Sketch-Based Animation Language

Eron Steger

Abstract

We present a visual language for the creation of simple 2D animations within a sketch-based interface. Motions are represented by motion paths, which are drawn as arrows. Events are used to synchronize disparate motions, marking where along motion paths multiple entities should be at the same time. Reuse of animations is supported with the notion of animation functions, which allow previously specified motions to be played back.

1 Introduction

The process of constructing an animation generally requires many steps going from design to completion. While there has been much work in tools for implementing animations [1], less work has gone into the important preliminary stages.

When planning an animation, it is useful to start with a simple 2D drawing of the scene. This sketch, through its images and annotations, describes how various objects move in relation to one another. These annotations can come in a variety of forms, such as lines to describe where an object will go, to text describing what an object should be doing. This design process is fluid, allowing the artist to quickly construct new animations on the fly.

We present a system for prototyping animations using a sketch-based interface. Given an annotated sketch as input, the system will generate an animation. The make-up of the sketch's images and annotations can be considered to be a form of visual language, in that they describe how the animation should look. The definition of this language is important, as it must be both easy to understand for the user as well as recognizable by a computer.

In this work, we will concentrate on high-level motions within a scene. That is, instead of dealing with the specifics of how a character walks, we concentrate more on the over all movement of multiple characters and how they interact. While the language doesn't concentrate on lower-level motions, it is flexible enough to deal with them.

2 Previous Work

Sketch based systems are a natural choice for the creation of graphical media, with much work focused on modeling. For example, the "Teddy" system allows for the creation of models by interpreting sketches and gestures from the user [3].

There has been some work on interpreting 3D pose from sketches. In [2], a system is presented that takes as input a set of 2D key frame sketches and with some additional user interaction, outputs a 3D animation.

Of particular interest for the sketching of animation are Motion Doodles ([5] and [6]). Motion Doodles specify animations by sketching a gesture of the desired motion. Such a system could complement ours, providing additional meaning to the shape of motion paths drawn in our system.

3 System

Our system for generating animations from sketches can be thought of in similar terms to that of a compiler:

a) Tokenizer: Identify lines, arrows, and other marks in the sketch.
b) Parser: Determine the structure of the sketch. Motion paths are connected to each other, and annotations such as events are located in reference to their location on the sketch.
c) Animation generator: Generates the animation clips from the parsed data and synchronizes them together.

Sketch ➔ Tokenize ➔ Parse ➔ Animation Generator ➔ Animation

While there are many interesting computer vision problems that need to be handled for the tokenizing step, we will concentrate mainly the parsing step and to some extent the generator step.

Outside this strict compiler context, our implementation has two key steps:

a) Parsing and generating animation clips
b) Synchronizing events

An animation clip is a motion that an entity will go through, from start to finish. An entity may have multiple animation clips associated with it, each played at different times.

Events specify times when two or more animation clips must be at specific point in their playback. They implicitly specify when animation clips start and end, providing a simple mechanism for synchronizing the entire animation.

4 Language

The visual language is key to the system, as it is the main interface for the user to specify animations.

A sketch is generally made up of the following components:

- Entities
- Motions paths
- Commands
- Events

From these components, animation clips are generated and synchronized.

4.1 Animation Clips

As explained earlier, an animation clip represents a movement an entity goes through, from start to finish. The attributes of an animation clip are as follows:

- **length**: The amount of time the clip takes to play from start to finish.
- **motion**: The position the object is at any point within the time period it represents. Many representations are possible, though we consider this as position at each frame
- **events:** A set of times relative to the animation clip's timeframe. Related events in different motion clips will need to be played back globally at the same time.

The motions of an animation clips must be played as a whole. If you wish to have the character go through motions that need to be synchronized to more than one event, it may be necessary to split up the clip into multiple parts. An example of this would be a baseball player catching two balls thrown at different times. One clip would represent catching the first ball, while the other clip would represent catching second.

To specify that a motion path is the start of a new animation clip, a **square** is placed at the beginning of it.

In between such clips, one can assign an idle animation, which cycles through until moving to the next clip. This idle animation is specified by using the '*' on the last motion of an animation clip.

4.2 Entities

Entities are the objects that will be animated. For entities that represent static objects such balls, their drawing only specifies which object will be moved.

4.2.1 Entity Handles

For more complex entities representing characters, the entity provides handles to control the character. In the case of a character with a bone structure, these controls are inverse kinematic (IK) handles. This allows one to control an entity as a set of individual parts.

