Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Jaakko Lindvall

Virtual tutor for dance games:

Design and implementation of procedural animation and artificial intelligence

Master's Thesis
Espoo, May 18, 2014

Supervisor: Professor Jaakko Lehtinen
Advisor: Professor Perttu Hämäläinen

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Jaakko Lindvall

Title:
Virtual tutor for dance games: Design and implementation of procedural animation and artificial intelligence

Date: May 18, 2014 **Pages:** vii + 92

Major: Media Technology **Code:** T-111

Supervisor: Professor Jaakko Lehtinen

Advisor: Professor Perttu Hämäläinen

Dance games are a type of rhythm games, in which the player steps on buttons on a dance pad, following the beat of a song. In these games, the learning curve for the novice player can be steep, because they rarely offer any explanation of how the player should actually move on the dance pad. Even for the experienced player, interpretation of patterns in the harder songs can be tricky.

In this master's thesis, we describe the design and implementation of an animated tutor character for dance games. The character will dance to a particular song in the game, demonstrating how the step patterns for the song are played. A person playing the game can then learn to perform the song by following the demonstration.

The dancer character is animated with procedural animation techniques, including inverse kinematics and PID control. Additionally, we describe an artificial intelligence for dance games, which drives the moves of the dancer.

Keywords: character animation, dance game, inverse kinematics, artificial intelligence

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jaakko Lindvall		
Työn nimi:	Virtuaalinen opetushahmo tanssipeleihin: Proseduraalisen animaation ja tekoälyn suunnittelu ja toteutus		
Päiväys:	15. toukokuuta 2014	**Sivumäärä:**	vii + 92
Pääaine:	Mediatekniikka	**Koodi:**	T-111
Valvoja:	Professori Jaakko Lehtinen		
Ohjaaja:	Professori Perttu Hämäläinen		

Tanssipelit ovat eräs rytmipelien tyyppi, jossa pelaaja askeltaa tanssialustalla ja painaa siinä sijaitsevia nappeja, seuraten rytmejä musiikkikappaleessa. Tanssipelien oppimiskäyrä voi olla aloittelevalle pelaajalle korkea, koska pelit harvoin sisältävät mitään opastusta siitä, kuinka pelaajan oikeastaan tulisi liikkua tanssialustalla. Kokeneellekin pelaajalle vaikeampien kappaleiden askelkuvioiden tulkinta saattaa olla konstikasta.

Tässä diplomityössä kuvaamme, kuinka tanssipelin animoitu opastehahmo suunnitellaan ja toteutetaan. Hahmo tanssii pelissä olevan kappaleen tahtiin, ja näyttää kuinka kappaleen askelkuviot tulee pelata. Peliä pelaava henkilö voi opastehahmoa seuraamalla oppia pelaamaan kappaleen.

Tanssijahahmo on animoitu proseduraalisilla animaatiotekniikoilla, kuten käänteiskinematiikalla ja PID-ohjauksella. Lisäksi työssä kuvataan, kuinka toteutetaan tanssipelejä pelaava tekoäly, joka ohjaa tanssijahahmon liikkeitä.

Asiasanat:	hahmoanimaatio, tanssipeli, käänteiskinematiikka, tekoäly
Kieli:	Englanti

Acknowledgements

I wish to thank Jenny Tirkkonen for supporting me during the thesis, and especially for designing a great-looking 3D character model for the dancer.

I would also like to thank my advisor Perttu Hämäläinen for steering me into the right direction several times, and my supervisor Jaakko Lehtinen for equally good insight.

Finally, I wish to thank Jari Vilkki and Jari Saukkonen at Trix Games for helpful comments, and for the joint development of our (still unfinished) game.

Helsinki, May 18, 2014

Jaakko Lindvall

Contents

Chapter 1

Introduction

In this master's thesis, we will design and implement an animated tutor character for dance games (see Figure 1.1). Our goal is to have this character dance to the routines (stepcharts) in the game, and demonstrate to the player how a particular song is played. Dance games rarely offer any explanation on how the player should actually move on the dance pad, so the learning curve for the novice player can be steep. Even for the experienced player, harder songs can contain difficult patterns that can be tricky to interpret.

In addition to being a learning aid, the dancer can help creators of the stepcharts (stepartists) verify that the patterns they have created are played the way the stepartist intends. In creating a stepchart, it is easy to make subtle errors in the step patterns, often altering the way the stepchart is played for a long section. Therefore, this kind of verification is helpful.

The focus of this thesis will be on how to animate the dancer, and produce smooth, good-looking and realistic motion. As we have to be able to play arbitrary content, the set of moves we have to support is large. Therefore, procedural methods have to be used to synthesize the required motion.

Note our choice of terminology in the following chapters: we use the term *dancer* or *character* of the virtual tutor character we are implementing. From an actual person playing the dance game, we use the term *player*.

1.1 Thesis structure

We will now lay out how the thesis is structured.

In Chapter 2, we explain the background details necessary for understanding the following chapters, in which the dancer is implemented. We describe dance games in greater detail, and motivate why the implementation of the tutor dancer is desirable. We then list all the use cases we would

Figure 1.1: Person playing a dance game [45]

like the dancer to be of value, and requirements for the project that support these use cases. Then we explain what are the possible contents of a dance game stepchart, and how the player of the game will move as a result of patterns in the stepchart. With this knowledge, we can understand the required moves the dancer has to be able to perform. We end the chapter by introducing some guidelines that apply to both traditional animation and computer animation, so that we can them into account in our implementation.

In Chapter 3, we then describe the design and implementation of an artificial intelligence for dance games. The AI will output the moves the dancer has to perform in order to play a particular dance game stepchart. The design of the AI is based on generating the decision tree for all possible sequences moves that can be performed to play the stepchart. From the decision tree, a best route is then searched and returned for the dancer to use.

In Chapter 4, we explain the methods that are used in animating the tutor dancer. We introduce fundamental concepts of computer animation, and then a few techniques that can be used for procedural motion synthesis.

Chapter 5 will then describe the design and implementation of procedural animation in the dancer character. We propose an initial design in which the

character is animated with inverse kinematics in a rudimentary way. Once the design is implemented, we propose several improvements to it that enhance the simplistic motion of the first prototype.

Ultimately, timing issues with animation clips with our platform prohibit the use of pre-recorded animation as a part of the system, so several possible designs cannot be implemented. In the final design, we then continue to use inverse kinematics, but in combination with PID controllers – smoothing and adding inertia to the generated paths of motion.

In Chapter 6, we will then evaluate our implementation. To determine the quality of the dancer animation, we release a demo of the system to the web, and run a public survey on the animation quality. The survey yields many responses, many containing some form of helpful constructive criticism.

Finally, we summarize key points of the thesis in Chapter 7.

1.2 Contributions

This thesis describes the implementation of the dancer character, and the dance game artificial intelligence (AI). The program code for both was written solely by the author of the thesis, although the former was developed as a part of the thesis, and the latter was earlier work. The dancer was illustrated and modeled by Jenny Tirkkonen – `http://jennytirkkonen.fi/`.

Chapter 2

Background and motivation

In this chapter, we explain the background details necessary for understanding the following chapters, and motivate why the implementation of the tutor dancer is desirable. We start by introducing dance games in greater detail.

2.1 Introduction to dance games

Dance games are a type of rhythm games, in which the player steps on buttons on a dance pad or mat, following the beat of a song. For every song, a step routine or several have been created in advance, defining the times in which the player has to press a button on the dance pad. These routines are called stepcharts of the song. The dance game grades the performance of the player by how accurately he or she has managed to hit the notes in the stepchart. Missing a note entirely will yield zero points or even deduct points, whereas hitting the strictest possible timing window around that note will yield maximum points. Usually there are several possible timing windows in-between the total miss and the perfect hit.

Figure 2.1 shows the gameplay screen from (our) dance game. Similar gameplay exists in every game of the genre. Here we see two players playing a song from the game, with player one on the left hand side and player two on the right hand side. Player one is playing a harder stepchart (of the same song) than player two, so different patterns of notes are shown.

