Visualizing Dance Formations

Kseniya Charova
Stanford University
kcharova@stanford.edu

Michelle Gumport
Stanford University
mgumport@stanford.edu

Anwar Hahn
Stanford University
ahahn@stanford.edu

ABSTRACT
The process of designing formation choreography can be very tricky because formations rely on both shape and motion. Choreographers must not only keep track of the overall geometry of the dance, but also of the movement sequences of individual dancers through time. Our system, Dancevis, is a Javascript API that enables animated dance formation rendering in the browser. It uses groups and a hierarchical structure to mimic the motion of dancers from a bird’s eye view. Movement paths are parameterized over geometric shapes assigned to groups. We allow for the movement of dancers between groups, the display of time dependent annotations, and the ability to play, pause, and reset the animation.

Keywords:
dance, simulation, javascript, api, choreography, dancevis, formation, HCI

INTRODUCTION
The process of designing formation choreography can be very tricky because formations rely on both shape and motion. Choreographers must not only keep track of the overall geometry of the dance, but also the movement sequences of individual dancers through time. It is difficult to keep track of these details without dancers at hand, but the creative process usually takes place before rehearsal where choreographers are forced to rely on static dance notations to keep track of these variables. As Michael Noll explains, “The problem might be likened to that of a composer who requires a full symphony orchestra at his disposal in order to try different orchestrations as he thinks of them” [5]. Today composers do not need to worry about this problem because there are many music software programs that allow digital composition and instantaneous playback. However, there are fewer tools that are as effective for choreography since its emphasis on position makes it harder to encode.

Prior to teaching routines, choreographers need a way to judge the feasibility of formations and detect problem areas in spacing and orientation. They also need a way to store this information so that the specifics of the choreography can be recorded and passed down to new dancers.

Our project proposes a solution to these problems in the form of a Javascript API that allows choreographers to programmatically create formations and animate them through time. The API clusters dancers in a nested hierarchy of movement groups that compose each formation. The API is general enough to allow unique specification of the orientation and position of all the groups as well as any transitions between them. Once the choreography is specified, it is animated in a browser with interactive features for tracking dancers and coordinates.

RELATED WORK
Ann Hutchinson Guest, an expert in dance notation, has written several books on the history and comparative advantages of various movement annotation systems. One method that gained popularity in the 1600s was called the “floor plan.” This technique mapped out the positions of dancers across time, and relied on color, shape, and shape orientation to differentiate between gender, formation membership, and orientation. Although these floor plans effectively illustrated the positions of many dancers, they quickly became unreadable once movement paths were added. For example, Figure 1 shows a formation at the beginning of a ballet. It is already difficult to decipher even with very limited movement paths included. For this reason, movement was often described by either singling out individuals or couples and drawing their long individual paths across the floor, or making small multiples of snapshots of group orientations across time [3].

More recent notations such as Labanotation, which was in developed in the 1920s, focused on communicating detail about the specific movements of individual dancers. Labanotation is very complicated to read, but for those who understand it, it offers the flexibility of specifying the position and orientation of every body part of the dancer at any point in time. Since Labanotation compresses large amounts of information in a compact amount of space, entire ballets which otherwise may have been lost have been documented and preserved. Unfortunately this notation is very difficult for novices to understand since instruction manuals regarding its encoding patterns are upwards of hundreds of pages in length. In addition, this method is highly specific to body movement, and although it sometimes includes a rough floor plan to indicate dancer position, it is not very useful for creating choreography that specifically focuses on positioning relative to the rest of the group [4].

In search of a better way to represent dance, Michael Noll explored how developing technology in the late 1960s could eventually affect the creation and teaching of choreography [5]. Using a primitive computer, he organized stick figures in a 3-dimensional scene and
animated their movement according to a basic three minute ballet repertoire. Even though he concluded that considerable research into choreographic languages would be required before the process would be efficient, computers could be used to facilitate the encoding of movement and storage of choreography in an easily digestible format.

