presumed to be executed by an external source that will embed them in the proper place in the presentation.

Each act contains additional clauses. **Dramatis personae** is a list of actor entities that can be part of actions, while

`Scenery` is list of static entities that remain in the same place throughout the animation, but can be referenced in actions (so that actions can be done relative to them). The last part, marked with the `Action` keyword, is an execution part, and contains the steps to create the animation. Each line in the action part is done in sequence, unless otherwise directed. For example, using the `meanwhile` operator actions can be performed in parallel. This is discussed further in Sect. 2.2.

THAPL is essentially an imperative language, but it has no variables, nor procedures. Instead it uses actors and actions on those actors, and acts to organize actions and actors together. Syntactically, it uses whitespace to denote blocks, like PYTHON [11], and not curly brackets ("{") as introduced by BCPL [17] and made popular by C [9]. Additionally it has several syntactic features intended to make the language more flowing and English-like, and to more closely resemble a dramatic play.

Outline. The remainder of this document is organized as follows. Sect. 2 contains an introductory exposition of THAPL, including a discussion of the basic concepts and some notes on syntax. Examples of prior research and similar implementations are presented in Sect. 3. Sect. 4 enumerates some challenges that may be encountered as part of the work being proposed. Finally, Sect. 5 concludes.

2 This Proposal

2.1 Thapl language principals

Inspiration As an inspiration for THAPL, let us look to our primary source, theaterical plays. In Fig. 2, we see a portion of such a play. Most of it is dialog, which is not very useful for describing slide-shows, but note the following:

1. Characters entering.
2. Characters acting (climbing the wall).
3. The description of the scenery.
4. The general shape of the script - indentation and so on.

Fig. 2 Act 2, Scene 1 from William Shakespeare's "Romeo & Juliet"

```
ACT II. Scene I.
A lane by the wall of Capulet's orchard.

Enter Romeo alone.


  Rom. Can I go forward when my heart is here?
    Turn back, dull earth, and find thy centre out.
                [Climbs the wall and leaps down within it.]

                     Enter Benvolio with Mercutio.


  Ben. Romeo! my cousin Romeo! Romeo!

  Mer. He is wise,
    And, on my life, hath stol'n him home to bed.

  Ben. He ran this way, and leapt this orchard wall.
    Call, good Mercutio.

  [ ... ]

  Ben. Come, he hath hid himself among these trees
    To be consorted with the humorous night.
    Blind is his love and best befits the dark.

  Mer. If love be blind, love cannot hit the mark.
    Now will he sit under a medlar tree
    And wish his mistress were that kind of fruit
    As maids call medlars when they laugh alone.
    O, Romeo, that she were, O that she were
    An open et cetera, thou a pop'rin pear!
    Romeo, good night. I'll to my truckle-bed;
    This field-bed is too cold for me to sleep.
    Come, shall we go?

  Ben. Go then, for 'tis in vain
    'To seek him here that means not to be found.
                                                  Exeunt.
```

Basic concepts

Actors are those things which appear on the screen. At its rawest, the language is concerned with manipulating actors over time. Actors have names and are addressed using that name.

Groups Actors belong to groups, which can be hierarchically organized. Groups are also named and addressed by name. Any operation that is allowed on a single actor is also allowed for a group, and is equivalent to applying the operation to all the members of a group. Open question: can an actor belong to more than one group?

Atomic actions Actions are operations to be performed, and execute pre-defined behavior. The actual actions are not part of the basic language, and may depend on external libraries like Ti*k*Z. They are defined by a configuration file and can be customized by the user.

Atomic actions can be categorized into the following types:

1. Actions with no receiver actions: "`dim the lights`", "`relax`" (insert empty slide.)
2. Nullary actions, which have receiver actions but no arguments: "`show`", "`hide`", "`exit`", "`enter`", etc.
3. Move actions, which act with respect to predefined locations.
4. Scalar Unary set actions: e.g., "`color: red`"
5. Scalar Unary actions which act on a stack: e.g., "`push color: red`", "`pop color`".
6. Invocation of a previously defined act (see below.)