In the screen, various elements related to the gameplay are noted. In the upper part of the screen, there are four gray arrows (a): these are the target arrows. The notes in the stepchart will scroll towards these target arrows, and once a row of notes and the target area completely overlap, the player should press the respective buttons on the dance pad. Here we can see that the $\rightarrow$ (right) note has just been hit, and the $\leftarrow$ (left) note should be hit

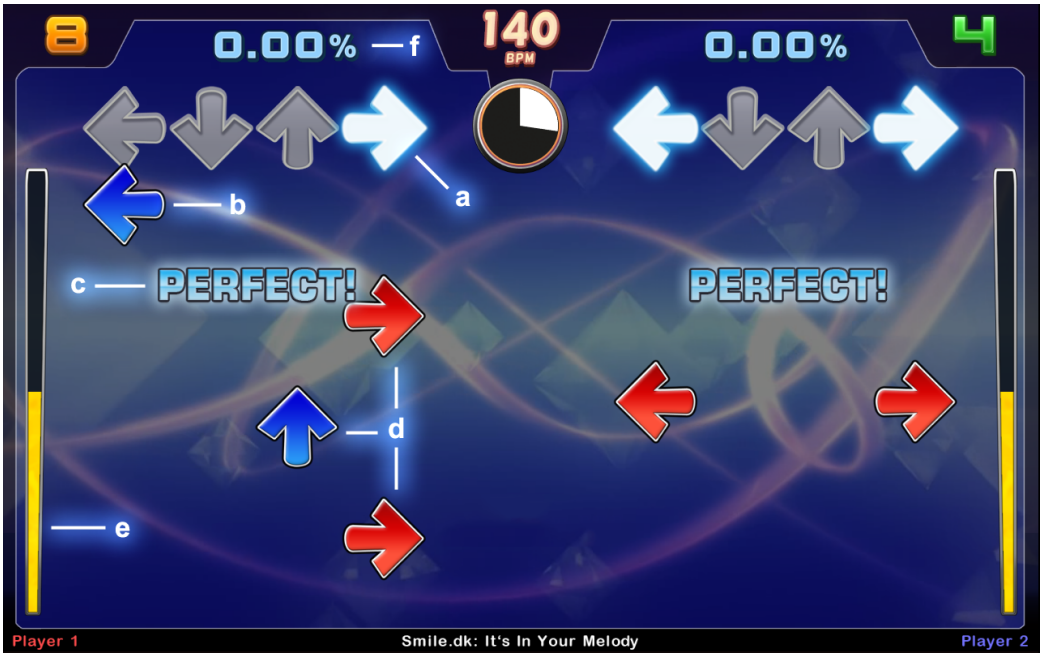


Figure 2.1: Gameplay of (our) dance game

shortly (b). The player has managed to hit the note with the best timing, so the "Perfect!" grade is displayed (c). After the last three notes that are visible (d), more notes will scroll to the screen from the bottom.

In the game, the player has a life meter (e), displayed on the left, next to the notes. If the player misses a note, or hits it with bad accuracy, the game will decrease life points from the meter. Similarly, hitting the note with good or perfect accuracy will increase the points in the meter. To pass the song, the player has to keep his life points above zero until end of the song. This is the primary objective of the game. If the player passes the song, the secondary goal is to hit the notes in the song as accurately as possible.

In addition to dance games that use the dance pad, there also exist motion capture based dance simulators where the goal is to match the movement of an in-game character as accurately as possible. One example of these games is Dance Central [17] for Microsoft Kinect. While such games are also often called dance games, in this thesis we use the term to only mean games played with the dance pad. There is indeed a large difference in how these two types of games are played. In games based on the dance pad, the player focuses only on pressing buttons as accurately as possible, whereas motion-sensing games try to judge the accuracy of full-body movement of the player from the camera image. Because of upper-body movement is ignored as input in

(a) (b)

Figure 2.2: Dance game arcade cabinets using the 4-panel (a) and the 5-panel (b) button configurations

games using the dance pad, players try to minimize the movement of their hands and torso, only moving them when it is necessary for body balance.

Dance games (from now on, using our preferred definition) are further divided by how many buttons or panels a particular game supports on the dance pad. The game that founded the genre was Konami's Dance Dance Revolution [23], which is a 4-panel dance game. These panels are made to represent all the cardinal directions: left, down, up and right. A more recent entry in the same 4-panel category is In the Groove [45], made by RoXoR Games. It is the tournament standard, and multiple competitions for In the Groove are held annually, such as the European Championships and the Finnish Championships. In addition to 4-panel games, there is the Pump It Up [1] series by Andamiro. Pump It Up uses five panels in an inverse layout compared to 4-panel games: all the inter-cardinal directions and the center of the dance pad are mapped as buttons. Figure 2.2 depicts the In the Groove and Pump it Up arcade cabinets.

2.2 Project motivation and our game

In the dance games genre, the games usually explain the basic gameplay mechanics: "Notes scroll towards the target area. You should step on the corresponding dance pad panel when the note reaches its target."

What the games do not explain, is what kind of movements the player should actually make on the dance pad to play the game in the optimal fashion. As we explain shortly in Section 2.4, one of the basic rules of play is to step so that you alternate the used feet on every step. But a beginner instinctively steps on a dance pad button using the nearest leg, and then returns to the center of the pad. This is very inefficient, and without any guidance, it can be hard to learn the proper way to play.

An experienced player is, then, is well-aware of all these rules to efficient play. But the actual step patterns that result in from these rules can be tricky to read, especially in a song with a fast tempo. The beginner and experienced player alike would both benefit from some visualization of how a particular stepchart is played.

We, at the games company Trix Games, rectify this situation by implementing a virtual tutor character for 4-panel dance games. This character, modeled in 3D, will dance to the stepcharts in a dance game, and demonstrate to the player how a particular song is played. For several years, Trix Games has developed a yet unfinished dance game called Dance Tricks (displayed in Figure 2.1); for the benefit of the player of our game, we would like to integrate the virtual tutor to this game.

Part of the necessary work for the tutor dancer has already been implemented. Prior to this thesis, we wrote an artificial intelligence (AI) for dance games, which can determine the optimal play for a given stepchart. The tutor dancer will be then controlled by the play generated by the AI system.

In the course of this thesis, we will cover the design and implementation of procedural animation for the tutor dancer. We will also cover design and implementation of the AI system. Focus will be given, however, on how to animate the tutor dancer.

2.3 Project use cases and requirements

We explained what is the motivation in implementing our project. We will now state more explicitly, in what use cases we would like the dancer character and the AI system to be useful. In the order of highest to lowest priority:

1. A new player of our game wants to learn how to play. The game should explain this by showing the player easy stepcharts and the intended way to play them.

2. A more experienced player of our game wants to see how a particularly difficult section of a stepchart is played. The game should clearly indicate what the intended way to play is.

3. A step artist wants to create new step charts for Dance Tricks. This can be either one of our own content creators, or someone making custom stepcharts for the game after release. The step editor should display how the patterns in the stepchart are played, and let the step artist review whether the motions correspond to the intended way of play.

4. A player of the game wants to see more eye candy during gameplay. It should be possible to have the tutor dancer rendered alongside the notes, playing the same stepchart as the player.

The first two cases are the ones we motivated this project with. And they are indeed the most useful applications. But for the content creators, the system should be useful too.

Let us consider requirements for the project that support these use cases. If we display the dancer next to the notes in the game, and have it synchronized with the gameplay, all of these usages are supported to a degree.

In the first two cases, the player would practice or review a stepchart in a special practice mode in the game. The player could scroll through the chart, or have a section of it played automatically, or actually play the said section. When scrolling through the chart, the dancer would display the way to play the selected note row; when playing a section, the dancer would perform this section. A music rate control should be present if the player wants to practice patterns by actually playing, so that he could slow the music down and match the movement of the dancer character more easily.

In the third case, the dancer character would display the way to play the selected note row in the editor software. Reviewing the step patterns like this is a bit slow though. As a secondary visualization, it would be best to display markers on the notes, about which leg (or hand) was used to tap the note. Example of this notation is shown in Figure 2.3. These markers can be useful for the player as well, so it should be possible to display them in the practice mode, too.

The fourth requirement is more of an extension of the first two, and thus fulfilled. In the gameplay though, the dancer should be able to be toggled off as well, because many players will surely want to just focus on the notes.

2.4 Stepcharts, notes and patterns

To understand the requirements for the AI system, let us cover what consists of a dance game stepchart. For a 4-panel game, the stepchart is a collection of multiple rows of notes, where each row may contain one or zero instances of the following notes: ← (left), ↓ (down), ↑ (up) or → (right). Therefore, one row may contain from zero to four notes. The most common case is that a note row contains just one note of some of the aforementioned directions. Then the task of the player in-game is to press the corresponding dance pad button using his or her left or right foot, whichever is more convenient given the particular pattern of notes.

2.4.1 Alternated stepping and crossovers

The basic rule to efficient play is to *alternate the used feet* on every step. So if the player starts on the left foot, then the next step should be done with right foot, then left again, and so on. Consider the three step patterns in Figures 2.3, 2.4 and 2.5. In the images, L denotes that a note is stepped with left leg of the player, and R is stepped with the right leg. On the sides you see the postures where the player will be after playing the related note row.

It is easy to see that for pattern in Figure 2.3, which contains only left and right steps, the notes are most efficiently stepped alternating between the left and right leg. And if we mix in ↑ and ↓ notes, such as in the Figure 2.4, it is still easy to press these notes using alternated stepping.

In Figure 2.5 the optimal way to play is less obvious. We start with the left leg on the ← panel and then move the right leg to the ↑ panel. But if we now continue to play using the alternated stepping rule, the left leg would move to the → panel. After this note, the player would face slightly left on the dance pad. Continuing the pattern, on the fourth note the right leg presses the same panel as previously, and then on the fifth step the left leg moves back to the panel ←.

