CS 4115 - SPINNER White Paper

William Beaver[1]
wmb2013@columbia.edu

Marc Eaddy
eaddy@cs.columbia.edu

Gaurav Khandpur
gaurav_k@cs.columbia.edu

Sangho Shin
ss2020@cs.columbia.edu

September 23, 2003

[1]team leader

Chapter 1

Introduction

1.1 Background

SPINNER is a general motion control language designed to control SPIN: a performance art installation. Slowly revolving orange wheels are projected onto a hanging grid of translucent Plexiglas circles. The wheels act individually and in groups, joining forces and breaking away in an evolving dance of whole and parts, synchrony and disorder. In SPIN, sounds are produced as if each disc were a phonograph record. As each disc spins fast or slow, backwards or forwards, speeding up or slowing down, its sound follows accordingly. SPIN was created by sound designer and multimedia artist Ben Rubin and presented at the Bumbershoot Visual Arts Exhibition in Seattle, Washington from August 26th to September 1st, 2003.

SPIN is built in the MAX/MSP programming environment; a graphical environment for music, audio, and multimedia. The basic environment that includes MIDI, control, user interface, and timing objects is called Max. Built on top of Max are MSP, a set of audio processing objects that do everything from interactive filter design to hard disk recording, and Jitter, a set of matrix data processing objects optimized for video and 3-D graphics.

To originally control and coordinate the movement of each of SPIN's six discs, a simple, asynchronous, linear scripting language - SPINscript - was created. This language is comprised of a number of initialization parameters and three simple primitive functions: *wait*, *line* and *speed*. The initialization parameters setup audio (gain, assign sound samples to each disc, etc) and define the physics of the system (for example, disc acceleration and deceleration). The primitives are used in concert to control and coordinate the motion of each disc. *Line* tells a particular disc to rotate to a particular angle within a specific timeframe. *Speed* changes the linear acceleration of a disc. By using the *wait* command, a programmer can set discs in motion and then wait a specific time period before issuing the next set of com-

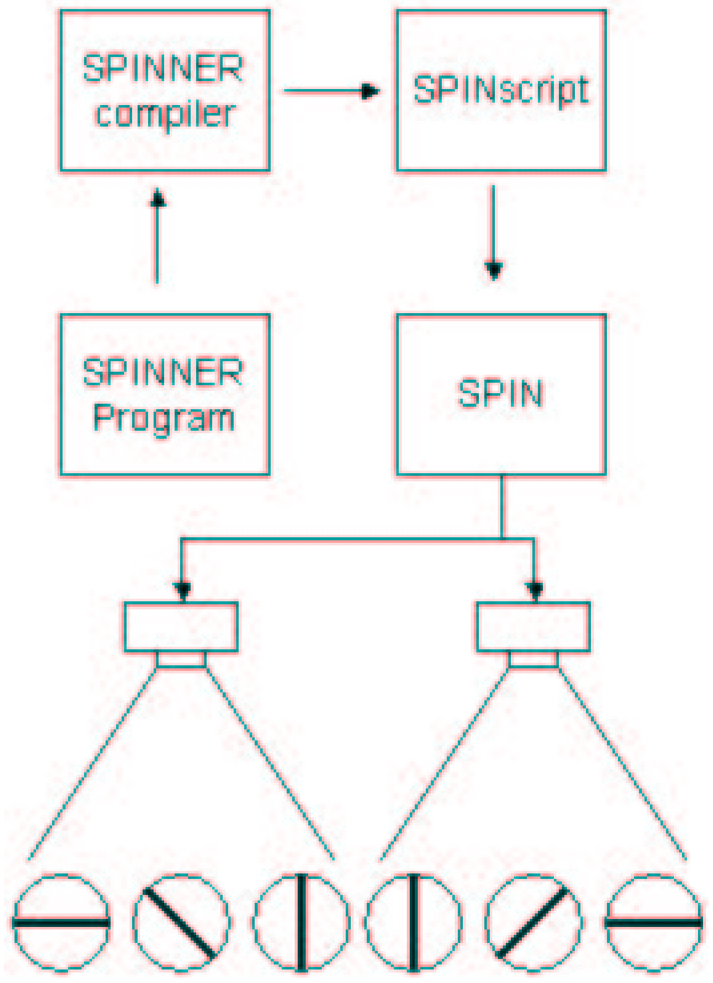

Figure 1.1: SPIN component diagram. SPINNER program read by SPINNER compiler. SPINNER compiler outputs SPINscript which is read by SPIN to control projection of discs.

mands; allowing the discs to move in space according to the direction of the line and speed commands. SPINscript is asynchronous. Without the *wait* command, a script attempting to coordinate disc movement by issuing a series of specific motions would in actuality set a disc into one motion which is exactly the additive result of all the commands issued in the script for a disc.

SPINNER is a new language for SPIN. SPINNER is intended to replace SPINscript as the primary interface for expressiveness in SPIN and to solve many of the problems that are inherent in the existing language (see below). SPINNER will generate a file that is compatible with the existing SPINscript language and is subsequently used to control SPIN.

1.2 The Problem

The original SPINscript language has a number of deficiencies that inevitably limit the expressiveness of individual disc motion and the potential complexity of disc interaction.