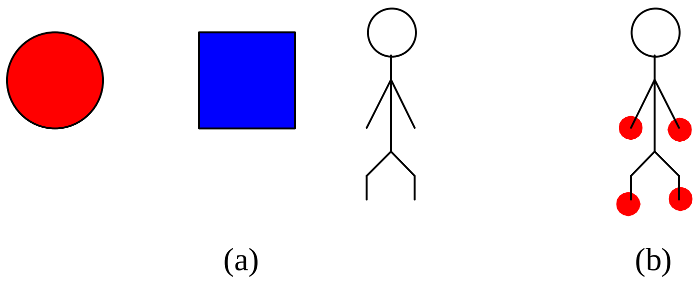

Figure 1

(a) Sample entities. The first two are simple shapes, while the latter is a bone structure. (b) Entity IK-handles of bone structure highlighted. Can be animated individually using motion paths.

4.3 Motion Paths

Motions paths are the core motion annotation for sketches. A motion path is defined using an arrow, describing the path along which an entity will move. To extend an animation further, motion paths can connect to the previous motion path.

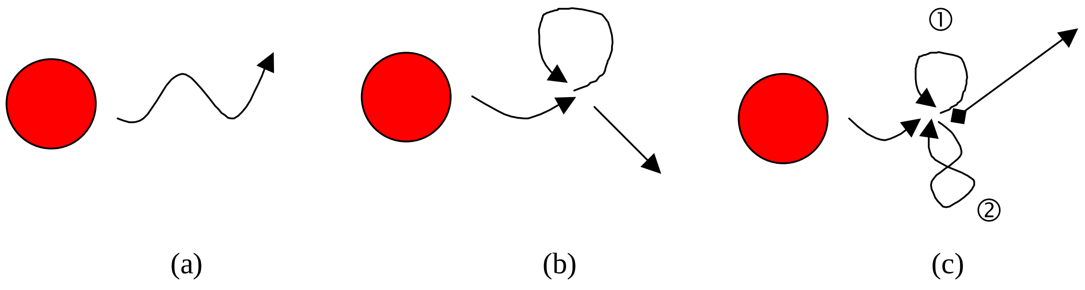

Figure 2
(a) Single motion path (b) Motion paths with implicit ordering, first through loop then out
(c) Motion path with explicit ordering, with paths numbered to show order.
The square on the last motion path of (c) means that motion path represents a new animation clip.

When parsing motion paths, we start from the entity and work outwards. From the entity, we determine motion paths whose start position is close to the entity. The first path connecting to the entity describes the first piece of motion in the animation clip to be generated. Next, we search the end point of each motion path, for new motion paths, assigning them to be the next part of the animation. This process continues recursively until there are no more connection paths.

Pseudo code of the process described is as follows:

```
AssignMotionPath(object, pos)
    For each unassigned motion path M:
        If dist(M.start, pos) < THRESHOLD:
            object.next = object.next ∪ {M}
            M.prev = object
            AssignMotionPath(M, M.pos)
        End If
    End For
```

Each motion path has a set of ‘next’ motions to go through. In order to disambiguate this situation when building our animation clips, we need to order these motions. In most cases, we implicitly determine an ordering by following motions whose ending position is the same as their starting position. By following these looping motions first, the path the entity takes will be continuous. In some cases this restriction may not be enough, such as when there is more than one loop. In these instances, we require the ordering to be explicitly labelled by drawing a circled number next to the motion paths.

4.3.1 Paths for Non-Translational Motion

By default, a motion path defines the translational position of an object. In some cases this is adequate, but we would like the path to describe more.

One possible solution is that of Motion Doodles [5]. Instead of drawing a straight line representing where the character should go, a doodle representing how the characters feet should step is drawn.

Another possible solution could incorporate an artificial-life system into the animation. The sketch itself could then describe a path that the character will follow as best it can, dependant on many factors, from objects getting in the way to other things catching its attention.

Later in this report, we examine how we can incorporate physics into a sketch by having the sketch represent a target position of a proportional derivative (PD) controller.

4.4 Compound Motions

Compound motions allow us to treat a set of motion as a single logical unit, in a similar way as brackets '( )' or braces '{ }' would in a programming language. By surrounding a set of motions paths in a box, they are treated as one.

There are several syntactic rules for compound motions. First, a motion path is considered inside the box only if the entire line is within it. This allows lines to pass through the area represented by the box without being considered part of the compound motion. Next, all paths inside the compound motion must be for the same entity. Finally, all paths inside the compound motion must be from the same part of the animation. Thus, a motion can't leave then re-enter a compound motion.

Compound motions may nested. Thus a compound motion can be inside another compound motion.

The main application for compound motions is commands, which are described next.

