Advl: A Visualization Language for Dynamic Visualization

T. Cerrah[1], **and H.-P. Bischof**[1 2]
[1]Department of Computer Science, Rochester Institute of Technology Rochester, NY, USA
[2]Center for Computational Relativity and Gravitation, Rochester Institute of Technology Rochester, NY, USA

Abstract— *Visualization of scientific data can help to analyze and explore the data in ways, which cannot be achieved with analytical methods. Most visualization programs are typically implemented using a data flow approach. A visualization programs consist of a set of components connected via directed graph, and the data flows through the program and this process creates images, which are later assembled into a movie. We often need to change the properties of the components dynamically during the visualization process in order to create the best possible movie. We propose a method to use the visualization program as an interpreter for a dynamic visualization program, which allows making these changes without rewriting the visualization program. This method allows us to focus on a particular visual after the visualization program has been written. This method allows us to create significantly more interesting visualization movies.*

Keywords: Visualization Languages, Data Flow, Dynamic Visualizations

1. Introduction

A visualization of data can produce one image or a movie, meaning many images. This paper concerns only the type of visualizations, which generates more than one image for a data set. Examples for these kind of data sets are simulations of Black Hole mergers [2], or measurements of fracture strains[7] etc. These data sets have one common property: one value changes during the simulation or the experiment, but not necessarily in a linear fashion. In a Black Hole merger simulation this property is time, which moves forward in a linear fashion; in a fracture strain experiment/simulation it could be force, or gauge which changes cannot be described with a linear function.

Most visualization environments are using a data flow framework, which was first described by Foulser[4]. A visualization program can be modified and executed as often as needed creating individual images, which are then mounted to a movie.

In principle, a visualization program consists of components, which are connected via a directed graph. A component has n input channels and k output channels, which are connected, which create the directed graph. The same graph can also be achieved by calling methods in a particular order. A component's functionality can typically be fine-tuned using component-specific arguments. These arguments specifying individual properties, like line width, the color or transparency of a visualized object, position of the viewpoint, look-at position etc. In most visualization systems these properties cannot be modified during the execution of the visualization process, or it is very difficult to do so.

We typically create many versions of a movie from the same data set because we are not satisfied with the final result. For example, the camera movement starts to late/early and is to fast/slow, or the camera speed for two different movements needs to be identical. We decide to change when and by how much an object becomes translucent often, because the desired effect has not been achieved yet. Light positions needed to be changed dynamically because the shadow of an object hides what should be visible. Using this technique allows us to experiment with much different visualization until we find the best fitting one.

This paper describes a visualization environment where all these modifications can be made without rewriting the visualization program. This is not a new idea. The LaTeX typesetting framework follows a similar idea for typesetting text. The text is written in a document including formatting ideas like *new paragraph, this is a bulleted lists, heading, sub-heading* etc.

The paper describes a use case in detail followed by a discussion of related work, the Spiegel visualization framework[5], and *A Visualization Language* for dynamic visualizations (*advil*).

2. Dynamic Visualization

Dynamic visualizations allow changing the properties of components during the visualization process. An example will help to illustrate this. Let's assume we visualize the simulation of a black hole's merger. One purpose of this visualization is to show the trajectory of the black hole's position over time. The number of past positions, also called the length of the trajectory, must decrease over time otherwise the trajectories will be on top of each other. The left part of Figure 1 shows the simulation at the beginning stage and the right part shows the state of the simulation close to the merger. The decreasing length of the trajectory cannot be described with a linear function, because the distance between black hole's during the merger is not linear.