Of course, the dance game doesn't care which leg the player presses the panels with. But to conserve energy, the player should use alternated stepping when possible. In the pattern in Figure 2.5, the player could step the first and last notes with the left leg, and in notes 3–5 use the right leg. But it's obviously more demanding to do those three consecutive steps with the same leg. Pattern such this is called a *crossover*, because it results in the left leg crossing over to the right panel, or the other way around. If the player forgoes alternated stepping and plays the crossover pattern without turning, every consecutive step of the same leg is called a *doublestep*, and it can be

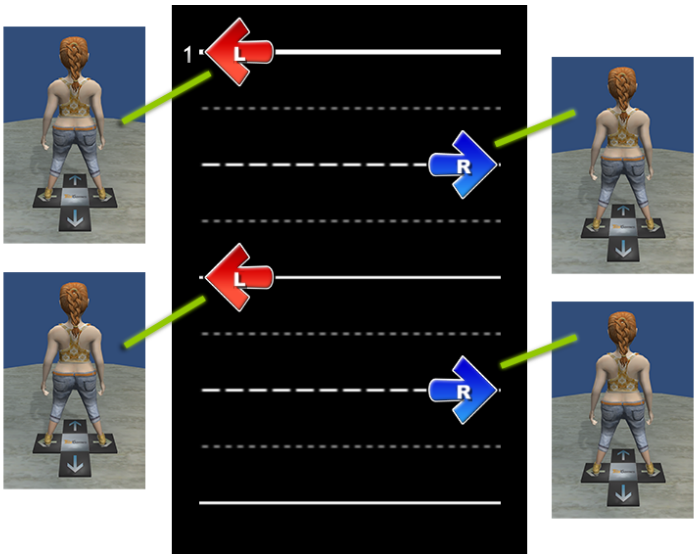


Figure 2.3: A step pattern using only the left and right panels

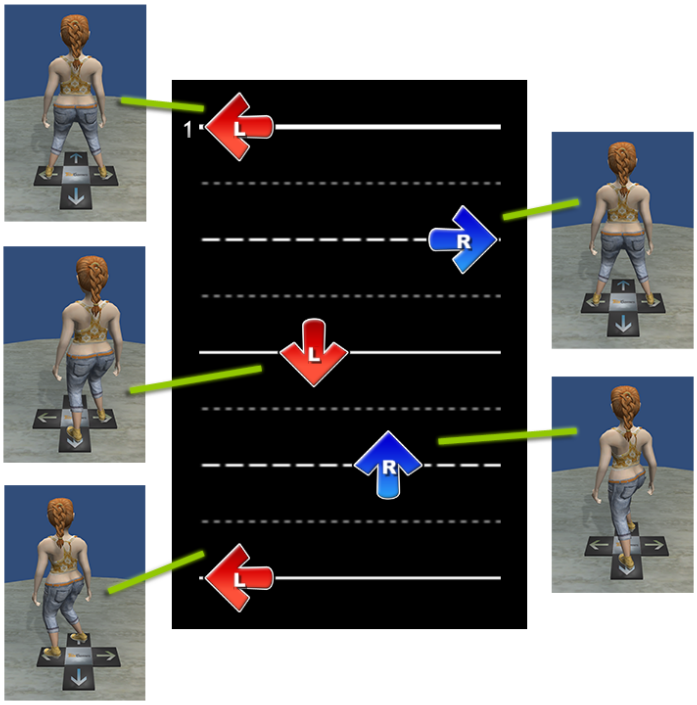


Figure 2.4: Another straightforward step pattern

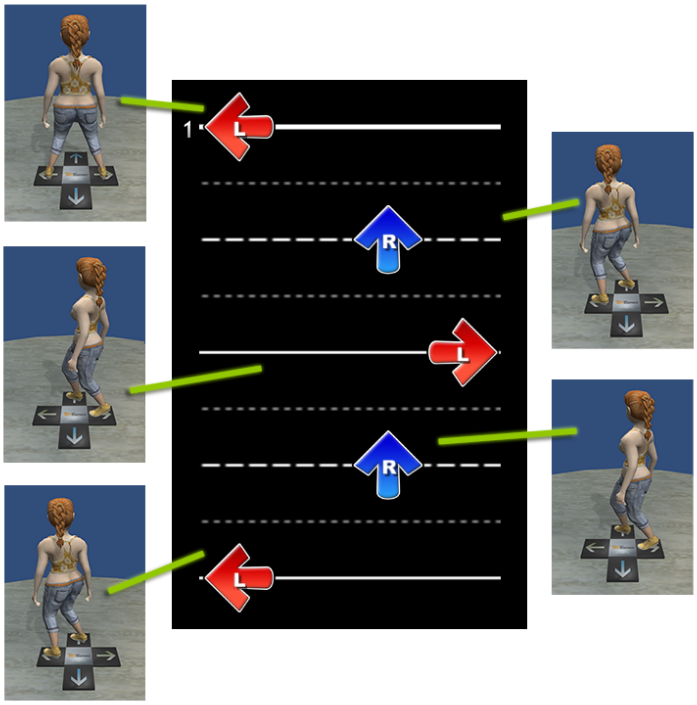


Figure 2.5: A crossover step pattern

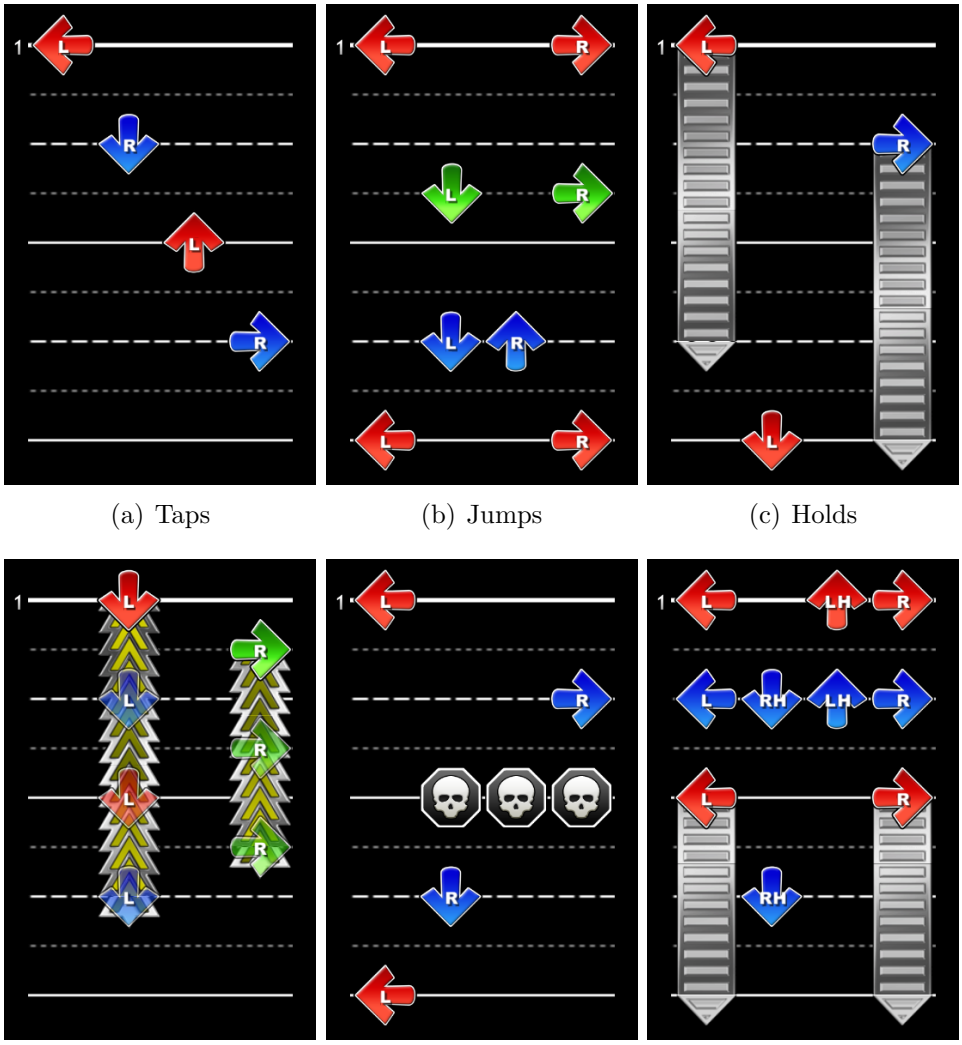
said that the player doublestepped the crossover pattern. In actual play, it's often hard to see whether crossovers should be doublestepped or played with alternated stepping. The optimal way depends on the patterns that follow, which the player might not be able to anticipate without knowing the stepchart beforehand.

2.4.2 Types of notes

In 4-panel dance games, there are various types of notes that can be used in stepcharts. In the last section, we saw patterns consisting of tap notes. Figure 2.6 lists rest of the note types that are available.

Image (a) displays the tap notes, single arrows to be pressed with left or right leg of the player.

Image (b) contains jumps, two notes in a note row that should be pressed with both of the player's legs. Jumps can also be called airs – this is to make a distinction between the actual step type (air) and the motion required to play it (jump). Still, jump is a valid term for both.



(a) Taps (b) Jumps (c) Holds

(d) Rolls (e) Mines (f) Hands

Figure 2.6: Types of notes

Image (c) contains hold notes. The hold begins with an arrow and has a body of certain length. The player has to successfully hit the note at the beginning of the hold, and keep the dance pad button depressed until the hold has fully scrolled past the target area. In stepcharts, holds are often mapped to long vocal or instrumental notes in the music.