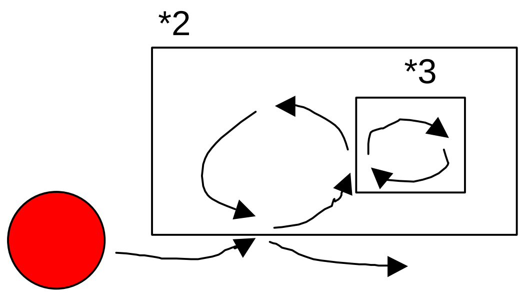

Figure 3
Nested compound motions, cycled a set number of times

4.5 Commands

Motion paths can have commands associated with them, which change or add to their motion. For our purposes commands are text-based, though there is no reason why certain commands can't be represented pictorially. A command is associated with the motion path or compound motion that it is closest too.

Commands themselves could define a whole language in and of themselves. For our purposes, we will consider only a simple command language. There are two major categories of commands:

- Modifier commands
- Playback commands

Modifier commands modify the animation generated by the motion path. For example, the command “run” could be associated with the line, and the line would represent a running animation. The line itself can be considered the ‘parameters’ to the command.

Basic modifiers include setting the velocity, such as ‘v = 10’. The effect of a modifier continues to motions paths coming after the motion path it was associated with.

Playback commands add to the animation generated by the motion path. A basic type of playback command is the cycle commands, which is specified in the form ‘* [number]’. This command will cause the associated motion path or compound motion to be played back a set number of times. Another playback command is ‘wait [seconds]’, which causes the animation generated by the motion path to wait the specified number of seconds after the motion has occurred.

4.5.1 Functions

While the above commands are hard coded, it is possible to extend the set of commands by declaring function. A modifier function specifies a set of motions an entity should go through while moving across a motion path. An example of this is a set of looping key frames for walking. The entity continues to follow the path while playing back this animation. A playback function, on the other hand, defines an animation that is played back before continuing through along the motion path.

Functions are defined using compound boxes. Inside the compound box, the motion is defined by specifying an entity and a motion. The fact that the box represents a function is specified by tagging it with the text “function [name]” at the top left, where [name] specifies the name of the function.

Calling a function is simply requires specifying the name of the function as a command.

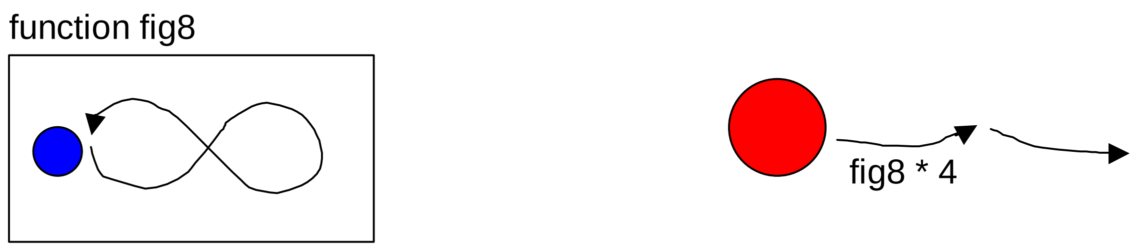

Figure 4
Left: Playback function “fig8” defined. Right: Animation that moves across the first motion path, then calls fig8 motion 4 times, then follows the second motion path.

5 Events

With many motion paths for different entities drawn on a sketch, a mechanism is needed to synchronize the resulting animation. For example, suppose we have a baseball scene with a hitter and pitcher. If we were to simply draw motion paths that cause the hitter to swing the bat and the pitcher to throw the ball, there is no way to guarantee that the swing will occur when the ball comes with in range. To properly synchronize such a situation, we introduce the notion of event marks.

Event marks are drawn as circles along motion paths in areas outside of compound motions, and are tagged with either a name or a number to specify which event they are. Sets of event marks on different motion paths represent a single event. When animation clips are synchronized, the entities associated with them will be located at the event marks at the same time.

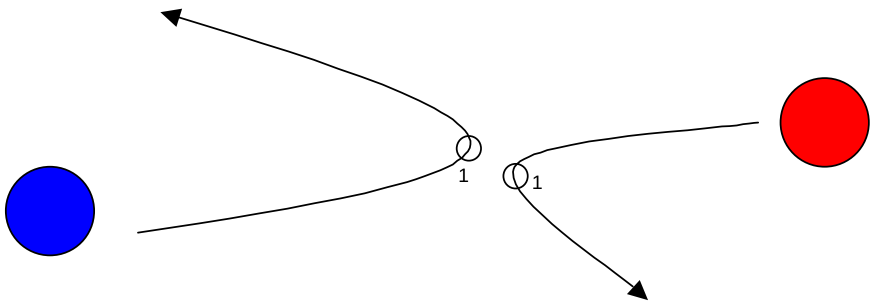

Figure 5
Event requiring the red and blue balls to be at the center of the animation at the same time, appearing to bounce of each other.