Another example would be to move the viewpoint from position a to position b and then to position c. The viewpoint

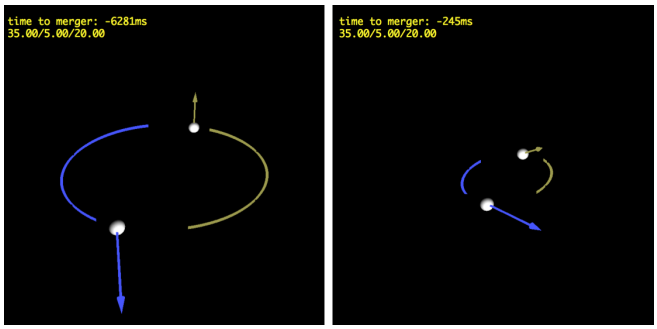

Fig. 1: Trajectory Length over Time.

position can be moved in a linear way between the anchor points, or on a spline curve. Moving the viewpoint along a linear function will cause a kink, if *a*, *b*, and *c* are not linearly aligned; moving them along a spline curve forces one to have less control over the exact movement because of the nature of splines[8]. We will later explain how these kind of problems could be addressed using Spiegel and *advl*.

3. Related Work

Many wonderful visualization systems have been developed. Most visualization systems focus on the quality of the images, being able to read a wide range of data formats, and particular visualization algorithms for very specific visualization challenges. We selected two highly used visualization systems here and a novel approach by Forbes.

The main focus of Spiegel is to create movies. Therefore it was logical to look into the production problems of cinematography[9]. We realized the world of cinematography and visualizations is so different that their approaches are not applicable for your problem.

3.1 behaviorism: a framework for dynamic data visualization

Forbes[1] created a framework which provides flexibility for visualizations of dynamic data. The framework is based on three connected graphs, and operators for each graph. The scene graph is used for rendering, a data graph is used for accessing the data, and a time graph to connect the two.

The framework provides a range of flexibility and aims to help visualization developer to focus on the visualization and not on the behavior. The paper provides little about how the behavior is controlled, it is more focused on the implementation and the design. Therefore it is very difficult to tell how it is used, but best to our understanding the behavior modifications are very limited.

3.2 yt

Yt[3] is a cross-code visualization tool that works with a number of astrophysical simulation codes, and is therefore very well suited for astrophysical visualizations. Yt is one out of four visualization systems supported by the Blue Waters Sustain Petascale Computing Center[10]. Python was the language of choice for the developer. Only the parts, which require high performance computing, have been implemented in C. Yt supports around 20 different data types, numerous algorithms to examine, and visualize the data, and MPI support for distributed visualization programs.

The code snippet in Listing 1 shows how a camera is created, rotated, and moved to a position. Lines 1-12 create the camera object. A camera rotation is shown in line 15, and a movement to a position in line 16. The first argument of *move_to* is the final position, and the second argument defines in how many steps the final position will be reached.

This example gives a glimpse of how yt is used. It is fair to say that dynamic programming is extremely difficult to achieve in yt. It can be done, but requires a rewrite of the visualization program. This is extremely time consuming and therefore not advisable.

Listing 1: Creating and moving a camera in yt.

```
1  center    = [0, 0, 0]
2  normalV   = [1, 1, 1]
3  width     = 1.0
4  xPixels   = 512
5  yPixels   = 512
6  transF = yt.ColorTransferFunction(...)
7
8  northV    = [0., 0., 1.]
9
10 cam = ds.camera(c, normalV, width,
11          (xPixels, yPixels),
12          transF, northV=northV)
13
14 theta=0.2
15 cam.rotation(theta)
16 cam.move_to([0,1,2], 10)
```

3.3 ParaView

ParaView[6] is a visualization tool supporting C++, Python and JavaScript. ParaView is one out of four supported visualization systems supported by the Blue Waters Sustain Petascale Computing Center[10]. The user guide for ParaView is 230 pages long. ParaView, differently to yt is a more general visualization tool. ParaView supports around 11 most commonly used data formats. A graphical editor can be used to create a visualization program, and it is also possible to script a visualization program.

A simple Python paraView script is shown in Listing 2 to give a glimpse of how it is used. Lines 1-4 define a sphere; Line 7 creates a renderer, which is connected with the view in line 9. Lines 11-13 shrink the sphere by a factor of 2 and are rendered in line 15.

