# Moving animation script creation from textual to visual representation

**Erik Haugvaldstad** **Tim Wright**

School of Mathematics, Statistics, and Computer Science
Victoria University of Wellington
New Zealand
{kyllingen,tim}@mcs.vuw.ac.nz

**Abstract**

Animation scripts are an integral part of developing computer games: they describe which character animations to play and when to switch between animations. These scripts are often written in text editors which can be an error-prone task since there are multiple variables and conditions that needs to be understood, and text editors gives no indication of how animations are linked together. This paper describes how a graphical user interface, using lines and circles to simulate direction and speed, can simplify the process of creating character animation scripts.

## 1 Introduction

The computer game industry is a multi billion dollar enterprise, with sales in the US alone being 7.3 billion in 2004 (ESA 2005). However, competition is stiff and many developers barely survive past their first game as costs are increasing and players are demanding more from the games they play (Williams 2002, Wikipedia 2005).

We have been working with a local game company where the developers wanted to find a better and more efficient way to create their animation scripts, which they are currently making using text editors. They felt that improvements could be made by having a tool to streamline the process.

In the game company the scripts are written so that the animators can use them to put together the animations for the game objects, but they often have to ask the programmers for help when building a script, defeating the purpose of having scripts.

Scripts are programs that evaluate user decisions and then points the main program (the game) in the right direction (Kerlow 2004). In games, animation scripts are used to glue animations together, as well as constrain which animations can be accessed so that they are played in the right order when objects move and interact in the game world. For example, if we are moving our game character left, the script might tell the game to play the `walkLeft` animation, and from that animation we can only stop, walk right or run left; but not run right. The sample script in Figure 1 shows that scripts are composed of different variables and conditions that a game engine needs to read in order to find out when and which animation to play.

These scripts are often written in a text editor, or in software with little to no visual aid. Currently, the problem of having to write these by hand in a text editor are twofold. First, having to write similar code over several pages where only a few names or variables change is tedious. It can lead to errors as users can have trouble determining exactly which parts to change and predicting how the changes will impact the rest of the script. Second, understanding the script's structure is hard: determining what animations exist and how to combine them. Visual aids, like charts or graphs, detailing how the animations work together can be helpful, but it assumes that someone had the time to write this down, and in a way that is understandable for everyone. Graphs and charts are of limited use as well since they will not necessarily explain the layout of the script nor what every variable and condition means.

```
Node {
AnimName = Run
AnimFile = RunAnim.ani
MinSpeed = 30 MaxSpeed = 50
}

Node {
AnimName = Walk
AnimFile = WalkAnim.ani
MinSpeed = 10 MaxSpeed = 30
}

Transition {
From = Run    To = Walk
When =  Speed < Run.MinSpeed AND
        Direction.Angle > 330 AND
        Direction.Angle < 30
}

Transition {
From= Walk    To = Run
When = Speed > Walk.MaxSpeed AND
        Direction.Angle > 330 AND
        Direction.Angle < 30
}

... plus 100's of other similar rules
```

Figure 1: Sample script showing animations nodes and transition nodes

In this paper we try to describe how a graphical interface simulating movement and direction using a 2D plane can make it easier to generate different animation scripts within the domain of character animations, and at the same time make the process more understandable.

## 2 Cognitive Dimension Analysis of Script Notation

The cognitive dimensions framework (Green 2000), uses a set of dimensions that lets designers conduct usability evaluations of visual programming environments.

Although the framework was designed to deal with visual notations, the notations can still be applied to a textual environment. We are using the notation to do an analysis

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at the Seventh Australasian User Interface Conference (AUIC2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 50. Wayne Piekarski, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

of the scripts when they are edited in a text editor. The scripts can be thought of as a programming notation for a particular domain, in this case, character animations. We assume that a general text editor is used, like notepad or emacs, as there is no default editor for scripts. The Cognitive Dimensions analysis follows:

A text editor can have partial high **viscosity**. Replace commands can make simple tasks, like substituting all instances of a keyword for another in the file, but more complicated replacements needs regular expressions. Depending on the complexity, this can be a challenge and having to create regular expressions can both be time consuming and difficult to learn. It also introduces a higher level of abstraction.

**Visibility** in a text editor tends to be poor, even with colour coding of keywords, as its contents is still represented as paragraphs of text going from the top to the bottom of the page. It is therefore hard to find exactly the parts we want, and even harder to find which ones are dependent on each other. For example, deleting animation node `Walk` in Figure 1. would introduce errors as one of the transition nodes is expecting to use a value from the 'Walk' node.