Today there are a few computer programs that allow users to manipulate figures on the screen and save the results [2]. Some extensions to these interfaces allow automatic generation of Labanotation and other tools to prevent the need to create multiple encodings of the same piece of choreography [1]. These systems have encountered a great deal of success, along with an endorsement by choreographer Merce Cunningham. However, it is unclear as to how much time and energy would be required to create extensive choreography using these systems, since they mostly focus on manipulating the movements of one or two dancers at time.

Our system builds on the ideas behind current work on the subject, but rather than illustrating multiple dimensions of movement, we specifically focus on dancer positioning and movement through time. By abstracting away body movement and focusing on position, we can make formation creation and position tracking much easier for large groups of dancers.

METHODS

We created a Javascript API that allows quick rendering of animated choreography in the browser. Javascript was our language of choice not only because it allows accessibility to a wide audience but it is also a lightweight dynamically typed language with robust error recovery. Due to the complex requirements of our system, we wanted the flexibility to use an object oriented class structure similar to that of C++ without the constraints of a strong type system. The ability to quickly add new features, object properties, and base class methods was key.

Although we originally planned to build an API on top of d3, we eventually decided in favor of using pure Javascript. Our decision to use Javascript for specifying movement was primarily motivated by our need to compute relative position coordinates on the fly. Animations in d3 rely on a system of binding data to selections, and the our specification of movement could not be easily represented by a static dataset. We did however use d3 to support interactions, as well as to set up the SVG canvas and draw the background grid lines.

The first step in our design process was to specify a high level structure for the system. Given that we only had a few weeks to fully implement this system, we thought it best to give some significant thought to how all the pieces should fit together before we set about writing any code. However, despite our best efforts, we found ourselves frustrated by how best to encode choreography and furthermore, how best to represent that state information in the computer. We will step through the various levels of representation we developed before explaining the final implementation we settled on.

Encoding Choreography

While factors other than position are of great importance for many dance forms (Note: we are using dance loosely here to refer to dancing, formation dance, group bodily motions, etc.), we are purely concerned with position and orientation. For example, if you were looking down at the dance floor, arena, etc from one hundred feet up, what interesting patterns and interactions could you observe? How could you write a program to recreate what you saw? From a bird's eye view, fancy footwork, hand positions, body posture, and facial expressions are lost and all that remains is the relative position of individuals to others in the formations. These formations often involve many interleaving and moving subparts and are much more difficult to choreograph than body movements of a particular dancer.

As we designed our encoding scheme, we tried to meet the following heuristics. First, a good encoding must provide path continuity for every dancer (hereafter referred to as continuity heuristic). Continuous paths are especially important for creating a realistic tool, because in the real world, dancers cannot teleport between positions. A good encoding must also allow flexible modification of specific paths without the need to recreate the entire encoding every time (hereafter referred to as flexibility heuristic). Finally, there must be a way to abstract movement of individual dancers into formations and create composite formations (hereafter referred to as composition heuristic). Fundamentally, formation choreography groups dancers spatially, and assigns each group a particular movement sequence, which may interact with surrounding groups. A
good system must be able to handle this flexibility in a smooth and natural manner. These criteria lead to the core features of our API: path continuity, flexibility and incremental design of choreography, and finally composite grouping.

We first discussed encoding strategies, and now we will delve into the implementation. One naive strategy of ensuring path continuity is to store an infinite number of positions for infinitesimally small time steps. However, since an infinite encoding is impractical to implement, we used an approximation that parameterized dancer position over time. With a series of time dependent functions, we can represent the path of a single dancer and even modify a dancer's path by inserting, removing, or modifying the update functions that govern its behavior. This approach satisfies the flexibility and continuity heuristics, but does not cover the composition heuristic. If we assigned unique movement paths to each dancer through the choreography, we would not be able to adequately encode composite formations, in which the change in a dancer’s absolute position is governed by the composite movement functions of each group that he or she is a part of. Given the need for a hierarchical movement system for formations, we created a data structure that contained dancer groupings.