5.1 Synchronization Algorithm

First, lets consider the simple case of two event marks for the same event on two different animation clips. Let e_1 and e_2 represent the time which those event marks represent relative to their clips. We need to solve for start times s_1 and s_2 for the clips, such that the events occur at the same time.

Thus,

$$s_1 + e_1 = s_2 + e_2$$

If we arbitrarily set s_1, we can solve for s_2. This can easily be extended to three or more event marks for the same event.

Now, to extend this to more events, first let us consider an acyclic graph made up of two types of nodes, animation clips (rectangles) and events (circles). For each event, we solve for the start time of the connecting animation clips. We then recursively solve for the event times of other events connecting to each animation clip, until finally all events have been scheduled.

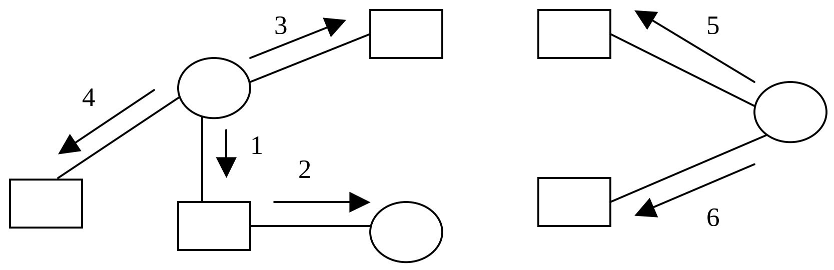

Figure 6: Event graph, with circles representing events and rectangles representing animation clips. Arrows show the order the graph is traversed to synchronize events.

The pseudo-code to handle this is as follows:

```
Synchronize:
    For each event e:
        SynchronizeVisit(e, 0);


SynchorizeVisit(Event e, float eventTime)
    If e has already been visited, return.
    For each unscheduled clip c synchronized by e:
        Let $e_c$ represent the time event e occurs relative to clip c.
        Schedule c with start time:
            $s_c$ = eventTime - $e_c$
        For each event f synchronizing clip c
            Let $f_c$ represent the time event f occurs relative to clip c.
            SynchronizeVisit(f, $s_c + f_c$)
```

5.2 Using events to determine relative clip lengths

The requirement that the graph be acyclic can be relaxed if we allow the amount of time a clip takes to be modified. In the case of a two events e and f across the same two animation clips, we now have two points where the event must occur. If we let m_1 and m_2 be multipliers controlling the size of the animation clips:

$$s_1 + m_1e_1 = s_2 + m_2e_2 \quad \text{and} \quad s_1 + m_1f_1 = s_2 + m_2f_2$$

If we arbitrarily set s_1 and m_1, we have 2 equations and 2 unknowns, s_2 and m_2. Solving for them allows us to synchronize the clips.

6. Physics

There are times where a drawn path may not provide an appropriate path for the object to follow. For example, when drawing ball bouncing off the floor, the sketched motion will naturally touch the floor. Since the motion path specifies the location of the center of the ball, when the path touches the ground the bottom of the ball will pass through the ground.

The approach we take to solve this problem is to control the entity in a physical simulation using a proportional derivative (PD) controller. The motion path describes the target location at any point of time, pulling the object towards that location.

In order to control the force of the PD controller, we augment the motion path so that its thickness describes how much the controller affects the entity. A very thick line results in a strong controller, whereas a dashed line represents the controller putting no force on the entity. Dashed lines would be used where the drawn lines would not adequately describe the motion, such as during collision with the ground or other objects, while darker lines would be used to force the object into a specific path.

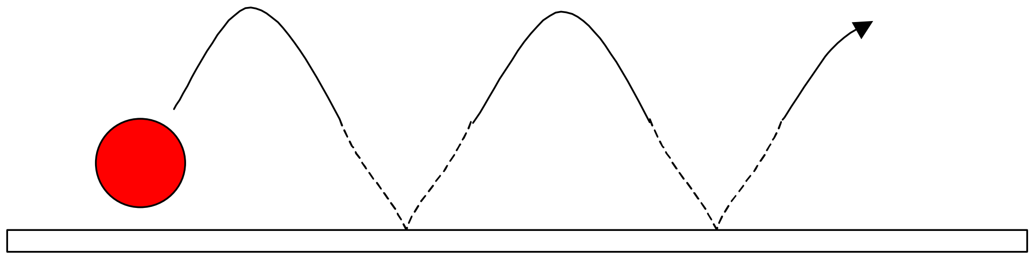

Figure 7: Motion controlled by PD controller. Controller's has no effect along dashed lines.