Image (d) contains rolls. Rolls are similar to holds, but they require the player to constantly tap the dance pad panels during the roll. The player must tap rolls at a certain minimum pace, or they will be dropped. This pace varies by game, but is constant for all the rolls within the game. In the image, the transparent arrows on top of the roll body are suggestions by the AI when the roll should be tapped on.

Image (e) contains mines (represented as skull icons). Mines are meant to be rather avoided than stepped on – if the player stands on a panel and a mine on that note column scrolls through the target area, the mine will explode. The player is then penalized with a loss of some life points and score.

Finally, image (f) contains hands. A hand note is any note row where the player is forced to press the dance pad panels with hands. This can occur in two situations. Either the note row itself has three or four tap notes, forcing a jump in which the players feet hit two of the notes, and hands hit the rest. The other situation occurs when both feet of the player are on hold notes, and there's a tap note in-between the holds. Combinations of these situations are possible too, for example, the left leg can be on a hold and during the hold, there is a row of three tap notes, which must be then pressed with the right leg and both hands of the player.

In all the images, the color of the arrows denotes length of the note. The images show half of one measure in a 4/4 musical time signature. All the red notes are quarter notes, meaning there can be four of these per measure. In the same vein, blue notes are eighth notes and green notes are 16th notes.

Even though we are talking about lengths of notes, it is important to understand that the length only describes the minimum duration between notes of same length (color). A minimum duration between one quarter note and the next is a quarter note, or quarter of one measure. Tap notes are still instants in time, not sustained during length of the note, and duration of a hold is unrelated to the note length on which the beginning of the hold rests.

The spacing of the notes also doesn't change the way how a particular pattern of notes is played (with the exception of the roll, but it can be interpreted as a sequence of taps). But it affects the animation of the model dancer – short duration between notes means fast movement and more constrained motions. For the real player, color coding of the arrows is also a very useful visual cue for interpreting the rhythm of the song.

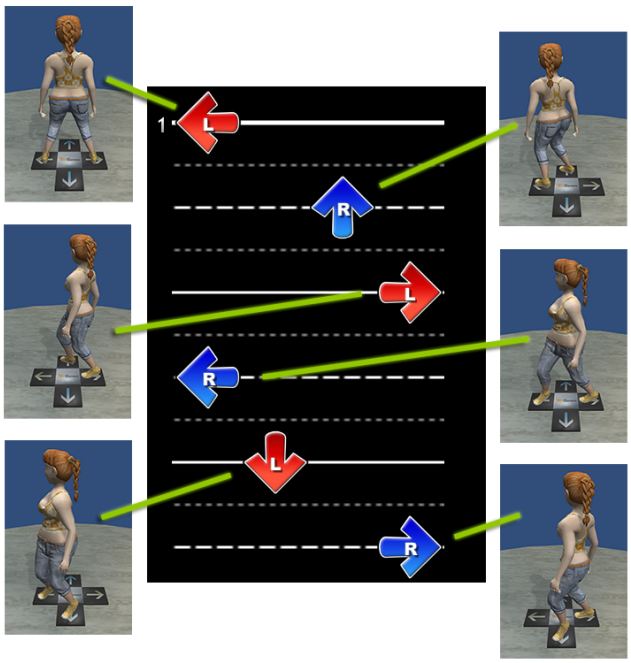


Figure 2.7: A 180° crossover pattern

2.4.3 More step patterns

To understand what requirements we have for the dance game AI and the model dancer, we will cover rest of the important step patterns that can occur in a stepchart. Consider the patterns in Figures 2.7 through 2.12:

Figure 2.7 contains a 180° crossover, that is, after the first four steps the left leg of the player is on the right panel, and the right leg is on the left panel. If we draw a line between the legs, normal of that line is at an 180° angle, if the zero angle is when the player faces the screen directly. In contrast, the regular crossover turns the player only to a 135° angle. Of course, the player tries to look at the screen all times, so the torso of the player is orientated at a lower angle and more towards the screen, than the feet normal.

In Figure 2.8 there is a pattern which turns, or spins, the player around a full circle. Compare this to pattern 2.7 – the first three steps are identical, but the fourth step turns the player from $-135°$ to $+135°$ angle (this is indicated with the green circular arrow symbol). In practice, spins are very distracting for the player, as they require a the player to turn his or her head over the 180° angle, during which the player cannot see the screen.

From the perspective of the step artist, that is, the person who creates stepcharts, spins should be used very sparingly. Moreover, notes of the spin

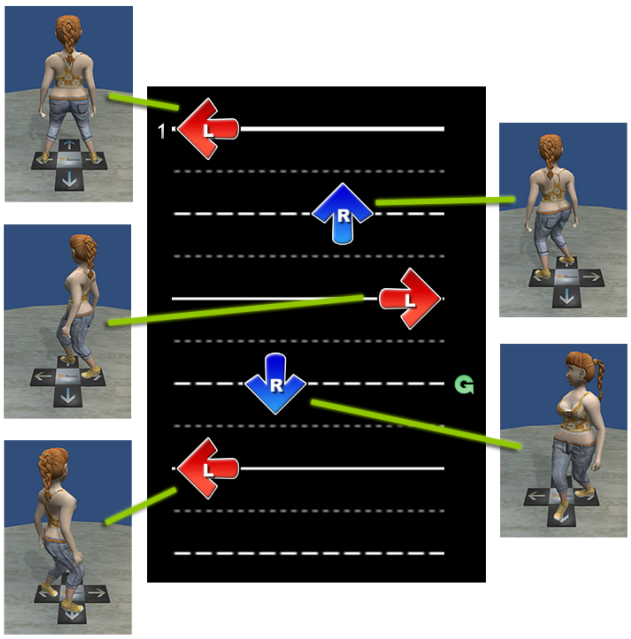


Figure 2.8: A spin pattern

should have enough time between them, so that the player can opt to doublestep the spin. The dance game AI should still play the spins using alternate stepping and actually spin, because this is technically the right way to play. And for the step artist it is useful to see if there are any spins in the stepchart – such patterns can be created accidentally with relative ease.

Skipping to Figure 2.10, this pattern has many taps on the same panel in a sequence. Such pattern is called a jackhammer, or a jack for short. This name originates from the repetitive motion when the player hits the same panel with the same leg multiple times. This is an important exception to the alternating stepping rule: if a leg rests on a certain panel, and the next note is for that panel, the note should again be tapped with that leg. Let us call this the *jack rule*.

The jack rule applies too if the next note is a jump, and one of the steps of the jump is for a panel on which a leg already rests. This may then result in a crossover, or even a spin. This is the case in Figure 2.9, where the third jump leads the player to a crossover and the fourth jump spins the player around. The final jump then orients the player to look back at the screen.

Again, in reality the player would tackle a pattern such as this by swapping the used legs in the second or third jump, disobeying the jack rule. But

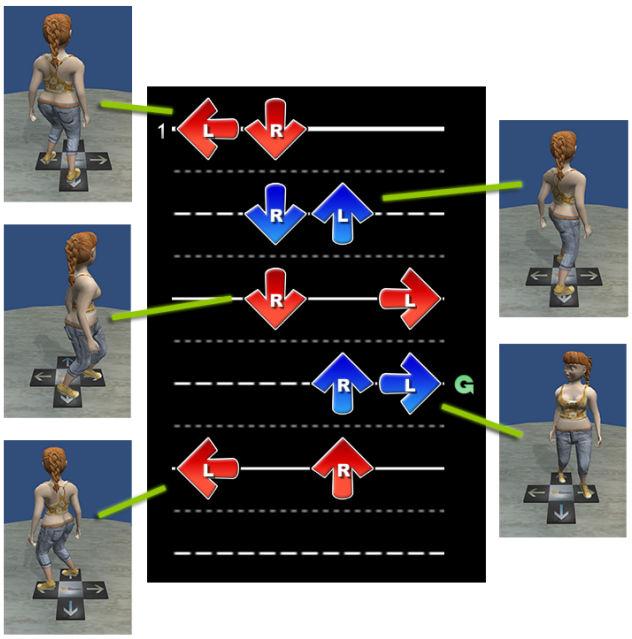


Figure 2.9: A jump spin pattern

as with the spin resulted from single steps, spins caused by jumps are hard to play, and should be discouraged in stepcharts too. For the step artist, they are visually even harder to spot than the regular spin, so it's helpful for the step editor software to outline them. No established term for breaking the jack rule in jumps exists, but let us call this a *jump footswitch*. Then it's the analogue for a doublestep in the case of avoiding a regular spin.

In Figure 2.11, there is a pattern that causes the right leg to move straight through the center panel of the dance pad. This is called a candle step, and occurs when a leg either moves between the ↑ and ↓ panels, or between the ← and → panels. This is noteworthy mostly because this is the longest distance to be traversed on a single 4-panel dance pad. And we can observe that in Figure 2.7 the left leg performs a candle step when it crosses over. But the pattern could be modified with extra steps so that no candle motions would occur. On the other hand, in the first three jumps of Figure 2.9, the player crosses over, but not through the center panel, so it's not a candle step. In the fourth jump however, the right leg moves from ↓ to ↑, so the spin portion of the pattern is a candle step.