Listing 2: ParaView Code Snippet.

```
>>> from paraview.simple import *
>>> sphereObject = Sphere ()
>>> sphereInstance.Radius = 1.0
>>> sphereInstance.Center [1] = 2.0

>>> sphereDisplay = Show(sphereInstance)
>>> view = Render ()

>>> Render(view)

>>> shrinkInstance =
        Shrink(Input=sphereInstance,
        ShrinkFactor = 2.0)
>>> shrinkDispl
>>> Render()
```

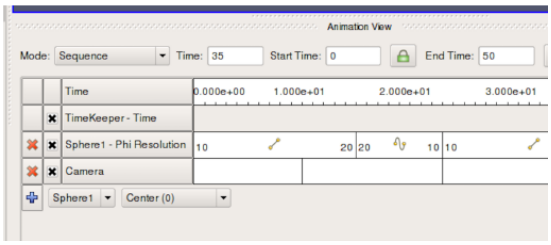

Fig. 2: ParaView Animation View [11].

ParaView supports the creation of animations using key frames. This is described in ParaView[6] guide on the pages 119-123. The key frames can be defined using the *Animation View*. The animation view is shown in Figure 2.

Only very simple animations can be created using the Animation view, like modifying a scalar, creating and modifying a camera path. Anything sophisticated cannot be done within this framework.

4. Spiegel

Spiegel[5] is a visualization framework written in Java. A program in Spiegel is a directed graph connecting individual components. An interpreter executes the program. A Spiegel program is most often implemented by using a graphical editor, but can also be implemented using a text editor. The language is type safe; this means only connections of connectors of the same type can be made. The graphical editor uses reflection[13] to ensure this property. The Spiegel language is simple, but it allows creating functions to create more complex components using simpler components or functions.

Figure 3 shows a very simple, but complete, *Hello World* program. The data flows from the component named *Stars*

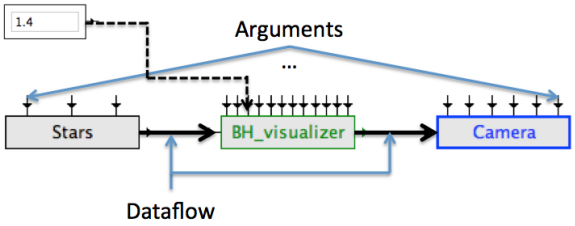

Fig. 3: Hello World.

to the *BH_visualizer* component, and finally to the *Camera*. The size of the black hole is set via an argument to be 1.4.

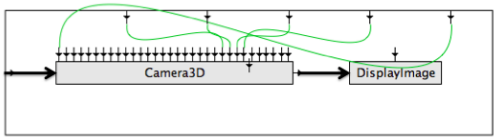

Fig. 4: A Graphical Representation of a Function inSpiegel.

Figure 4 shows a representation of the camera function used in Figure 3. As can be seen, some arguments from the components inside the function are not accessible within the *Camera* component. This encapsulating can be done with input and output channels. Encapsulation and using functions reduces the complexity of a creating a program with a graphical editor significantly.

A selection of the available Spiegel components and their categories are:

- Visuals: for visualizing Black Holes, Stars, Gas, Mesh
- Extractors: for extracting data from different data formats and origins (disk/network)
- Filters: For finding intersections, extracting positions
- Inputs: for data types like double, int, point
- Light: for point light, ambiguous light
- Util: for advl, orbiter, linear value supplier

5. Dynamic Visualization and Spiegel

Simulations or experiments, which produce the data sets, have one common property. A value changes, which drives the simulation or the experiment. In most simulations this variable is a scalar, like time, temperature, pressure, or light intensity. This value is typically used to determine which part of the data set will be used for an individual image, and which data set will be used for the next image. We will use this property to drive the programming of the dynamic visualization.

We will explain this with the help of Figure 5. The goal is to move the view point at the times 0, 3, 7, 8, and 9 to the positions outlined in Figure 5. The points are called anchor points. The positions of the view point locations in between

the known positions can be interpolated. The dotted line represents a linear and the dashed line represents a spline interpolation.