7. Prototype

We have constructed a prototype to parse a subset of the language described in this report. The prototype is constructed using the Maya 5.0 [1] as both MEL scripts and Maya plug-ins.

The following features are supported:

- Entities
- Motion paths (however direction specified internally, not using arrow heads)
- Compound motions
- The following commands:
    - Cycles (* [number])
    - Playback functions
- Events, where the event graph contains no cycles
- PD-controlled motion (by manually setting PD controller field as an entity)

The following features are not supported:

- Entity handles (they are only supported in the sense that you can consider a handle as a separate entity)
- Implicit and explicit ordering of motion paths
- Idle animations in-between animation clips
- The following commands:
    - wait, setting velocity
    - Modifier functions
    - Keyframes

8. Results

Using this prototype, we created several animations to test the strengths and weaknesses of our language.

We will consider results from the following scenes:
1. Two paddles playing a round of Ping-Pong.
2. A bat, controlled with an IK-handle to make it swing, hitting a ball
3. Ball bouncing on floor

In most cases motion paths worked well for specifying translational movement. There is of course a clear meaning as to what the line represents in terms of the animation. We also found that motion paths worked well for controlling rotation, when assigned to an IK-handle of a bat for the second animation. In this case, however, it can be somewhat difficult to determine the exact location of the bat along the motion since the drawn rotation isn't necessarily an exact circle.

In terms of the organization, we found that it was difficult to keep sketches for fairly long animations from becoming cluttered. In the Ping-Pong scene, it took many bunched up motion paths to represent a volley. A possible solution in this case would be to allow functions that control multiple entities. This would allow us to represent a volley from one side to the other then back as a function.

We found that all cases, the event system worked well. It was easy to determine appropriate spots to place events to get the desired results. In the Ping-Pong animation, the events were clearly labelled and it was easy to see where it would be appropriate to split animation clips for the paddle movement.

For the scene of the ball bouncing on the floor, we tested it with the PD-controller and without. Without the PD-controller, the ball goes through the ground and doesn't appear to bounce very realistically. With PD-controlled animation, we found we could allow the ball to bounce realistically by lowering the effect of the controller just before the ball bounced. Unfortunately, the transition between PD-controlled and uncontrolled animation is somewhat noticeable.