In the real gameplay the candle step definition can be useful when the player is judging whether to avoid a spin or a crossover. If it contains a

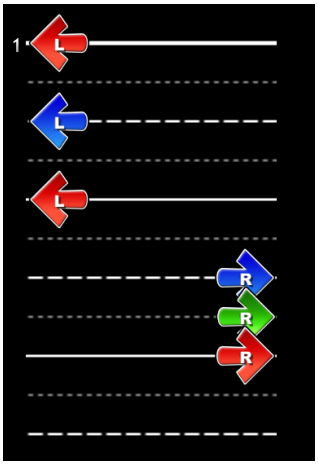


Figure 2.10: A jack pattern

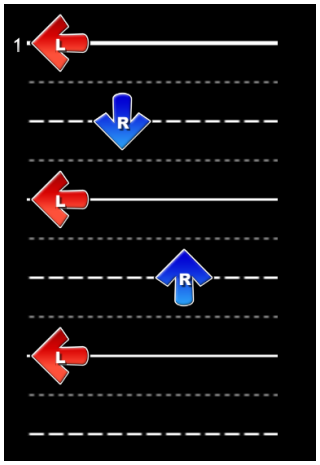


Figure 2.11: A candle pattern

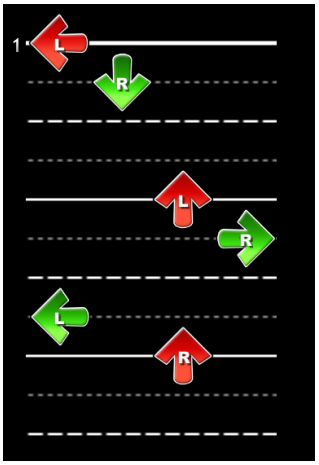


Figure 2.12: A gallop pattern

candle motion, it is harder than a pattern with shorter motions. Another parameter to judge the difficulty of the motion with is the angle difference of the foot normal between two steps. These can be employed in the AI system.

Finally in Figure 2.12 there is a pattern called the gallop. This refers to what the player does when playing the said pattern: he jumps and lands so that one leg hits the ground first and the other in a delayed fashion – galloping like a horse would, only with two legs. In the image, the player would jump and land on the first ← note on the left leg, then on the green 16th ↓ note with the right leg. Two subsequent pairs of 4th and 16th notes would be played in the same way. The important thing to note is that the pattern is a gallop pattern only due to how the notes are timed, otherwise it would be a regular *stream* of notes (an evenly spaced series of single tap notes is called a stream).

In the last section we said that the timing of the notes does not affect the way a particular pattern is played. And this is true if we only question which legs are used – in this case only the relative order of the steps is important. Due to this, it's not mandatory for the AI system to recognize a pattern as a gallop. But if the model dancer wants to play like a human would, at least in that end the gallop should be recognized and played with a jump motion instead of two steps.

2.5 Animation guidelines

John Lasseter has published a paper on how to apply principles of traditional animation to 3D computer animation [26]. We will go over these guidelines, so we can account for them in the design and implementation of the model dancer, and later evaluate how well we met the guidelines.

2.5.1 Squash and stretch

The mass and rigidity of an object can be communicated to the viewer by squashing or stretching it during its motion. In real life, any organic creature will squash or stretch when it moves, and only the most rigid objects (such as chairs or frying pans) do not. Animating a bouncing ball is a standard test for beginner animators: here we should squash the ball when it hits the ground, and stretch it when it bounces back up.

But when animating humans, for example, squash and stretch should not necessarily be applied as to deform the body. A jumping person would crouch (squash) in preparation of the jump, and straighten up his body when taking off, but this would be achieved just by rotating the limbs accordingly.

An important rule to squash and stretch is that if the object is deformed, its volume should always stay constant. For example, the bouncing ball would be squashed vertically and stretched horizontally when it hits the ground.

In addition to being a tell about the objects mass and rigidity, stretch can fix disturbing strobing effects that may occur with fast moving small objects, when the object would not overlap between consecutive frames in the animation. If the object is stretched so that it overlaps itself again, then the eye doesn't perceive the motion as separate images. A more realistic alternative solution to this is motion blur.

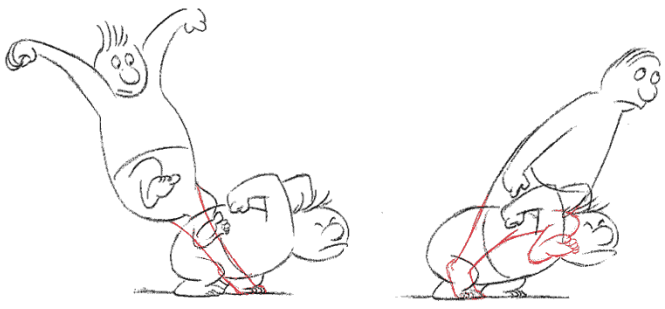


Figure 2.13: A jumping character exhibits squash and stretch [47]

2.5.2 Timing

Speed, or timing of the action, gives meaning to the movement. It communicates about the size and weight of the object, and may also convey emotion. Enough time should be spent that the audience is prepared for the anticipation of an action, the action itself and reaction to the action, but not any longer, so that the audience does not lose attention. A heavy object has a lot of inertia, so quite a bit of time should be spent in accelerating and decelerating it. Light objects accelerate almost instantly but the air resistance quickly slows them down (think an air balloon, for example).

Lasseter then cites Thomas and Johnston [37] for examples on how the timing of the action gives it different meanings. The examples are listed as a number of in-between frames that should be spent on different types of action. Lasseter does not explicitly mention what the frame rate is, but because this is from a book on traditional film animation, we assume it is 24 fps. The examples range from none to ten in-between frames. If the character is hit by a tremendous force and his head is nearly snapped off, this should have no in-between frames. At four in-between frames, the character could be giving a crisp order to somebody: "Get going! Move it!". At ten in-between frames, the character could be stretching a sore muscle.

At 24 fps, this gives us approximately the lower bound of 42 ms and upper bound of 420 ms for desired duration of a single motion.

2.5.3 Anticipation

An action occurs in three parts: anticipating the action, performing the action itself, and terminating the action. The anticipation or preparation of the action is discussed here. For organic or human action, the anticipation is necessary due to how the muscles work, that is, by contraction of the muscle tendons. For the muscle to contract, it has to be extended first. If a character is kicking a ball, he first has to bend the leg back and then execute the kick.

Another aspect of anticipation is that actions rarely consist of just a single movement of a hand or leg. Rather, if the character is about to grab something, he first looks at the object and then reaches and grabs it. This has the advantage that the audience is prepared for the action to happen. Also, if the actual action is going to be very fast, the anticipation can be made longer so that the audience is surely prepared for it. Conversely, slow actions may have just a minimal anticipation period.

2.5.4 Staging

Staging is the principle of underlying the main action that the audience should be following in a scene. Anticipation and timing of the action are very integral parts of staging. Contrasting the main action from the rest of the motion happening in a scene will help in underlying it as the principal point of interest. In a calm scene, fast and wide motions catch the audience attention. On the other hand, in a very busy scene, a stationary character will stand out.

It is also suggested that the camera should be oriented so that the action would be understandable even as a black and white silhouette. This idea was originated from Disney, where in the early days the animators had to work only with silhouette animation, and later found out this restriction results in better animation, even if the end result is in color [37].

2.5.5 Follow through and overlapping action

If anticipation is the act of preparing the action, follow through is the act of terminating it. Just as with anticipation, actions rarely terminate abruptly, and motion continues in some way past the termination of the action. For example, after throwing a ball, a character's hand continues past the point of release. The actions that follow should at first overlap the termination phase of the first action, so that the motion never stops completely. Related secondary movement should also follow the primary movement with a delay: for example, when walking, the hips lead the movement, then the legs follow, then the torso, shoulders and arms. With any action, eyes will generally lead.

Any appendages or loose parts of the character will move at a slower speed and drag behind the main motion. When the motion stops, the loose parts will continue moving for a while until they settle down.

2.5.6 Straight ahead action and pose-to-pose action

This refers to two possible methods of drawing the animation frames in traditional animation. Straight ahead means that the animator has planned what he wants the character to do, and then draws frame after next in a linear fashion. Pose-to-pose means that the animator draws the important key-frames first, and then begins to fill the gaps with in-between frames.

It is suggested, that for computer animation, neither of these techniques should be used. Instead, the animator should consider different layers of the animation separately, and animate them one at a time. For example, in a jump, first animate the movement of the torso and legs, perhaps first define

the start and end poses, then the apex of the jump. Then animate the arms, taking advantage of the local positioning as they are parented to the torso.

2.5.7 Slow in and out

Slow in and out means that the animation starts and ends with a slower velocity than in the middle. This is also called easing. In traditional animation the in-between frames must be spaced accordingly. In computer animation, we can interpolate between the start and end poses with various types of splines. Usually we want to achieve C_2 continuity for the motion when the spline segments are joined together. Sometimes we want the motion to end abruptly and break continuity, such as when a leg hits the ground. Then the parameters of the spline must be tweaked, requiring a type of spline that allows for this, such as the Kokhanek-Bartels spline (TCB spline) [22]. We introduce splines and types of curve continuity Chapter 4.