Note that actions may or may not operate on an actor, and may accept zero or more arguments.

Atomic elaborators Actions may take elaborators, which modify the way an action isperformed. All elaborators an action may take are defined along with the action in the configuration file. Example of elaborators include "`enter slowly`", "`along curve foo`", etc.

Action constructors There are several ways to string together actions. By default, each action is separate and is executed separately and sequentially. However, it is possible to combine actions in the following ways:

Concatenation e.g., "`show a then hide a`" or "`show a + hide a`". This will execute the two base actions in sequence.

Meanwhile e.g., "`show a meanwhile hide b`" or "`show a || hide b`". This will interleave the two actions, performing both at the same time (see below for a discussion on timing.)

Repeat e.g., "`Repeat (show a then hide a) 3 times`", or "`3 * (show a + hide a)`", repeats the base action n times in sequence.

Repeat endlessly e.g., "`Repeat (···) endlessly`" or "∞`*(···)`", repeats an action forever. This can be useful in conjuction with meanwhile.

Acts are animations, described by a series of actions to be executed, with additional context to allow the executions of those actions. An act may be named, enumerated, or anonymous. Acts may also be nested within each other.

Each act is comprised of three parts:

Dramatis Personæ is a hierarchy of groups of actors.

Scenery A list of pieces of scenery. Pieces are similar to actors, but cannot be acted on by actions. However, actions can be performed in relation to them, e.g., "`move actor to piece`", after which the actor will have the same on-screen location as the static scenery piece. This part is optional and may be omitted.

Another possibility is "`move actor along curve`, which allows an actor to move in a pre-defined path across the slide.

Action A series of action statements to describe the animation.

Acts can be executed using the `perform` keyword. The first act to be executed is selected externally in the presentation file.

2.2 Time, Frames, and Pauses

Basic definitions

Frames are single slides in the resulting slideshow.

Time is real wall-clock time, measured in seconds.

An act defines an animation. Also, for each atomic and compound act we know the $\mathsf{frames}(\cdots)$ number, which is the minimal number of frames required for the act. In the resulting slideshow, we would like each act to take a certain length of time.

Translation of frames into time

Timing an act is to manually specify its length in time, either completely or partially.

Pause is a chance for the presenter to halt the presentation temporarily, perhaps to explain a particularly pertinent point. The presentation will continue only after the presenter manually resumes it (i.e., by clicking). While `pause` is very similar to `relax`, they are not identical.

Executing actions in sequence and in parallel Basic operations in an act translate to a frame. For example,

```
x show.
y hide.
```

translates to two frames added to the original stack, while,

```
x show || y hide.
```

translates to one frame added to the stack.

At this point it is important to note that we can also include "blank" or "nop" frames by using the `relax` action. So, we can combine actions like

```
(x show + relax) || (relax + y hide).
```

which will add two frames to the stack, and would in fact be identical to the first example given here.

Length in frames of actions executed in parallel In general, if A_1 and A_2 are executed meanwhile. And A_1 takes f_1 frames and A_2 takes f_2, then the number of frames of $A_1||A_2$ is the least common multiplier of f_1 and f_2

Moreover, there are actions with an unbounded number of frames, which we will denote as 0. An example is

```
Hamlet move to balcony.
```

The function $\mathsf{frames}(\cdot)$ takes as argument an action, which can be atomic or compound and returns the number of frames.

If

$$\mathsf{frames}(A_3) = 0$$

and

$$\mathsf{frames}(A_4) = 0$$

then

$$\mathsf{frames}(A_3/A_4) = 0 \tag{2.1}$$
$$\mathsf{frames}(A_3||A_4) = 0 \tag{2.2}$$
$$\mathsf{frames}(A_1/A_3) = f_1 \tag{2.3}$$
$$\mathsf{frames}(A_2||A_4) = f_2 \tag{2.4}$$

Time Time is both discrete and infinitely divisible. In other words, Thapl can always make something happen twice as fast. Time can be scaled, using elaborators. For example, the elaborator `slowly` makes an action take twice as long as it would have otherwise. Most primitive actions, like `show` and `hide`, take one time unit.