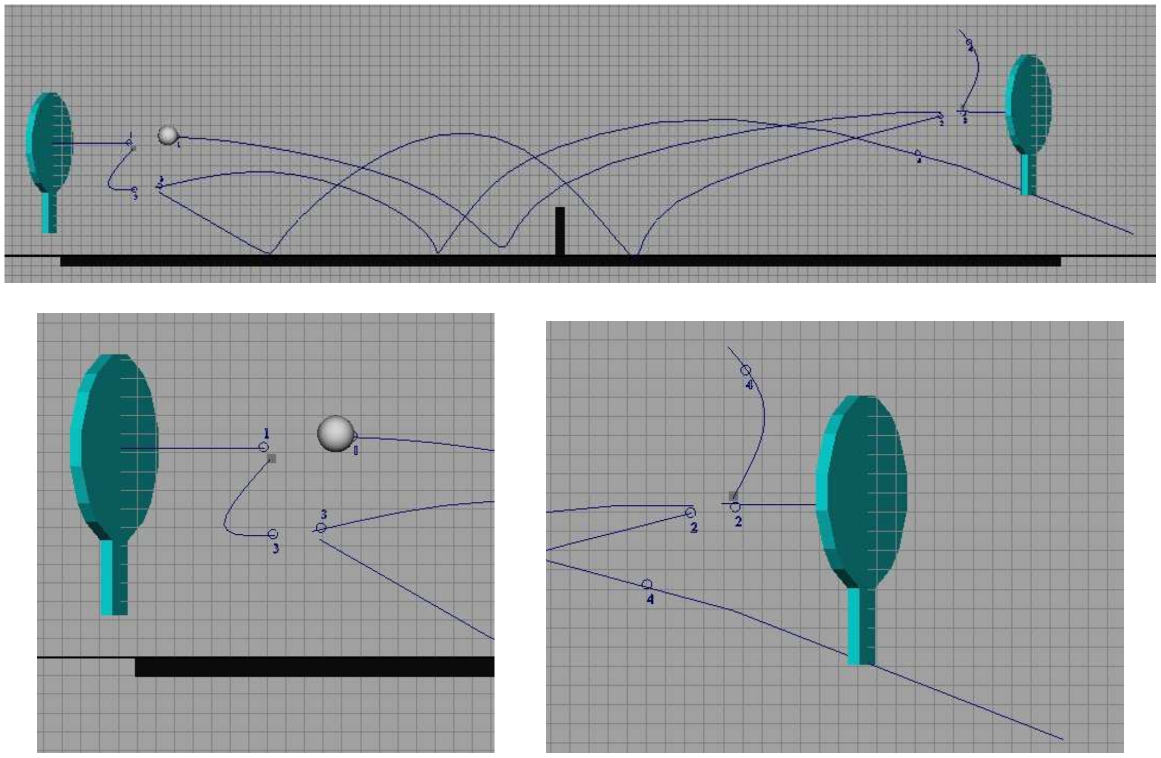

Figure 8: Ping-pong scene

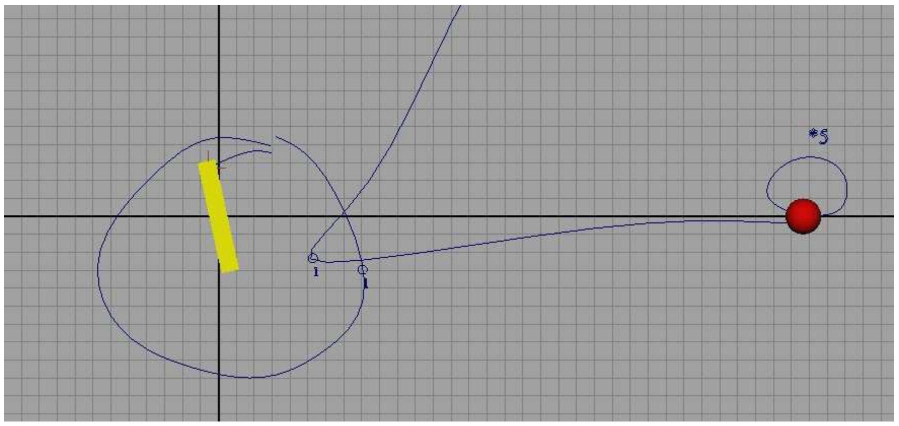

Figure 9: Bat hitting a ball

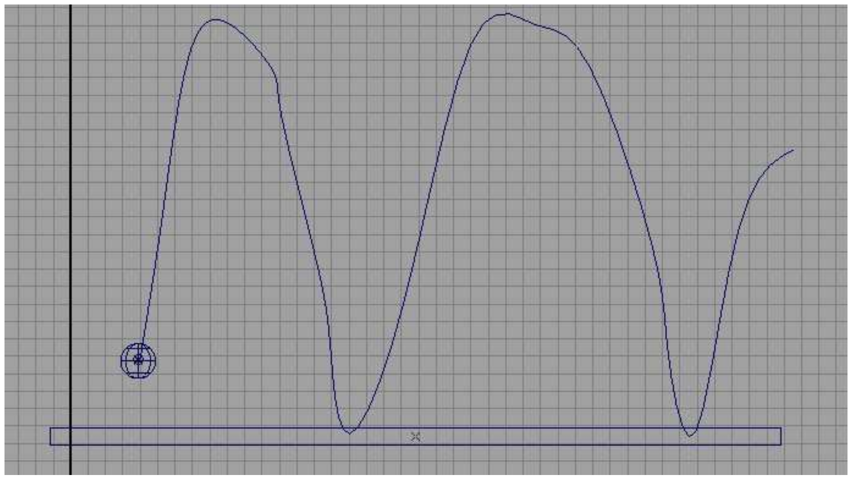

Figure 10: Ball bouncing scene. Tested both with and without PD controller

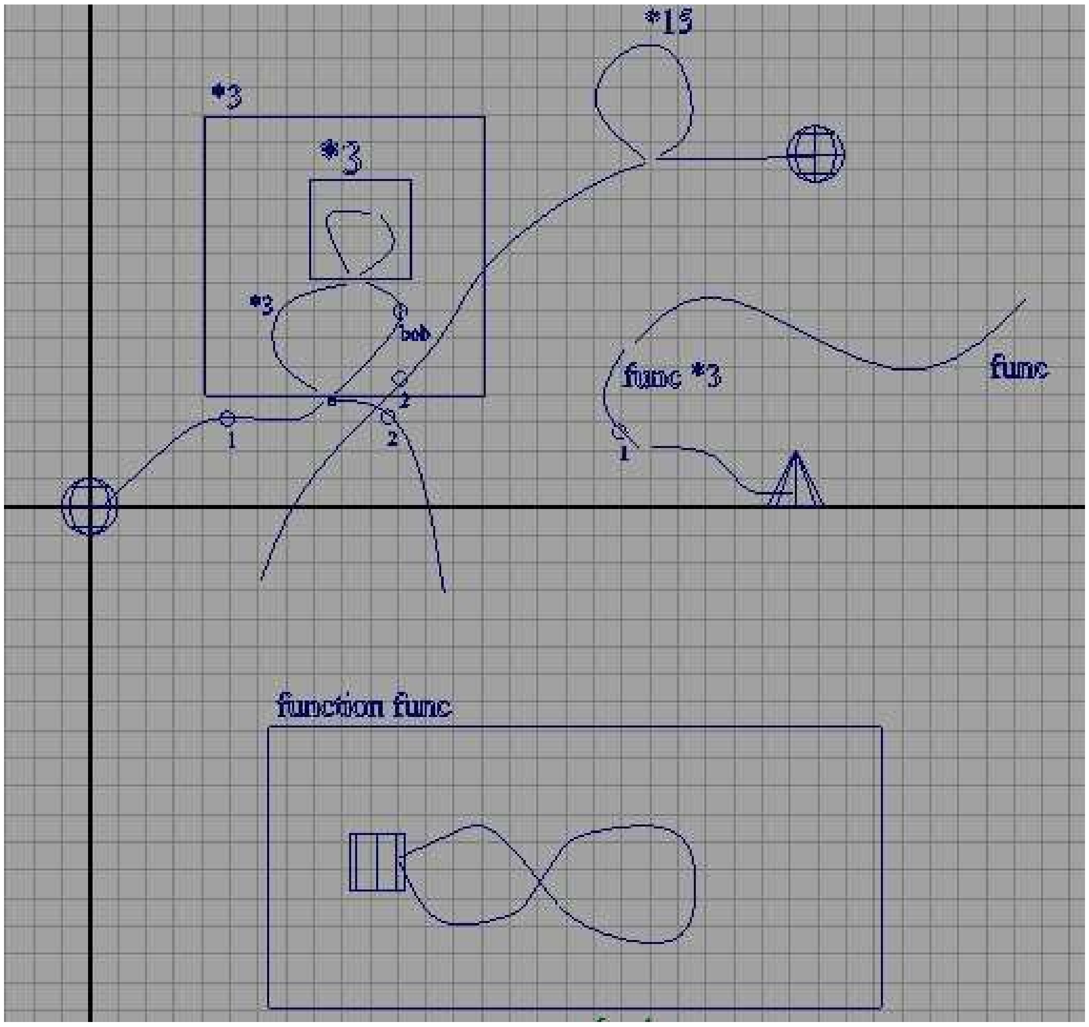