**Groups**

Groups are the main building block of our system. On a basic level, a Group is a collection of elements that follow a specific movement pattern over a given period of time. Each Group contains a geometric shape represented by either a line or an arc, and is responsible for updating the position of its elements along that shape. The Group is also responsible for notifying its elements whenever its own position changes as a result of updates higher up in the movement hierarchy. The hierarchical structure of nested groupings means that higher level Groups contain other Groups, while leaf node Groups contain Dancers. This way, within-Group positioning consistency can be maintained through recursive updates of child nodes. The final and perhaps most important feature of Groups is that they provide hooks for user-defined update functions. The client can specify a custom function for a Group that is executed whenever a child is added or removed from the group. The client can also specify multiple update functions that the Group will use to update the positions of its children for each time step.

**Dancers**

Dancers are abstraction of actual people on the dance floor. Each Dancer is represented by an SVG element on screen and stores state such as its position, orientation, and name. Dancers belong to groups and are updated according to their parental chain. They also provide the client with mouse click event handlers that change the color of the SVG and display the name of that Dancer in a side panel of tracked Dancers.

**Shapes**

Whereas Groups provide the ability to compose formations, they would not satisfy our flexibility heuristic on their own. Our definition of flexibility is having the ability to modify a path without recreating the entire encoding. Although Groups allow nested movements, they would still require the client to specify positions at each time step, which is cumbersome. Our solution is to introduce the concept of Shapes that parameterize movement paths along well-specified pieces of geometry. Each Group must have a Shape, which defines the movement of its children. For example, a Group with three children (either other Groups or Dancers) and a circle Shape would move those three children along the path defined by the circle at each time step. Similar geometry specific motion is defined for both lines and grids, and in this way we can take our hierarchical structure of nested groupings and define a hierarchical structure of movement along parameterized paths. As a result, we satisfy the flexibility heuristic by allowing sweeping modifications of entire sub trees of our hierarchy through the simple change of an underlying shape or a slight modification of its parameters. Using Shapes to define movement is consistent with our goal of encoding formation choreography from a bird’s eye view.

![Figure 2: Dark circles and lines indicate Shapes that determine movement paths of different Groups.](image-url)
Exit Points
After defining formation groups, a choreographer may wish to transition from one formation to another. Transitions between Groups are governed by Exit Points that are placed at the intersection of each Group’s underlying geometric shape. The idea behind this concept is that dancers cannot instantaneously teleport from one position to another. Instead, they must continue traveling on their paths during transitions. As a result, the exit point of one formation must somehow overlap with the entrance point of the next. Exit Points act like magnets, and attract dancers that travel within their radius. Each time an Exit Point captures a Dancer, it transfers it to the next Group and the hierarchical tree is re-computed. Since clients can specify the time intervals over which Exit Points are active, Dancers will only transition between shapes when needed.

Time Manager
In addition to position, time is a key aspect of choreography. We built a Time Manager object that keeps track of the current simulation time and schedules the intervals over which Groups are allowed to receive position updates. Once the client has specified which groups will receive updates, he or she can use the Time Manager to play and pause as well as reset the animation.

Abstract Data Types
We used various custom data types to provide consistency across classes and error handling. For example, the Position class stores x and y coordinates, and provides utility functions that calculate equality, distance, and the ability to find a position in a particular direction. The Orientation class stores an angle and provides utility functions that convert between degrees and radians, test for equality, and compute cosine, sine, and angle differences. Our Time class stores a count of milliseconds and provides utility functions that return current time, time in milliseconds, seconds, hours, and test for equality. Our Speed class stores a number representing a speed in pixels per second and provides utility functions such as equality, and speed. Since all of our API functions use these data types, we can ensure that values are stored consistently across Groups.

User Interface Decisions and Interaction Techniques
When creating demos using our API, we approached the user interface from a minimalist design perspective. We only provided play, pause, and reset buttons and a light blue grid in the background to help with quick perceptual reasoning about distances. Since our main focus was on building a functional API that was flexible enough to encode spatial and time related data, we spent less time on building a complex user interface. Additionally, by providing only a sample UI, we hope to encourage our clients to explore the potential of building more complicated user interfaces on top of our utility functions.