2.3 Thapl Language Syntax

Basic syntax The Thapl language is case-insensitive. It is expected that the user will use this language feature to make the code as *English-like* as possible, for example by properly capitalizing proper nouns and the first letter of every line.

Identifiers Identifiers are of the form `[A-Za-z][A-Za-z0-9-]`. Each identifier may or may not have a "s" suffix, to allow for ease of pluralization. This is useful in plural contexts, and users are expected to use this feature to make the code more readable.

Thapl allows multi-word bare identifiers, e.g., `foo bar` is a legal identifier.

Number literals There are three different number literals in Thapl:

Arabic numerals - these are the usual Arabic numerals, such as `1`, `2`, `3`.

Roman numerals - string literals that use only one of the seven roman numeral symbols (I, V, X, L, C, D and M) and can be parsed as a roman numeral, are treated as numbers.

English number names - string literals that are a proper name for a number in the English language are treated as numbers. E.g., `one`, `one hundred and twenty one`, and so on.

Roman numeral literals Roman numerals are included in Thapl so as to induce a general Shakespearean style and feel to code. The roman digits are specified in an additive notation where the digits are ordered from left to right in decreasing values. For example, `III` has the value of 3, and `XXII` has the value of 22. In order to avoid repeating the same character 4 times. If a character were to repeat 4 times, roman numerals use subtractive notation instead, e.g., `IV` means $5-1$, `IX` means $10-1$, and so on.

We artificially cap the maximum value re presentable using roman numeral literals at 3999, or `MMMCMXCIX`. This is a reasonable limitation since for large numbers roman numerals are quite unwieldy.

English named numeral literals The English named numbers are supported as numeral literals.

The basic rules for English numeral literals are as follows. For 0-20, there are atomic names: `zero`, `one`, …, `eleven`, `twelve`, …, `nineteen`, `twenty`. For 21-99, we concatenate a name for the tenth's place with a name for the unit's, unless the unit's place is zero, like so: `twenty one`, `twenty two`, …, `thirty`, …, `ninety nine`. Then from 100 on we concatenate an atomic numeral literal for 0-999 and a name for the higher decimal place (hundreds, thousands, millions) with "and" and the numeral literal for the number in the 0-99 place. E.g., `one hundred and twelve`, `twenty million nine thousand and eleven`, and so forth.

Whitespace Thapl uses whitespace and indentation in order to mark scopes, as made popular by Python. For example, when declaring actors, the actor hierarchy in denoted using differing whitespace:

```
Dramatis Personae:
  one:
   two.
   three.
```

In this example, "`one`" is an actor group which contains "`two`" and "`three`". Thapl makes heavy use of nesting.

2.4 Demonstration

As a parting note, inspect Fig. 3 for an example of more complicated animations possible with Thapl. It is very similar to Fig. 1, but shows off possibilities like moving a figure along a curve and scaling time using `meanwhile`.

3 Previous Research

When researching existing work in this topic, we found quite a few tools for making presentations & animations, but none that fit in the exact niche that we are trying to fill with Thapl. The programming languages that exist are either operating at a too low level of abstraction, are not well suited to creating animations, or are aimed at authoring some other type of media, and can create slideshows as an after-thought.

There are other tools for creating graphics and animations [20] not mentioned here since they are less useful for creating slide-shows.

It is useful to create a dichotomy between approaches that focus on creating slideshows and allow for animations, as contrasted by entries that are mainly animation engines that can also be used for slideshows.

Slideshow software & tools

PowerPoint [13] is a very impressive application for building presentations. Through the DCOM interface, we can fully control everything that is done with it [18], enabling us to create presentations and animations programmatically.

This is not a very user-friendly interface, and it is not well documented, but it does enable one to make pretty impressive presentations (see the citations, which uses R as the backend language).

Animations are limited to whatever PowerPoint can do, but are still rather impressive.

The Animation Pane [12] is of particular note as it provides a graphical user interface for creating and editing animations. One might even consider the animation capabilities provided by this as some sort of primitive programming language. It is however still limited when compared to a more free-form programming language.