Figure 11: A scene utilizing all features of the prototype, including compounds, functions, and cycles

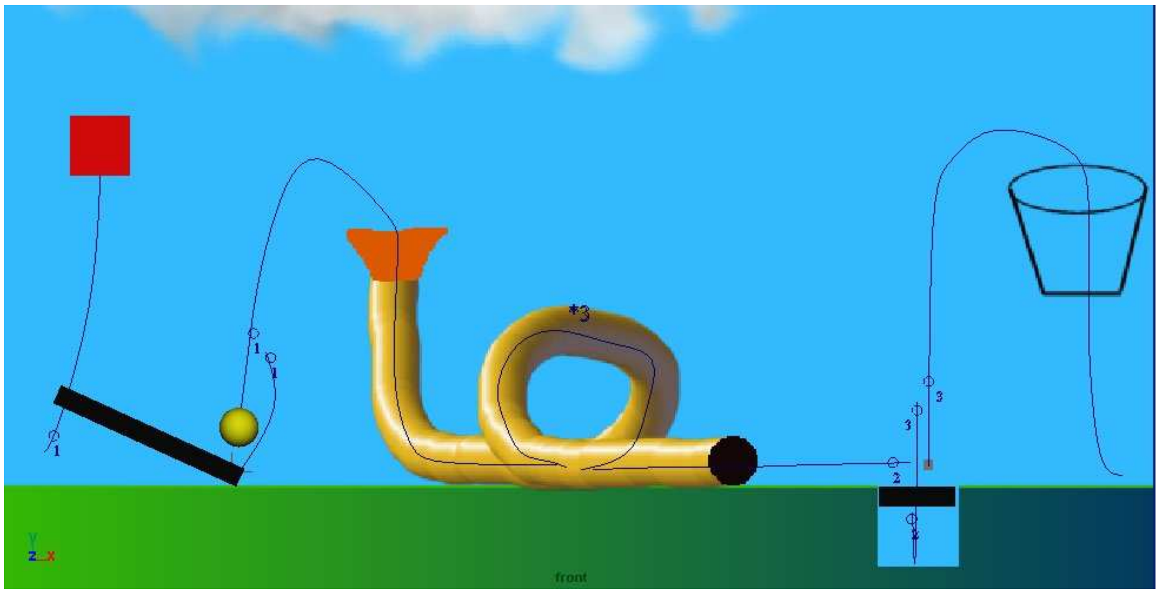

Figure 12: A Rube-Goldberg Style Machine for basketball

9. Future Work

There is room for improvement to the language that has been presented here.