Since it would be extremely difficult to extract meaningful information from the simulation without the ability to play and pause the animation, our basic UI includes several interaction techniques. To run the simulation, the client can start and stop the animation either by clicking on the Play and Pause buttons, or by hitting the spacebar. The animation can also be reset via the Reset button or the ‘R’ key. Pausing and playing allows users to take snapshots of
the scene, which they may be accustomed to having through other notation methods like Powerpoint, Keynote, or drawings.

We also added the ability for users to track individual dancers by clicking on their SVG. If a Dancer is clicked, it becomes instantly highlighted, and the name of that dancer is displayed in a list along the left side of the page for reference. To disable tracking, the client can click either on the Dancer a second time, or hit the [x] by the dancer's name in the tracking list. Either action will return the dancer's element to the normal color and remove that dancer's name from the tracking list.

Each dancer SVG element contains all of the information about that dancer's current state, so we can use that information to display detail on demand. When a user mouses over a dancer, we display that dancer's name and current position. This prevents individual dancer information from getting too messy, but allows users to see the information when they need it.

Each of these interaction techniques that we chose could easily be extended to do even more interesting things according to the needs of particular users.

RESULTS
We built several choreography demos with our API to explore its basic functionality. Some of the features that we tested included group switching, selective activation of groups, composite shape formations, and dancer tracking.

The API can handle arbitrary compositions of shapes and movements, but to work with this level of generality the client must specify the orientations and positions of each group individually.

Groups
The main building block of our API is the Group. All Groups are attached to a geometric shape and a time interval during which they are active and will update the positions of their children. Since each group controls movement, it also contains a speed and a position.

Creating a sample Group containing a circle:

```javascript
var outercircle = new dancevis.Shapes.Circle(origin, radius);

var outer = new dancevis.Group({
    shape: outercircle,
    startTime: dancevis.Time.now(),
    endTime: new dancevis.Time({seconds:20}),
    speed: new dancevis.Speed({speed:70}),
    position: outercircle.center,
    orientation: new dancevis.Orientation(0)});
```

The Shapes of the Groups represent the movement paths of its children, and we provide the option of toggling the visibility of these paths. This is useful when the choreographer would like to switch between creating formations and showing what they should look like to their dancers.

Setting the parameter to “false”:

```javascript
outer.showShapeOnScreen(false);
```
removes the shape path from the screen.

Update Function
Since not all groups will have one continuous motion associated with them, there is a way to specify custom update functions over time intervals, as long as the interval fits within the lifetime of the Group. The following update function will rotate all of outer’s children clockwise from 12 seconds until the group is no longer active.

```javascript
outer.setUpdateFunction("clockwise" = new
dancevis.Time({seconds:12}),
outer.endTime,
    function(){ this.shape.clockwise = true; });
```

Transitions
Transitions between groups are marked by exit points. Since dancers transition between formations smoothly, the exit of one formation must align with the beginning of the next. These exit points act like magnets, switching each dancer that travels within its radius to the next Group. The client must specify a time interval over which the exit point is active.

Set an exit point:

```javascript
outer.addExitPoint({
    startTime: new dancevis.Time({seconds:15}),
    endTime: line1.endTime,
    position: line1.shape.startPosition(),
    nextGroup: line1,
    showOnScreen: true,});
```

Dancers
Dancers live at the bottom of the group hierarchy as leaf nodes. Dancers calculate their positions based on relative offset from their parent groups and draw themselves on screen. Each dancer can have a unique color and name, so he or she can be later identified and tracked during the animation.

```javascript
var dancer = new dancevis.Dancer({orientation: new
dancevis.Orientation(60*i, false),
    dancerColor: "red",
    dancerSize: dancevis.DancerShapeSize.LARGE
    dancerName: "Sally"});
```

Dancers are added to their parent by the following call:

```javascript
dancer.setParent(outer);
```