LaTeX [10] is the canonical language for making presentations programmatically. A collection of packages allow us to make moving pictures–Beamer [22] lets us talk in terms of slides, Ti*k*Z [21] allows us to draw pictures, and the animate module [7] makes them move.

This is a low-level interface that we would like to abstract away.

Racket [4] is basically a Lisp [6], or more precisely a Scheme [2]. It has a slideshow module that has a very interesting approach to defining slideshows, with definitions of actors and so on that can be re-used across slides.

It also has a very rudimentary "animate" function, that allows to generate a set of slides from a function.

The downsides is that it is not very mature, and in our opinion, not very aesthetically pleasing.

Reveal.js [3] is a slideshow library written in JavaScript [5]. It can do fancy slide transitions, however it has very limited support for animations - it only supports animated SVGs.

It allows extension via plugins.

Others

Scratch [16] is intended to be an educational programming language, with which young programmers can learn to program in a visual programming paradigm by dragging "blocks" on the screen.

It also makes it very easy to draw things on the screen and act on them (rotate, morph, etc), making animations very easy.

It doesn't seem to have a slideshow standard library. However, making a slideshow ("press space to continue") is very easy [23], and there are many animations and tutorials written in Scratch that are basically slideshows.

Processing.js [15] is a programming language intended for artists in the visual art space, controlling LEDs and motors and such to create artwork. Processing.js was created as a variant of Processing [14] targeting the web using JavaScript.

Processing.js essentially provides a canvas and an API that makes it simple to draw things–however, any slideshow functionality must be added externally.

D3.js [1] D3.js is a data visualization library written in JavaScript. It can do pretty fancy visualizations and transitions, and it looks very good.

It is unable to create slideshows, focusing on figures. However, it can be embedded in any HTML/JS slideshow framework. That would prevent the use from creating inter-slide animations, though.

FOAM [8] is an MVC framework that is written in JavaScript but claims to be able to generate any language for any platform. One of its features is a slideshow module. It seems to allow very intricate animations and interactions. However, its main focus is modeling and interactively displaying data, and not slide-based presentations.

4 Challenges

During the implementation of this proposal, there are some challenges that will have to be overcome. These include:

1. Translating the look & feel of a dramatic play, and specifically a Shakespearean play, into a programming language.
2. Making the concept of time and parallel execution useful and intuitive in the context of authoring slideshows.
3. Ensuring that even reasonably complicated slideshows can be expressed using Thapl. That is, designing the language to be expressive and flexible so that it can create slideshows that were not imagined by the language designer.
4. Implementation details, such as:
   (a) Compiling to LaTeXwith Ti*k*Z, including having some integration with these technologies. However, the Thapl language itself should stay independent of the backend implementation.
   (b) Specifiying a way for an animation to be spliced in as part of an existing slideshows.
   (c) Defining a configuration file format for declaring actions and their precise meaning.

5 Conclusions

We proposed to develop Thapl, a prototype domain-specific language inspired by scripts of theatrical plays, which is intended to be used in the context of generating slide-shows in presentations. Thapl attempts to expand the declarative/imperative paradigm by introducing an actor/action model, and adds specialized constructs that describe actions occuring simultaneously. We presented the language concept, and demonstrated its application through examples.

The proposed work contributes to the current state of the art by exploring an innovative approach to the declarative/imperative paradigm of programming languages.

Future development of the concept may be in the context of a graphical medium with which to describe Thapl directives.