The prototype itself could be extended to allow for more generalized types of motion paths, such as Motion Doodles [5]. This is especially true in the case of motion that occurs mainly in one dimension, such as walking. Using the extra dimension to describe the style of walk not only makes the system more flexible but also provides an appealing visual depiction.

The language could also be extended for non-rigid motion. This would allow one to create deformable entities, which could be deformed in various ways depending on the annotations used.

We currently deal with 2D motion, but the system could be extended to deal with 3D motion. Such a system shouldn't be too difficult to represent if the sketching interface is on the computer, but it may prove difficult to implement a system for interpreting sketches done offline on paper. Possible implementations could require the user to provide sketches from different views to constrain the set of possible animations.

10. Conclusion

The sketching language presented provides a promising method for specifying animations. The use of motion paths, events, and commands are straightforward, making up a simple enough language that should be easy to learn and powerful enough for many animation needs.

The current system is a bit lacking in terms of features, however there is room for additions to the language to support more complex animations.

References

1. Alias|Wavefront, “Maya 5.0”, Silicon Graphics, 2003.

2. J. Davis, M. Agrawala, E. Chuang, Z Popović, D Salesin, “A sketching interface for articulated figure animation,” ACM SIGGRAPH 2003.

3. T. Igarashi, S. Matsuoka, H. Tanaka, “Teddy: A Sketching Interface for 3D Freefrom Design,” ACM SIGGRAPH 1999.

4. K. Marriott, B. Meyer, “On the Classification of Visual Languages by Grammar Hierarchies,” Journal of Visual Languages and Computing 1997.

5. M. Thorne, “Motion Doodles: A Sketch-based Interface for Character Animation,” M.Sc. Thesis, Dept. Comp. Sci., University of British Columbia, 2003.

6. M. Thorne, D. Burke, M. van de Panne, “Motion Doodles: An Interface for Sketching Character Motion,” To Appear In ACM SIGGRAPH 2004.

Appendix – Prototype Language Grammar

The following is a partial listening of the reductions for the visual language as a Constraint Multiset Grammar (CMG) [4]. It describes the syntax to convert from drawn symbols into components of our language. The process of converting from these into animation clips is semantic and not syntactic, and thus not described here.

We write the reductions for motion paths and compounds in reference to their previous motions. The ‘next’ motions for these can be implicitly determined from these.

Motions Paths

Motion Path from an entity:

```
M:motion ← A:arrow,
where exists E:entity (
        A.start close_to E.pos
) and {
        M.start = A.start
        M.end = A.end
        M.prev = e
}
```

Motion path from a previous motion path:

```
M:motion ←A:arrow,
where exists prevM:motion (
        A.start close_to prevM.end
        compound(M) == compound(A)
) and {
        M.start = A.start
        M.end = A.end
        M.prev = prevM
}
```

Compounds

Motion path entering a compound box:

```
C:compound, M:Motion ← B:box,
where exists prevM:motion, A:arrow (
        A.start close_to prevM.end
        A is_in B
) and {
        M.start = A.start
        M.end = A.end
        M.prev = C
        C.dimensions = B.dimensions
        C.startMotion = M;
}
```

Motion path exiting a compound box:

```
M:motion ←A:arrow,
where exists prevM:motion, C:compound (
        prevM is_in C
        A.start close_to prevM.end
        A !is_in C
) and {
        M.start = A.start
        M.end = A.end
        M.prev = C
}
```

Commands

Command connected to motion path:

```
CMD:Command ← T.text,
where exists M:motion (
        M close_to T.bottom_left
) and {
        CMD.affect = M
}
```

Command connected to compound:

```
CMD:Command ← T.text,
where exists C:compound (
        C.top_left close_to T.bottom_left
        C.top < T.bottom
        C.left < T.left
) and {
        CMD.affect = C
}
```

Events

Connecting event to motion path

```
E:Event ← Circ:Circle
where exists M:motion (
        Circ is_on M
) and {
        E.time = arc_length(M.start, E.pos)
        E.motion = M
}
```

New-Motion Mark

Connecting a new-motion mark to a path

```
NM:NewMotionMark ← S:Square
where exists M:motion (
        S.pos close_to M.start
) and {
        NM.motion = M
}
```