2.5.8 Arcs

The visual path of an action should always be an arc, for it is the most economical route to move from a point to another, Lasseter writes, citing [36]. Seeing as the hands and legs rotate around a point, which results in circular movement, this should be true for humans and for other creatures as well.

It is noted that you should have separate curves for the actual path of motion, and for the position in regard to time in the motion path curve. This way you can tune the speed at which the path is traversed, and have desired acceleration and deceleration when the motion starts and when it terminates. For the path of the action, a B-spline is appropriate for smoothness, whereas for the timing curve a Catmull-Rom spline [8] is a good candidate, because you can adjust its tension and smoothness to get the slow in and out. [5]

For the timing curve, though, the TCB spline should be appropriate as well, as it has similar properties and offers as much control as the Catmull-Rom spline.

2.5.9 Exaggeration

Exaggeration of the key ideas and actions in the story or the scene makes the animation more entertaining to the audience, and underlines the important elements. Properties that can be exaggerated include the design and shape of objects and characters, the action, the emotion, the color, and the sound. Exaggeration should be balanced, so that enough parts of the scene

are exaggerated and they don't stick out and seem unrealistic. However, not everything should be exaggerated, and especially elements that are based in nature should be realistic. This then sets the ground floor to compare against the exaggerated elements. Surprisingly, proper exaggeration of scene can make it seem more realistic, while also making it entertaining.

2.5.10 Secondary action

An action that directly results from the main action is a secondary action. The loose parts discussed earlier are one type of secondary action. A facial expression during the movement can also be a secondary action. The facial expression should not change during the move, not because it would risk the facial animation being promoted to the primary action, but because the change in expression would likely go unnoticed. Therefore, the change must come before or after the movement. [37]

2.5.11 Appeal

Appeal is anything that a person likes to see, such as a pleasing design or simplicity. A complicated or hard to read design or a clumsy shape lacks appeal, and so does a character with awkward movement. [37]

When trying to create an appealing pose for the character, you should avoid the "twins" pose, where both arms and legs are in the same position, doing the same thing. If there is some discrepancy between the movement of the left side and right side of the body, it will be seen as more natural and appealing. This applies to facial animation as well, one side of the face should not mirror the another.

Chapter 3

Designing the dance game AI

In this chapter we will describe the design of our artificial intelligence (AI) for 4-panel dance games. Based on the plays generated by the AI, our dancer character will then dance stepcharts of the songs.

The desired output of the AI is a dictionary structure, which for every note row of a stepchart defines the state of the player at the time of the note row. This state will then tell on which dance pad panels the player is standing at that time.

The way to play the stepcharts that the AI should follow is described in Section 2.4. For single notes, we should primarily play them with alternated stepping, unless a leg already is on a panel which contains the note – then that leg should be used (the jack rule). For jumps, if no leg is on the panel belonging to the jump, we can freely choose which leg lands on which panel. If either or both of the panels have legs, those legs should again press those panels, and the way to play is unambiguous. Otherwise, the leg placement of the jump should be chosen so that rest of the stepchart is easier to play.

How can we then compare which way of play is easier? As a general rule, the more we can keep our orientation towards the screen, the easier it is to play. Specifically, playing patterns as crossovers or spins is definitely harder to perform than straight up play facing the screen. In crossovers you have to turn your body away from the screen to facilitate the motion of the left leg stepping on the right panel or vice versa. The shifting of body momentum from the usual positions requires extra effort from the player. With spins the caveats are same, but you have to turn your body even more, and momentarily not see the screen.

Depending on the leg on which we start to step the pattern with, it can sometimes be a crossover or a spin when started with one leg, and can be just played straight up when started with the another leg (see Figure 3.1 for such a pattern; detailed explanation of the image will follow in the next

section). In these cases it is important to find which is the better leg to start the pattern with. Generalizing this idea, we want to find such play of the whole chart that it contains the least possible amount of undesired moves and positions, such as crossovers or spins.

A fast way to find a good-enough play would be to use some look-ahead when scanning the chart. When we hit a branch point, such as a jump, we could scan 10 or so steps ahead, and determine which way of play results in more comfortable patterns. Of course, if we hit another branch point in the look-ahead window, we would have to either stop there, or branch again and look through all the $2 * 2 = 4$ possible ways to play. This method would actually simulate the way a real player plays, because he can after all look at only a handful of the following notes when determining which path to take.

But because the AI is also a tool for the stepartist creating new steps for songs, we want to display the play taking absolutely the least effort, and the stepartist can then evaluate whether that play is the intended play for patterns he created. To find this optimal play, we have to find the lowest cost path of all the paths that can be taken in the decision tree of the stepchart. Let us then consider how the decision tree is built; this is the core of the AI system.

3.1 Building the decision tree

The decision tree of the stepchart is a tree that contains every path in which the stepchart can be played. The nodes of the tree are the states (positions on the dance pad) where the dancer is currently at. All the nodes at a specific depth in the tree are possible states for the same note row. Edges are moves the player does to transition between the current state and the next. Strictly speaking the decision tree is actually a directed graph, because in many situations we can move into the same state from two different states. This reduces the number of total paths in the structure quite heavily. We shall refer to it as the decision tree despite this merging of paths, because that is the established term for such structure, and could be implemented as a tree with equivalent behavior, only less efficiently.

An example decision tree is shown in Figure 3.1. Here we can start to play the pattern with either leg, but the path in the right branch leads to a spin. The diagonal edges from states in the third row to states in the fourth row are jump footswitches, and therefore expensive. The green path on the left has least cost, and is the optimal way to play.

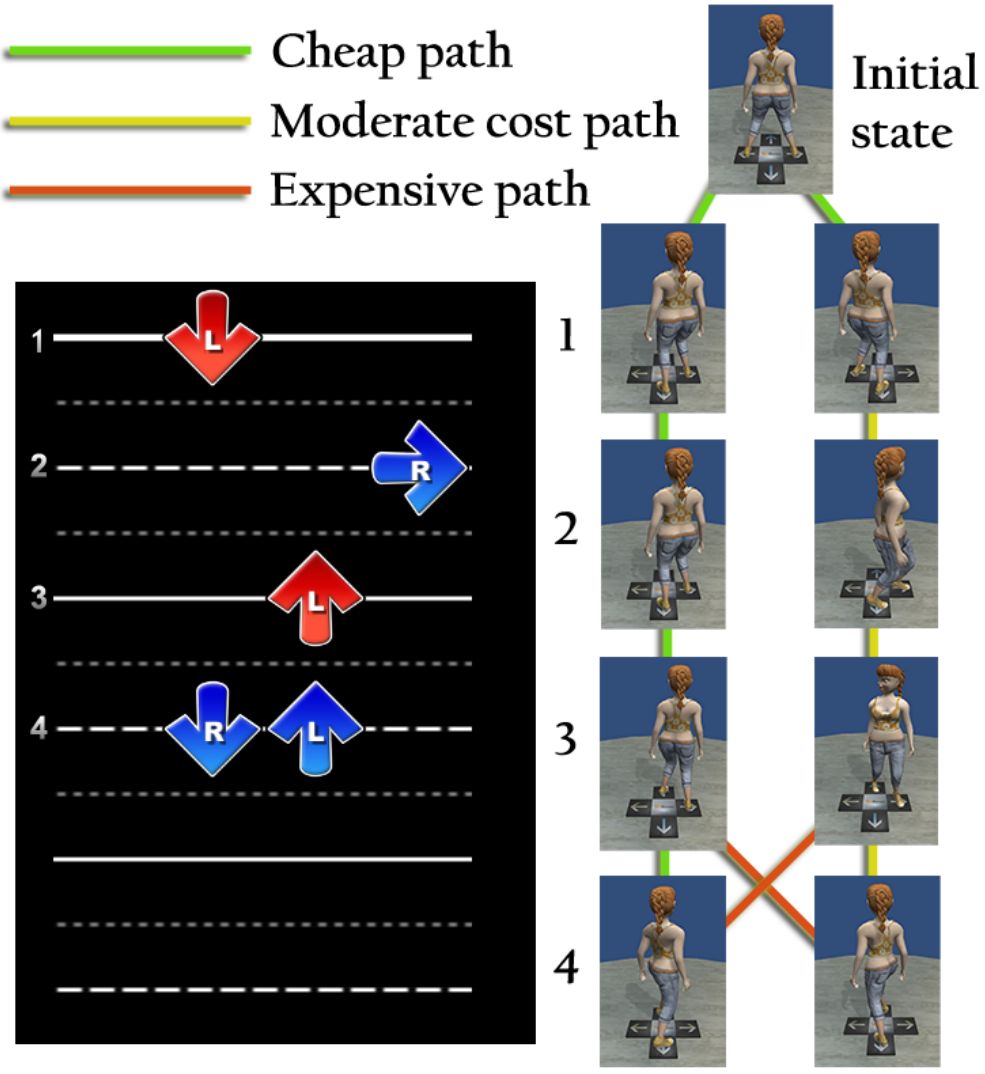


Figure 3.1: Example decision tree where states are visualized using the dancer. Doublesteps are disallowed (steps are not branch points), but air footswitches are allowed (the jump is a branch point).