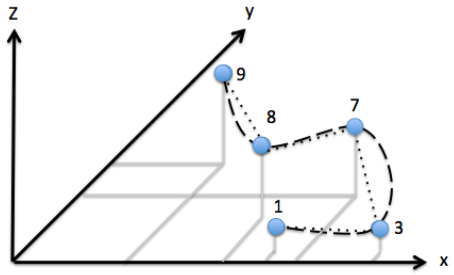

Fig. 5: Linear vs. Spline Interpolation.

6. Spiegel and Advl

Figure 6 illustrates how the advl program is used within the Spiegel framework. An interpreter component, *output Interpolator*, reads the program and provides for every stream connection. This connection, in this example *location of the camera*, is connected to one or more components. The input for the *Interpolator* is provided by the *Time* component. This component produces a series of values beginning at $t_{beginning}$ to t_{end} with intervals of t_{δ} and there fore the camera moves along the defined path.

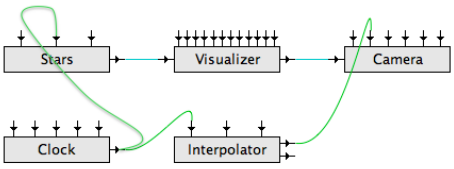

Fig. 6: Interpolating between Anchorpoints.

One component in the Spiegel framework can be programed in advl to achieve the interpolation between points and send out the values between the anchor points. Figure 6 depicts the graphical version of the program. The component *Interpreter* interprets the advl program seen in Listing 3. Lines 1-4 define two constant values. Lines 6-16 define the output stream *viewP*. Line 8 defines the type of the output stream. The type of interpolator used to calculate the position between two anchor points is a TCP spline interpolator is defined in line 9. Lines 12-16 define the output stream *simulationTime*. The interpolator for this output stream is linear as defined in line 15. Lines 18-25 define the anchor points and the position of the viewpoints.

The *Clock* component is programmed to send out values from 0 to 9 with a δ of 0.1. The *simulationTime* component output of the *Interpolator* will send out exactly the same values through the *simulationTime* output, because the line 24-25 specifies the *simulationTime* equal to the *Clock* time. This means the camera moves along on a TCB spline[12] path in 0.1 time units. The data set is accessed for the same time units. This visualization program will create 90 frames.

Listing 3: Interpolating between Anchor Points.

```
1  var {
2     const double startClock = 0;
3     const double endClock = 1;
4  }
5
6  stream {
7      viewP {
8           type vector;
9           interpolator TCB;
10   }
11
12   simulationTime {
14          type double;
15          interpolator Linear;
16 } }
17
18 startClock { viewP = (3, 1, 1);
19 }
20 3.0 { viewP = (5, 1, 1);       }
21 7.0 { viewP = (3.5, 2, 4);     }
22 8.0 { viewP = (2.5, 1.5, 3); }
23 end { viewP = (1, 1, 5);
24              simulationTime =
25              endClock;          }
```

6.1 Spiegel and Slow-Motion

We now would like to change this program to achieve a different kind of visualization. First, changing *TCBh-pbEdMSV06* to *Linear* will move camera along a linear path.

The data between simulation time, 1 and 2, might be very interesting and therefore we would like to show this part in slow motion. This means we must generate more visuals for this time period versus the other time periods. One way to achieve this is to move the clock time faster forward than the simulation time. As a result, more images will be generated and therefore a slow-motion effect will be created.

The modified advl code is shown in listing 4. We added a few constants to make code easier to modify. Line 12 was changed to move the camera on a linear path. Lines 23-26 will produce 40 clock ticks. This means instead of 10, 30 images will be generated. Lines 30-33 are needed so such

the of the visualization produces for every 0.1 time unit one image.