A text editor has a **hidden dependency** problem with the scripts. For example, an animation allows a character to move left or right. The animations would change depending on the angle of our character, so that if we turned more than 40 degrees to the left or right the animation would change. These angle values have to correspond with each other so that the correct animation will be played. If there is an inconsistency between these values the text editor will not find it and problems may arise when the game engine tries to read the script.

It is quite easy to make **errors** in a text editor, as it does not tell us if we have written something wrong semantically, nor if we have followed the structure set by the script. We could delete content or change values and we would not know if it affected anything negatively until the game engine tries to read the script.

The text editor does not put any constraints when writing the scripts, which means it does not introduce **premature commitments** — we do not have to write a part of the script before any other parts and it is easy to change anything we have written before, although we might have to worry about hidden dependencies.

There are several helper tools in a text editor that help achieve **abstraction**. Examples include regular expressions or the replace command. These fit within the context of manipulating words or sentences, but they fail to make the scripts more understandable.

Although a text editor will get the job done, and there are tools in the editor that can simplify some of the script creation, there are many problems which makes it cumbersome. Green (1996) argue that something that is hard in one environment may be easier in another. We want to move this script creation to software that will reduce the problems with text, while keeping the power of the notation.

## 3 Using spacial arrangement of nodes to create animation scripts

Character animations often have a direction and a speed, for example, a person walking to the left or running forwards. Using a 2D plane (see Figure 2), we can use a pattern of lines going from the centre and outwards to divide areas into directions, and circles to further divide these areas into different levels of actions in a given direction. In other words we are using a vector from origin to a point to define the speed and direction of an object.

Each of these regions consist of an animation and its corresponding variables and conditions. The variables are used to store facts about an animation that a game engine needs to know in order to process the files like the name, file path for the animation, or values to be used in a transition condition specifying when a change between two animations is going to happen. Examining Figure 1, we see that each animation node has a minimum and maximum speed that is used in the transition to determine when the animation should change between, say, walking and running. Variables consist of a variable name and its value. Since the scripts are in plain text they are easy to add without having to think what type a variable is. For example, whether an array of numbers are represented as `values = (2, 3, 4)` or `values = Array[2, 3, 4]` should not matter for the tool. We want to ensure users have an easier time putting the right variables to the right animations.

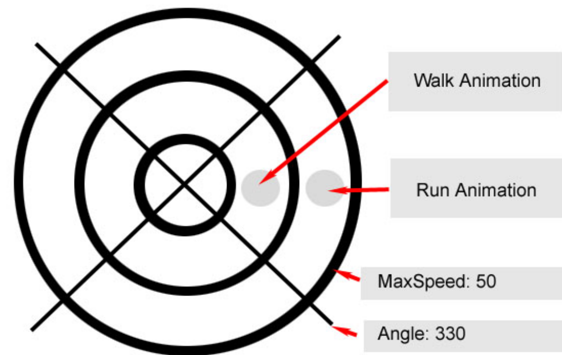


Figure 2: Using lines and circles to create several areas that contains animations and code. Lines and circles can store values as well.

Through the animations and the variables the tool has enough information to create scripts. The tool can create the transitions by changing animations to neighbouring or even distant areas. Animation transitions could be used between levels, or just in a given direction. Empty areas may or may not be allowed, and there could even be a complicated mesh of transitions between the nodes. This means we can create a multitude of different scripts just by changing a few rules regarding how animations are connected together. Since the 2D plane allows for so many different ways of specifying a change between animations, it will be up to the developers to state how they want the transitions to function.

To make these transitions there needs to be one or several conditions that tells us when a change can happen. Animations often change when we change the direction of our game characters. The lines can therefore be used to set the constraints for when a change happens or which animations are to be played in a given direction. The most obvious way would be to store the angles of the lines and use those values to specify the different directions that can be achieved. Of course, other (non-numeric) values could be used as well, like `left` or `right`, if that is what the game engine needs to read.

The circle's radius value could also be used as a variable and constraint for when we need to go a step up or down in a given direction (changing speed). The variable could be represented as a minimum and maximum value by retrieving the radius value from the two circles closest to the animation node. For example, when an animation should change from running to walking we could look at the value of the circle between these two nodes and decide if we should change an animation based on the speed we currently have in the game.

The other variables for each animation can also play an integral part, as these often will give additional constraints for when a transition between animations can happen. We therefore need a way of specifying how options affect the conditions. As mentioned before, since condi-

tions can have such diverse structures depending on the script, the best approach would be to use a selection model that tells whether an option should be used in a condition.