Fig. 3 THAPL code for a presentation on a Lilliputian war against Blefscu

```
Atomic actions:
  show.
  hide.
  color.
  blur. # dull the actor a bit so as to move it to the background.
  focus. # the opposite of blur
  play. # play a video or animated graphic
  stop.
  move.

Act MilitaristicJingoism:
  Dramatis Personae:
    planes: # some planes
      F16 one # note multi-word identifier
      F16 two
    fireworks # some sort of animated SVG of fireworks
    bombastic exclamation # text of some sort, probably decorated

  Scenery:
    # invisible curves that the planes move along, ending outside the slide
    curve a
    curve b

  Action:
    fireworks play / planes show.
    move F16 one along curve a, meanwhile move F16 two along curve b.
    # note that verb noun ordering is flexible
    stop fireworks meanwhile show bombastic exclamation.

Act Battle plan:
  Dramatis Personae:
    countries:
      Lilliput
      Blefscu

    troops:
      infantry:
        first infantry
        second infantry

      mechanized:
        tank
        tank destroyer

    enemy king

  Scenery:
    ocean

  Action:
    show countries
    pause # give historic context for endianess war
    infantry show
    pause # talk about our brave troops
    mechanized show
    pause # mention our superior technology
    hide tank destroyer # not relevant today
    move tank to Blefscu
    # 1st inf. will invade alone to lull the enemy into a false sense
    # of complacency, then the 2nd inf. will double time to join them.
    move first infantry to Blefscu meanwhile (
      relax then move second infantry to Blefscu)
    move enemy king to ocean
    # Lilliput wins
```

References

[1] M. Bostock. D3.js — data-driven documents. https://d3js.org, 2012. Accessed: 2016-10-08.

[2] K. Dickey. The Scheme programming language. *Comp. Lang.*, June 1992.

[3] H. El Hattab. Reveal.js. https://github.com/hakimel/reveal.js, 2013. Accessed: 2016-10-08.

[4] R. B. Findler and M. Flatt. Slideshow: functional presentations. *Journal of Functional Programming*, 16:583–619, July / September 2006.

[5] D. Flanagan. JAVASCRIPT*: the definitive guide*. O'Reilly Media, Inc., 6^{th} edition, 2011.

[6] P. Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[7] A. Grahn. The animate package. http://tug.ctan.org/macros/latex/contrib/animate/animate.pdf, 2011. Accessed: 2016-10-08.

[8] K. Greer, A. Vy, and J. Stone. Foam — Feature Oriented Active Modeller. https://github.com/foam-framework/foam, 2014. Accessed: 2016-10-08.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, 2^{nd} edition, 1988.

[10] L. Lamport. *Latex*. Addison-Wesley, 1994.

[11] M. Lutz. *Programming* PYTHON. O'Reilly, first edition, Oct. 1996.

[12] Microsoft Corporation. Powerpoint 2010: Change or remove an animation effect. https://goo.gl/e05Oii. Accessed: 2016-10-08.

[13] Microsoft Corporation. Powerpoint product page. https://products.office.com/en/powerpoint. Accessed: 2016-10-08.

[14] C. Reas and B. Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[15] J. Resig, B. Fry, and C. Reas. Processing.js. http://processingjs.org/, 2008. Accessed: 2016-10-08.

[16] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[17] M. Richards and C. Whitby-Strevens. BCPL*, the language and its compiler*. Cambridge University Press, 1980.

[18] A. Salam. Create amazing PowerPoint slides using R — the basics. http://asifsalam.github.io/R-and-PowerPoint-Part-1/, 2015. Accessed: 2016-10-08.

[19] W. Shakespeare, J. E. Burdick, and H. N. Hudson. *Complete works*, volume 1. Current Literature Publishing Company, 1909.

[20] F. X. Suñol Galofre. Tools for creating latex-integrated graphics and animations under gnu/linux. *The PracTeX Journal*, 2010(1):236–248, 2010.

[21] T. Tantau. The tikz and pgf packages, manual for version 3.0.0. http://mirrors.ctan.org/graphics/pgf/base/doc/pgfmanual.pdf, 2013. Accessed: 2016-10-08.

[22] T. Tantau, J. Wright, and V. Miletic. The latex beamer class. http://latex-beamer.sourceforge.net, 2003. Accessed: 2016-10-08.

[23] The Scratch Wiki. Animation Projects — the scratch wiki. https://wiki.scratch.mit.edu/w/index.php?title=Animation_Projects&oldid=152223, 2016. Accessed: 2016-10-08.

[24] D. Zongker. Chicken chicken chicken: Chicken chicken. *Annals of Improbable Research*, 12(5):16–21, 2006.