Listing 4: Slow Motion.

```
 1  var {
 2     const double startClock = 0;
 3     const double endClock   = 9;
 4     const double slowStart  = 1;
 5     const double slowDelta  = 1;
 6     const double clockDelta = 5;
 7  }
 8
 9  stream {
10    viewP {
11          type vector;
12          interpolator Linear;
13    }
14
15    stream simulationTime {
17          type double;
18          interpolator Linear;
19    } }
20
21 startClock { viewP = (3, 1, 1)
22           }
23 slowStart {
24      simulationTime = slowStartÕ }
25 slowStart + clockDelta {
26      simulationTime += slowDelta; }
27 3.0 { viewP = (5, 1, 1);      }
28 7.0 { viewP = (3.5, 2, 4);    }
29 8.0 { viewP = (2.5, 1.5, 3); }
30 end + clockDelta {
31      viewP = (1, 1, 5)
32      simulationTime = endClock +
33            slowDelta;         }
```

7. Advanced Advl Program

A more complicated example is show in Lisiting 5. Lines 1-8 define variables; lines 9-14 create the camera position stream. A function, *moveCam* , is defined in 15-21. The anchor point, line 23-27, defines the variables *x* and *y*. The scope of these variables is this block. The lines 23-34 move the camera position to a given point and back. The value of the built in variable *time* is equal to *deltaTime* after line 27 has been interpreted. It is worth to point out that the speed of camera is identical for both movements. A modification of *deltaTime* would change the speed for the camera movements for both segments. Lines 35-39 moves the camera in the time along a varying x value.

Listing 5: Advl and Spiegel in Concert.

```
 1  var {
 2    const double deltaTime = 42;
 3    const double z = 2;
 4    const double radiusC = 10.0;
 5    const double middleC = (1, 2, 2);
 6    double midX = 10.0;
 7    double midY = 20.0;
 8  }
 9  stream {
10    cameraPos {
11      type point;
12      interpolator TCB;
13    }
14  }
15  point moveCam(double x) {
16    double r = radiusC ^ 2;
17    double xComp = (x - midX) ^ 2;
18    double y = (sqrt(r-xComp))+midY;
19
20    return (x, y, z);
21  }
22
23  0.0 {
24    double x = 90.0;
25    double y = 20.0;
26    cameraPos = (x, y, z);
27  }
28
29  0 + deltaTime {
30    cameraPos = (x + delta , y+delta , z)
31  }
32  time + deltaTime {
33    cameraPos = (x  y, z)
34  }
35  time + deltaTime {
36    for (i = 1 : 20) {
37      cameraPos = moveCam((x - 10) + i);
38    }
39  }
```

Changing of the variable *deltaTime* (line 2) would change the speed of the camera movement, but not the path of the camera movment.

8. Conclusion

Advl is a language, which allows controlling the behavior of visualization systems effortlessly. It would be relatively easy to add this framework to yt, or ParaView, which would allow developers to use and control very sophisticated visualization with the same language. Using small, domain specific languages allows for an ease of use which can not be achieved general purpose languages.

9. Future Work

Future work will include to add the functionality to yt, and ParaView. Spiegel and advl do not support much user interaction during the execution of the visualization program. Domain specific programming languages drive the complete visualization process. It might be useful to allow user interaction during the visualization process to change the visualization process if *interesting* things can be seen. It might be useful to add an AI component, which can direct the visualization process to direct the visualization process instead of advl.

10. Acknowledgements

The authors would like to thank all members of *The Center for Computational Relativity and Gravitation at RIT.* Their visualization needs drove much of the development of advl.

11. Appendix: Advl

This section describes the syntax of *advl* .