Although the layout looks only suitable for scripts that deal with changing directions and moving at different speeds, it can be easily modified to fit a large variety of other animations, as the structure of these scripts are often quite similar. For example, the circles could be different modes of firing a weapon in a different direction, or we could use it for a car to specify which animations to play when a car turns or changes gears.

A tool implementing these ideas should achieve a higher degree of visibility and a lower level of viscosity and error-proneness since it is easier to see how each part of the script connects to each other, and many errors can be avoided. The cost comes as a higher level of abstraction, although we believe that the abstractions introduced will not add to the complexity of the task — the layout should make it easier for users to see how the animations are tied together and where each animation node is supposed to be.

## 4 The Animation Scripting Utility Tool

The Animation Scripting Utility (ASU) tool is a prototype being developed for a local computer game company in New Zealand. The animators there have been creating their animation scripts by hand in text editors and were looking for some kind of software that could simplify the process of creating these scripts. This tool takes advantage of many of the key points discussed in the previous section to help with that process. It is still more constrained than what has been described in the previous sections, as many of the ideas were not implemented due to time constraints. The tool will most likely be redesigned and include additional content by them to better fit their own standards after they have reviewed it.

The tool draws upon programming by demonstration principles (PBD) (Halbert 1984, Myers 1986, Wolber 1997) as we are trying to use an interactive graphical interface to construct code. However, the animations in our context have already been created by professional 3D animators. The task we are trying to solve is not creating the animations, but, rather, providing an efficient and easy mechanism to sequence different animations depending on game variables like character speed and movement direction.

The tool was created using Java and Java2D for graphics. It uses a graphical plane where circles and lines can be added and manipulated by a user to specify the number of directions and levels that should be added. The lines store an angle that can be displayed as degrees using either [0-360] or [(-180)-180]. This angle can then be used as a constraint or variable in the scripts. The circles store the radius value which can be used as a variable in animation nodes or used in a transition condition. A scale value can also be applied so that very small values can used for the circle's radius whilst still being able to see and manipulate the circles on the graphical plane. Changing values is as easy as selecting a circle or line and moving it to the desired position. For more fine-grained control, the value can also be supplied in a text field.

The areas between the lines and circles are used to store animations files and its variables. When a user clicks on any area, they are presented with a table that shows the animations currently in that area. To help distinguish between areas that contain animations and those which do not, a small circle is drawn to show which areas contain an animation. These animations store the name, debug info and the animation file name. Animation file names can be written in the text field or they can be supplied by browsing through a file menu. Whenever an animation is selected, another table is presented where variables can be added or changed to the selected animation node. There is

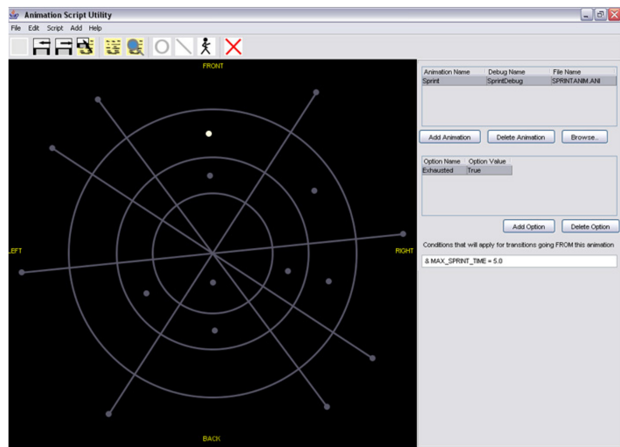


Figure 3: Main screen of the ASU tool

also the possibility of adding global options that will apply for all animations, saving time having to add the same variable to every area. Global options which can then be overridden at each area.

After having added all necessary animations and options, ASU can create an animation script by first extracting the animation details and its variables and adding this to the script as an animation node. Next, ASU adds the transition details between neighbours. To fit with the game company's scripting specifications, neighbours are defined as a non-empty area either a level above or below an animation in a given direction.

We allow areas to be empty, in which case a neighbour will be the next closest area that contains an animation. Again, this is to fit with the scripting requirements of the game company, but this is likely true for many other animations as well. Since we always have an equal number of areas in all directions, we might end up with surplus areas that can not be used. The only animation node that does not follow these constraints, is the root node.