Annotation
The API provides a tmanager object that handles time sensitive functionality. The client can use the tmanager to schedule groups, add a timer to the screen, or to create annotations that will pop up and last for the specified time interval.

```javascript
tmanager.annotateAt(new dancevis.Position(-300,100))
   .annotate("Notice the short stop here!",
       new dancevis.Time({seconds:3}),
       new dancevis.Time({seconds:5.5}))
```
Ease of Use
A client of our system created a short demo using our API after having previously created a similar one in PowerPoint. The time spent using both methods was roughly equivalent, but our API produced a more accurate visualization since it automatically handled all movements and transitions. Our visualization also provided a timer and a way to track individual dancers which was not available in Powerpoint.

Figure 5: This simulation of dancers weaving in a figure 8 pattern would involve a lot of manual specification in PowerPoint, but can be achieved with simple arcs using the API

A major advantage of our API is that it produces visualizations that can be easily scaled and extended. For example, an existing encoding of choreography can support the addition of an arbitrary number of dancers without forcing the client to specify each new dancer’s path individually. Furthermore, complex spatial formations can be build on top of existing Groups without the need to specify each frame of the sequence.

The majority of time spent using our API and the bulk of the code belongs to the initialization phase, where the client must specify the positions and orientations of Groups. Although this is a little bulky, there is no easy way to support arbitrary formations without asking the client to specify custom coordinates and update functions. However once the setup is complete, the underlying motion and spacing is handled behind the scenes so the amount of work required to add more elements and simulate formations is much smaller than other methods with an increase in scale.

DISCUSSION:
One realization that users of our system will have is that defining choreography is a difficult problem. Dance is ubiquitous in our lives and most people have been exposed to marching band routines, parades, as well as formation dependent performances such as ballet, synchronized swimming, and dance. Although the formations we see and the transitions between them seem fluid and seamless, designing such dynamic performances in which many moving pieces must seamlessly interact requires a lot of thought and coordination. For example choreographers must pay attention to possible collisions, spatial constraints of the floor, and start and end points of each dancer, in case formations require dancers to separate and later rejoin their partners in a different location. These require experimentation to get right, regardless of whether they are first tried with a simulation or real people. Moreover, different types of choreographies rely on different interactions between individuals (couples for ballroom formations, instrument sections for marching band), and work with different shapes. As a result, a system that can encode any type of spatial choreography would offload a lot of the customization to the client.

Breaking down the idea that choreography is simple and shedding light on the complicated mechanics behind elegant choreography is one of the motivations for building our system. We created an API that greatly facilitates the encoding of movement data through time, but it is only a step towards a solution for effectively encoding all aspects of dance choreography.

FUTURE WORK
There are many possible extensions for this work. Firstly, movement in our system is restricted to circular and linear primitives. It would be valuable to add in motion over more complex paths, such as Bezier curves. It would also be helpful to provide users with more control by ensuring that every configuration of dancers supports the typical translation, rotation, and scale operations used in animation.

We only encode position right now, relying on annotations to convey most of the meaning about specific dance steps. Our system could be extended to support dancer orientation (what direction are they facing?). We could add in support for specifying specific movements, or allow integration with Labanotation systems.

We also currently only support playing, pausing, and restarting the simulation. We would like to add more fine-tuned time control such as rewind, fast forward, and jumping to certain times in the choreography. Given our current design of the time manager, we cannot easily allow this functionality because we would need to restructure the way we break apart and form Groups when time is reversed.

Our UI could certainly use some polishing, but it is only a sample of one of many UIs that could be built using the system we developed, and it integrates some possible interaction techniques. More research would need to be done into how our system actually gets used to determine the optimal usage of screen real estate. There are also a wide variety of interaction techniques that could be used to interact with the elements on the screen, from toggling on/off the display of groups or dancer information, to more
advanced tracking of dancers across the screen. In addition, since dance requires music, we could integrate the music with the timer, to give further context to the dance. Looking even further ahead, we could build a more complex UI, which allows users to create choreography through the interface, so that choreographers without programming experience could use it as well.

REFERENCES