3.1.1 Defining state properties

For a state in the tree, we maintain the following information. A state which contains the same information will be considered as an equivalent state. The properties, which concern the current note row, are:

- **Limb placement** on the dance pad panels. Minimally it describes where the feet are situated, but for notes tapped with hands also the hand placement is defined. The limb placement will be the primary output of the AI.
- **Free limbs**, the limbs that are not pressing a hold note. A reserved limb cannot be used for stepping, so we need to track these.

Then there are properties that result in from the previous state, and the move used to reach this state. One of them is:

- **Last leg** we stepped with. This is used to drive alternated stepping for single taps.

The rest of the properties that result in from the previous state are used to assign relative cost to when this state is visited (this will be explained in Section 3.1.3):

- **Amount of moved legs**: how many legs (0-2) we moved to different panels to reach this state.
- **Angle delta**: the change in the angle of the dancer from the last state. The dancer's angle is considered to be the angle of the feet normal vector (the normal vector of the vector between the feet).
- **Resulted by a doublestep?** If we stepped with the leg that we also stepped previously with (the last leg), the move is a doublestep, and this state will be marked to be resulted by the doublestep.
- **Resulted by a spin?** If we turned past the 180 degree rotation (where the 0 degree rotation is directly facing at the screen), we spun over, and will mark this state to be resulted by the spin.
- **Resulted by a jump footswitch?** If a leg was situated at a panel belonging to this jump, and in the jump move we landed on the another panel of the jump with this leg, breaking the jack rule, the move is a jump footswitch. We will then mark the state as being resulted by the jump footswitch.

3.1.2 Branching on different ways to play

To build the decision tree, we now iterate over all the note rows in the stepchart. For a note row, we list the possible ways it can be played as different states. For subsequent note rows, we list all the different ways it can be played when arrived from any state in the previous note row, and connect the previous and next states with edges.

If the note row contains a single step, we step with a single leg, branching into different paths as follows:

1. If a leg is on the panel, the only option will be to use this leg (obey the jack rule).

2. If either leg is reserved for a pressing a hold note, we use the other leg that is free. If both legs are reserved, we have to use a hand.

3. If we don't allow doublestepping (this is a configuration parameter to the AI), and the last used leg is defined, we step on the other leg (obey the alternated stepping rule). If the last used leg is undefined, we branch so that either leg can be used with equal cost. The last used leg will be undefined at the beginning of the stepchart, and after a jump, which is considered a reset point for alternated stepping.

4. If we do allow doublestepping, we branch so that either leg can be used. If the last used leg is defined, choosing the doublestep route will be penalized with higher cost than the alternated stepping route.

If the note row contains multiple steps (2 to 4), we jump on the note row panels (unless both legs are reserved). Any panels not pressed with legs are tapped with hands. More accurate description is as follows: We consider the number of notes in the note row, and form all possible permutations of limbs that can be used to play the note row, but in such way that any free legs are always included in the permutation, only after then possible hands are filled to rest of the panel slots.

For the common case of two steps, where the both legs of the player are free to use, we obviously branch into two paths depending on which leg steps on which panel. But the upper bound for the number of branches we can take is considerably higher. The upper bound occurs when we have notes on all four panels and all limbs available. We choose the panel of the left leg from 4 options, panel of the right leg from 3 remaining options, panel of the left hand from 2 remaining options, and assign the right hand to the remaining panel. This yields a total of 4*3*2 = 24 options and possible paths. In our system these paths converge fast to fewer paths in the next states, so this degree of (short) branching is not a concern.

3.1.3 Assigning cost to states

When a new state has been generated to the tree, we will compute the properties of that state (described in Section 3.1.1), updating the limb placements and the last used leg, checking if the state is a doublestep, reserving limbs that pressed a hold note, and so on. When these properties have been determined, we assign a cost to this state, which describes the relative difficulty of playing the move that results in this state. So the higher the cost of a state is, the more undesirable it is to play in such way.

We will now describe our cost function. The general idea is that alternated stepping should be preferred, but if playing in this fashion forces the player to play long sections of the stepchart as crossover patterns, or multiple spins, even, we want to doublestep to avoid this. If the foot normal angle of the player is 135 degrees or 180 degrees, the player is performing a crossover move. For every state when the player is in this orientation, we assign +1 cost. Now the important question is, how many crossover states should be tolerated before they should be skipped with the doublestep.

Based on extensive testing with different stepcharts, we assign +12 cost for the doublestep. Therefore, a doublestep may be done if it avoids at least 12 steps in the crossover orientation.

The phase of alternated stepping may also be changed, if there is a jump before the pattern in question, which shares a same panel with the beginning of the pattern. Then, we start the step pattern depending on the way the jump is played. If, furthermore, a leg is already on one panel belonging to the jump, the playing the jump such that the leg moves is considered a jump footswitch. The jump footswitch can then be used to avoid to play the pattern in crossover orientation. In general, doing jump footswitches is more common in real play than doing doublesteps, so we assign the cost of +10 for a jump footswitch, preferring it slightly over the doublestep.

We also assign some cost for when the player spins, that is, turns over the 180 rotation. We build this cost function considering first and foremost which is the "correct" way to play, in which the spins resulted by alternated stepping are a part of. This makes this tuning most useful for the stepartist, reviewing the indented way to play for a stepchart. Therefore we assign a low cost of +3 for the spin. It is now preferred over the doublestep and the air footswitch, but some cost has to be assigned. Then the AI will still try to avoid starting a pattern so that it leads to a spin, if multiple ways to play with equal cost are available.

Finally, we assign a costs for additional things that should be discouraged:

- +5 cost if the delta angle is 135. Such change in angle is quite severe and should be avoided.

- +15 cost if the delta angle is 180. This is the largest possible change in angle, and almost never justified.
- Various costs for using hands to tap on panels: +1 for every state where a hand is pressing a hold, +5 if an arm is extended to a panel that is not in front of it, and from 0 to +2 cost depending on the orientation of the legs when a hand is used (if you bend backwards when tapping with a hand, this assigns the +2 cost).

3.2 Computing the optimal play

After the decision tree has been built and costs of the states computed, it only remains to find the path with least total cost from the tree. We can find this path using the following algorithm with O(n) time complexity, where n is the number of states in the graph:

1. Starting from the next to last tree row, for every state in the row, mark the lowest possible total cost to traverse from this state to the last row. In other words, you sum the cost of this state and the cost of the cheapest state in the last row that is connected to this state. Then we move one row up, and for every state, sum cost of the state and cost of the connected row with the lowest possible total cost to the last row, which we previously computed. We continue computing the lowest possible costs until we reach the initial state.
2. Now we can find the cheapest route by traversing through the tree, such that we always select the state, which has the lowest possible total cost to traverse to the last row.

3.3 Extra considerations

3.3.1 Performance

We have measured the performance of our AI implementation with tests that run the algorithm for every (single play mode) stepchart in our game. For total of 311 stepcharts, running the AI took 3056 ms. On average, a single stepchart was processed in 10 milliseconds. The test was run on a fairly powerful Intel Core i7-2600k CPU at 3.4 GHz clock frequency. Nevertheless, this means that it is quite feasible to run the AI algorithm every time in a stepchart editor software when the user makes changes to the stepchart.

There is also no need to pre-compute the plays of the charts when they are used as input for the dancer, and the dancer is animated during gameplay. Computing them on demand does not add much to the loading times of the songs.

3.3.2 Handling mines

The description of how the AI should avoid mines was omitted from Section 3.1. We consider mines at a pre-processing step for a state. Before the follower states for a state are generated, we make several changes to the state, and this changed state is then considered as the previous state for the follower states. We release limbs pressing holds that end in this state (this is because the end of the hold can be considered an open interval, and you can release it a bit before it actually ends in gameplay).

And then in the pre-process step, we avoid mines. If a leg is situated at a panel in the previous state that contains a mine in the current state, we will move that leg into the center of the dance pad. The order of operation where we release the mine at the pre-process step is useful, because like the hold, the reaction window to the mine is an open interval in regards to where the mines is. In other words, you have to lift your foot from the mine panel before the mine actually passes the target arrows.

This handling of mines, where we just avoid the mine into the center panel, is a bit simplistic, though. In real play, we often step over a mine to a next tappable note (see Figure 3.2 for a trivial example pattern). For the following patterns, always evading a mine into the center panel still results in the same play, but for the particular move it will look unrealistic, if the dancer is controlled literally to step on the center panel. Because of these problems, the mine avoidance animation is completely omitted from the dancer side in the current implementation, even though we handle it in the AI.

3.3.3 Handling roll notes

The roll is a note type similar to the hold, but it has to be repeatedly tapped in order to keep it held. In the AI, we can deal with rolls in two ways. The simpler way is to treat them as holds, and just ignore the fact that the player taps on them when they are held. The first problem with this implementation is that we then have no information for the dancer on when the roll should be tapped on. This could be computed at the dancer side, though. But the real problem is that some stepcharts mix tap notes beside rolls, and then we actually want to play the resulting pattern using the alternated stepping rule.