The root node is the centre of the circles and lines and is not represented on the plane as its own area, as it is not an animation, but rather a common centre that all animations can go to and from and is a point of entry for reading a script. Being a special case, the root node is added under an option screen where its variables can be specified, and we can also choose whether a root node should be included at all, as not all scripts will necessarily have one. The root panel could also include an idle animation to be played if the game character is just waiting, but the idle animation is not included by default as there may be many different idle animations to play, which means a script for these transitions should be instead created individually first, and then combined with other scripts.

The script, when produced, can be displayed in a text window for further analysis or it can be saved to a file which can then be used with the game engine. Alternatively, the contents on the drawing plane can be saved to a file as well for further processing at a later stage.

## 5 Usability Analysis

We performed a usability evaluation of ASU using four university students. The analysis consisted of three phases. In the first phase, the evaluators used the tool freely; trying different functions, reading through the user manual, and asking questions. In the last two phases participants created a complete animation script by following a set of instructions. The instructions to create the script in the second phase were more detailed than in the third phase.

The positive the participants quickly found a good

work pattern and learned how the tool worked and how the animation nodes were linked together on the graphical plane. None of the evaluators had any animation scripting experience which shows that the interface has been helpful — it allowed the evaluators to create simple scripts without having to know the details and structure of the finished script file.

The evaluation found several issues that were resolved: most of these were minor bugs. The two main issues the evaluators had were moving between selecting values from the animation table and the graphical plane, and old values not automatically being overwritten. This was fixed by setting the focus on the animation table cells and making sure that values would be overwritten instead of appended. A third issue was the lack of a copy function for areas or not being able to select several animations at once for quick editing. This would take longer to implement and was left for future work.

## 6 Future Work

There are several features we would like to implement for the ASU tool which were not implemented due to time constraints. Since the process of adding new animations can be cumbersome we would like to have a copy function that makes it easy to copy contents from one area to the other so that similar animations can be constructed more quickly. Preferably a drag and drop function that allows us to copy an area's contents to another.

Two important features that we would like to see implemented are support for multiple animations and extending the graphics plane for 3D support. This will generalise our tool enough to allow for more complicated character animation scripts, as well as other domains of animation scripts that previously would not fit with our model. By allowing more than one animation per area, a different animation can be selected during a transition based on either defined constraints, or even chosen randomly. For example, whenever we want our game character to kick, it might start as a standard kick animation and then depending on where we want him to kick the animation will play two different animations. The first animation might just show our character getting the leg off the ground; but depending on whether we want to kick high or low it will play a different animation.

## 7 Conclusion

Creating animation scripts can be a difficult and time consuming task. The CD usability analysis shows that text editors are not the best tools for creating animation scripts, which means we have to have alternate software that can handle the task more effectively (Green 2000). Moving difficult and repetitive tasks to a more graphic oriented interface needs to take several things into consideration. The software needs to make sure the job is simplified, and that the interface is understandable for the users as well as providing the level of detail that is needed for a script.

Focusing on the domain of character animations, we have come up with a general scheme of how to create these scripts. We built and evaluated a tool using these ideas. Creating animation scripts by simulating a region of directions and movements will make the task easier, even for people who have little experience with scripting, since the environment should feel more natural to them and it gives a better overview of the relationships between animations. Variables and constraints can be added individually to an animation or as a global value that affects all animations. The sample scripts we had showed that over half the content was transition data, meaning, on average, users need to do half as much work when using ASU.

## References


ESA (2005), 'Esa top 10 industry facts', http://www.theesa.com/facts/top_10_facts.php.

Green, T. (1996), An Introduction to the Cognitive Dimensions Framework, *in* 'MIRA Workshop', MIRA, Monselice, Italy.

Green, T. (2000), Instructions and Descriptions: some cognitive aspects of programming and similar activities, *in* 'AVI Proceedings', AVI.

Halbert, D. (1984), Programming by Example, PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkely, California.

Kerlow, I. (2004), *The Art of 3D Computer Animations and Effects*, third edn, Wiley.

Myers, B. A. (1986), 'Visual Programming, Programming by Example and Program Visualization: A Taxonomy', *CHI Proceedings* pp. 59–66.

Wikipedia (2005), 'Video game developers', http://en.wikipedia.org/wiki/Video_game_developer.

Williams, D. (2002), 'Structure and Competition in the U.S. Home Video Game Industry', *The International Journal on Media Managements* **4**(1), 41–54.

Wolber, D. (1997), 'Pavlov: An Interface Builder for Designing Animated Interfaces', *Transactions on Computer-Human Interaction* **4**(4), 347–386.