Listing 6: advl Syntax.

```
prog: vars? streams func* anchor+

vars: 'var' '{' varDecl* '}'

streams: 'stream' '{' stream+ '}'

varDecl: 'const'? basic ID
         ( '=' expr )? ';'

stream: ID '{' 'type' basic ';'
        ('interpolator' interp ';')? '}'

func: type ID '(' params? ')' block

block: '{' stmt* ('return' expr ';')? '}'

params: basic ID (',' basic ID)*

anchor: DOUBLE ('+' ID)? block

stmt: ID ('='|'+='|'-=') expr ';'
    | ID ('++'|'--') ';'
    | ID '(' args? ')' ';'
    | varDecl
    | ifBlock elseIfBlock* elseBlock?
    | 'while' '(' expr ')' block
    | 'for' '(' ID '=' expr ':' expr ')'
      block

ifBlock: 'if' '(' expr ')' block

elseIfBlock: 'else if' '(' expr ')' block

elseBlock: 'else' block

expr: ID '(' args? ')'
    | expr '==' expr
    | expr '!=' expr
    | expr '<=' expr
    | expr '>=' expr
    | expr '>'  expr
    | expr '<'  expr
    | expr '&&' expr
    | expr '||' expr
    | expr '*' expr
    | expr '/' expr
    | expr '+' expr
    | expr '-' expr
    | expr '%' expr
    | expr '^' expr
    | '-' expr
    | 'sin' expr
    | 'cos' expr
    | 'tan' expr
    | 'sqrt' expr
    | 'abs' expr
    | '(' expr ')'
    | bool
    | INT
    | DOUBLE
    | point
    | bool

args: expr (',' expr)*

interp: 'Linear' | 'TCB'
type: 'int' | 'double' |
      'point' | 'bool' | 'void'
basic: 'int' | 'double' |
       'point' | 'bool'
point: '(' doubleValue ','
        doubleValue ',' doubleValue ')'
bool: 'True' | 'False'

doubleValue: DOUBLE | id

ID: ID_LETTER (ID_LETTER | DIGIT)*
INT: '-'? ('0'| NZD DIGIT*)
DOUBLE: '-'? ('0' | NZD DIGIT*)?
        DOT DIGIT*

ID_LETTER: 'a'..'z'|'A'..'Z'
DIGIT: '0'..'9'
NZD: '1'..'9'
```

```
COMMA: ','
DOT: '.'

LINE_COMMENT: '//' .*? '\n'
COMMENT: '/*' .*? '*/'
WS: [ \t\r\n]+
```

References

[1] A., Forbes, T., Hoellerer, and G. Legrady, " behaviorism: a framework for dynamic data visualization," *IEEE Transactions on Visualization and Computer Graphics* , vol. 16, issue 6, October, 2010, pages 1164-1171.

[2] C., Lousto, and J. Healy, " Flip-Flopping Binary Black Holes ," *Phys. Rev. Lett.* , vol. 114, April, 2015.

[3] M. J. Turk, et all., " yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data ," *arXiv*, vol. 192, Jan. 2011.

[4] D., Foulser, "IRIS Explorer: a framework for investigation," *ACM SIGGRAPH Computer Graphics - Special focus: modular visualization environments (MVEs)*, vol. 29, Issue 2, pp. 13-16, Nov. 1995.

[5] H.-P. Bischof, E. Dale, and T. Peterson, "Spiegel - a visualization framework for large and small scale systems," in *Proc. MSV'06*, 2006, paper, p. 199-205.

[6] C. Quammen, "Scientific Data Analysis and Visualization with Python, VTK, and ParaView," in *Proc. Python Conference'15*, 2015, paper, p. 32-39.

[7] G. Yuang, B. Yan, and H. Zhu, "Measurement of Fracture Strains for Advanced High Strength Steels (AHSS) Using Digital Image Correlation ," in *Proc. SAE'09*, 2009, paper, p. 482-486.

[8] Donald House.(April/2016). Chapter 14: Spline Curves. [Online]. Available: https://people.cs.clemson.edu/~dhouse/courses/405/notes/splines.pdf

[9] H. Moore: Production Problems: Cinematography. (Journal of the University Film Producers Association, 1(1), 2Ð9. (April/2016) [Online]. Available: http://www.jstor.org/stable

[10] Blue Waters - National Center for Supercomputing Applications. University of Illinois of Urbana-Champaign. (April/2016) [Online]. Available: http://www.ncsa.illinois.edu/enabling/bluewaters

[11] Paraview. (April/2016) [Online]. Available: http://http://www.paraview.org/

[12] David Eberly. (April/2016). Kochanek-Bartels Cubic Splines (TCB Splines). [Online]. Available: http://www.geometrictools.com/Documentation/KBSplines.pdf

[13] Ira R. Forman, and Nate Forman. *"Java Reflection in Action."* Manning Publishing Company, 2004.