1.2.1 Primitive programming model

SPINscript has a strictly sequential control flow. It does not support looping, function calls or jumps. If the user wants to gradually increase the speed of a disc or play the same sequence several times they have to laboriously duplicate the scripting code by hand.

Another problem is that SPINscript does not support variables or constants. This leads to a proliferation of "magic numbers" throughout the script whose intent is impossible to discern. If the artist wants to increase the speed of an entire sequence by a factor of two, she has to calculate and modify each number by hand. This is one area which makes the language very brittle.

1.2.2 Syntax is non-intuitive

SPINscript lines follow the following format:

```
<number>, [disc-]<cmd> <args>;
<number>, --wait-- <ms>;
<number>, --[comment];
```

Here are some examples:

```
1, 6-speed .50 10000;
1, --wait-- 14000;
1, ---------------straight up now;
```

There are several problems with this syntax:

1. The first value *(1,)* is unnecessary. The script language designer has indicated that the <number>, portion isn't used.
2. The disc number that a command operates on is prefixed to the command's name (eg. 6-speed). This syntax is limiting; it does not allow actions on multiple discs, and it is unintuitive.
3. Arguments to functions are not bound to the function name.
4. The wait command does not have the same syntax as other SPINscript commands. It almost looks like a comment.
5. Comments implicitly indicated by the lack of a keyword after "- -" rather than explicitly stated through their own syntax.

1.2.3 Modeling limitations

SPINscript models changing the angles of six independent discs over time. However, this only models part of what the artist is trying to achieve. In many of the "scenes" the artist requires that the behavior of the discs be dependent on each other. For example, a scene might start out with all discs rotating at different speeds. This goes on for several seconds. Then when two discs happen to both be at 0 degrees rotation, the artist wants both discs to synchronize their rotation speed.

Unfortunately, SPINscript does not have a built-in mechanism to perform this synchronization. The artist has to do this calculation manually which is tedious and prone to error. Because of the manual labor involved, scripts must be kept small and simple. This ultimately limits the artist's expressiveness.

1.3 Our Goals

1.3.1 Improved programming model

We will support various methods for control flow; loops, iteration, non-recursive function calls. We will support integer and floating point variables. Additionally, SPINNER provides some built-in functions and operations and the ability to define new functions and procedures using these primitives. For example, we are considering supporting sine, cosine, and tangent functions in addition to the basic math operations of addition, subtraction, multiplication, and division. Here is an example[1]:

```
; Start all the discs spinning at different speeds based on sine function
for i := 1 to 6
   revolve(i 1 sine(i * 1000));
```

In addition to the for-loop, function call, and variable use, this example also shows how the disc number will become an argument to the function instead of being part of the function name.

1.3.2 Intuitive language syntax

The programmer must be able to transform concepts of the motion of images in space into the language easily. SPINNER supports disc referencing in a consistent manner. Inline comments are handled in a manner more consistent with existing programming languges like C and Java. Use of procedural language constructs provide a natural mechanism for control flow and reference.

1.3.3 Natural high-level semantics

For SPINNER to be successful, it must closely model the problem domain. The programmer must be able to transform concepts of the motion of images in space into the language easily. We will provide a mechanism for synchronizing discs based on different conditions, queriying disc state (angle, speed), etc. For the first time the artist will be able to use the language to define behavior dependencies between discs.

For example, let us say that the artist wants to start two discs rotating at different speeds:

```
1, --Start Disc 1 rotating at 2 revolutions per second--;
1, 1-line 2 1000;
1, --Start Disc 2 rotating at 1 revolution per second--;
1, 2-line 1 1000;
```

The artist wants them to rotate like this for 2 seconds:

```
1, --Wait 2 seconds--;
1, --wait-- 2000;
```

[1]The final syntax may be different. We are working closely with the end-user to determine what functions and operations would be most useful

Now she wants synchronize the speed of Disc 2 with Disc 1 *when both are at 0 degrees*. How does she do this? Previously, she would need to calculate the exact time this condition would occur. In this example, it would take an additional 1 second before both discs are at 0 degrees. She would then create a wait statement for this amount of time:

```
1, --Wait until both discs are at 0 degrees--;
1, --wait-- 1000;
1, --Now make the speeds the same--;
1, 2-line 2 1000;
```

The synchronization problems centers around the $1, --wait--1000;$ statement. Currently this requires the artist to know the current speeds and starting times of each disc in order to determine their angles at any point during the scene. Our goal is to have our compiler determine this number and allow the artist to specify a conditional that closely models the problem domain.

Here's a pseudocode example[2] of a synchronization statement that the artist could specify in SPINNER:

```
// Wait until both discs are at 0 degrees
WaitUntil (disc 1 position = 0) =(disc 2 position = 0);
// Now make disc 2 speed equal to disc 1
disc 2 speed == disc 1 speed
```

In order to support a WaitUntil-like command, it is necessary for our compiler to keep track of the speeds of each of the discs *as it is outputting the SPINscript*. We may also have to take into account the time it takes the scripting commands themselves to execute. This is analogous to the amount of clock cycles needed to execute an assembly language instruction.

[2]We are still designing the synchronization command syntax to make sure it is expressive and flexible. Because of this the syntax will likely change.