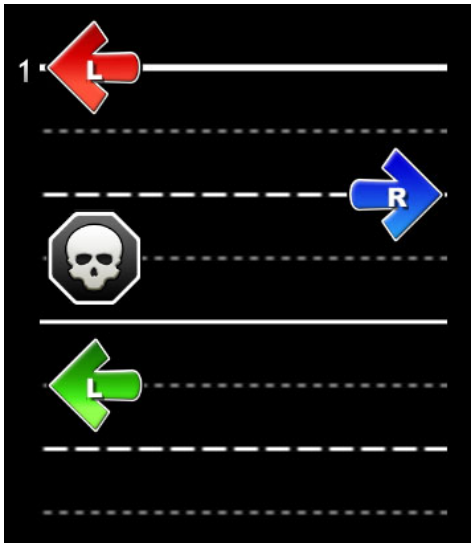


Figure 3.2: A pattern in which the player avoids the mine by stepping over it, rather than steps explicitly on the center panel to avoid the mine.

The better solution is to transform all the rolls in the chart into virtual tap notes. We write a decorator class [15] for the stepchart class that shows the roll notes as taps. Distance between the taps will be the longest possible note length that does not drop the roll. We pass this decorated chart as input into the AI, and it will play the rolls in the desired way without changes to the AI implementation.

3.3.4 Extension to the double play mode

Nothing in the proposed AI implementation actually precludes its use in the double game mode, where the player plays on two adjacent dance pads. The cost function has to be tuned to assign costs for the additional angles occurring in doubles play. Otherwise the implementation works fine, and our AI supports doubles play at this time. The dancer animation, however, does not, but this is mostly because it was not in the scope of this thesis. The current implementation of the dancer, described in Chapter 5, should be able to adjusted to play double mode stepcharts also.

Chapter 4

Computer animation techniques

In this chapter, we explain fundamental concepts of computer animation, that are prerequisites for understanding our solution to the problem of animating our dancer character. We will also introduce some advanced techniques that can be used to synthesize animation, or improve the quality of existing animation. Let us start by introducing *skeletal animation*, which is a popular technique of animating humanoid characters.

4.1 Computer animation fundamentals

4.1.1 Skeletal animation

The most basic way to animate any virtual object would be to just alter its position, rotation and scale over time. If the initial and target values of one of those parameters are known, we can interpolate the value over time and produce animation. If the object in question is completely rigid, such as an aeroplane or a robot, we can model its movable or rotatable parts separately, and animate them this way.

However, if we want to animate living creatures, more complex methods have to be used. Consider modeling a human, such that for every bone, the part of the body that encompasses the bone would be a separate object. Then we could combine these objects to a hierarchy and rotate them to produce animation. This would work at the basic level, but it would look relatively poor for several reasons. First, the ends of the limbs would intersect when rotated, or leave open gaps. Second, skin of a living creature does not deform in a clear-cut way, but rather in an organic and elastic fashion. We have to use a method which deforms a body made of a single mesh, and lets the skin near a joint of the creature to be less affected by orientation of the creature's

(a) (b)

Figure 4.1: An example of a human skeleton rig: our dancer character, Elizabet. Image (a) displays the bone hierarchy, and image (b) displays bone weights that are set up for her clothing for the selected bone, outlined in cyan. Red color in the weights' heat map represents areas where the bone has full influence, blue represents no influence.

bone than in middle of the bone.

The widely used method to solve is called just *skeletal animation*, or bones and weights animation. In this method the character is modeled as a single mesh, and is linked to a support structure, the character's skeleton (see Figure 4.1(a)). More specifically, this is a hierarchy of bones, which are transforms that contain a position and rotation. Bones that are supposed to be moved by other bones are their children in the transform hierarchy.

Bones also contain a map of weights per vertex of the character mesh (see Figure 4.1(b)). A weight value from 0 to 1 represents the strength with which the rotation of the bone will transform the vertex. With the value 0 the vertex is unaffected by the bone, and with the value 1 the vertex will be fully deformed. The process of assigning the bone weights is commonly known as

weight painting. This is because the bone weights are usually represented as a heat map on top of the character mesh, in 3D modeling programs such as Blender. The heat map can be then modified visually with brush-like tools. When combined with weight painting, the skeleton creation process is also commonly known as rigging the model. [20]

Skeletal animation is widely supported in 3D and game engines including Unity3D [40], Unreal Engine [12], Source [41], Cryengine [9], Ogre3D [38] and so on. Many 3D model formats can host this kind of animation, including FBX [4] and Collada [21]. Skeletal animation can be also baked to vertex animation, meaning that the location of every vertex per keyframe is stored. Then it can be used with older formats such as MD3 [19], and older engines.

4.1.2 Forward and inverse kinematics

Forward kinematics and *inverse kinematics* are the two main ways of assigning certain rotation configuration to a transform hierarchy, such as the character skeleton. Forward kinematics is very straightforward. We directly assign rotations to the nodes in the hierarchy, and the child nodes are rotated by combination of the parent rotation, and their own local rotation.

The problem with controlling the character like this is that it's inconvenient for creating animation. Imagine trying to animate a walk cycle. Over time, we would have certain points where the character's feet and hands should be. To reach these points, we would manually rotate the thigh and knee joints, and the shoulder and elbow joints. To hit the point exactly, we would need to actually calculate the angles based on the goal position, using some trigonometry.

This is actually one form of inverse kinematics. Inverse kinematics, or

Figure 4.2: Example use of IK: a target position for this bone chain is set, and the rotations of the joints are solved automatically [25]

IK, is the process of automatically computing the rotations for a system of linked joints, given a desired goal position for the last joint in the chain [7]. In skeletal animation, the joint hierarchy is the skeleton. Using IK, we could for instance set the goal position of the character's left foot, and let rotations of the knee and thigh to be computed with IK. The tip of the bone for which the goal position was set is known as an IK effector, and the algorithm which determines the parent rotations is known as an IK solver. The set of produced rotations is called the IK solution. [30]

Also, in addition to a position goal, the IK solver may support setting a certain rotation goal for the IK effector. Rotations of the parent bones are then additionally constrained by this rotation goal. Combined with constraints for the joints, such as the knee being a hinge joint (only bending around one axis), realistic rotations for human bones can be produced.

There are several algorithms for IK, and we will discuss them next. The choice of the algorithm first and foremost depends on how many bones in the hierarchy must be controlled, and how many degrees of freedom do the joints of the bones have.

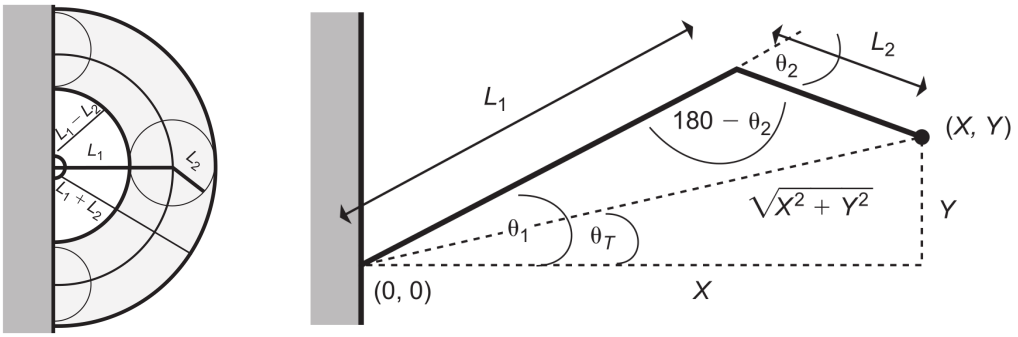
4.1.3 Two-bone analytic IK

For the chain of only two bones that bend around one axis, such as a leg, the IK solution can be computed analytically [30]. To demonstrate that the two-bone IK solution is straightforward, let us walk through the formulation of the solution.

We will denote the lengths of the two bones in the IK chain with L_1 and L_2, and their joint angles with θ_1 and θ_2, respectively. First we have to make sure that the target position can be reached at all. Consider Figure 4.3, part (a).

When both bones are fully stretched out, this gives us the maximum total length of $L_1 + L_2$. When the bones face in the opposite direction, this gives us the minimum total length of $|L_1 - L_2|$. Furthermore, assume that the target point is (X, Y) and the starting point of the first bone is $(0, 0)$. Then distance between the points is $\sqrt{X^2 + Y^2}$. Now the target point can be reached if:

$$|L_1 - L_2| \leq \sqrt{X^2 + Y^2} \leq L_1 + L_2 \tag{4.1}$$



(a) Minimum and maximum reach

(b) Analytic solution

Figure 4.3: Two-bone IK [30]

If the target point can be reached, we can compute the bone angles. Consider Figure 4.3, part (b). Using the definition of tan(), we compute the angle θ_T:

$$\tan\theta_T = Y/X \tag{4.2}$$

$$\theta_T = \arctan(Y/X) \tag{4.3}$$

Then we use the law of cosines, which states that for any triangle with sides a, b, c and angle γ between sides a and b (facing the side c):

$$c^2 = a^2 + b^2 - 2ab\cos\gamma \tag{4.4}$$

$$\cos\gamma = \frac{a^2 + b^2 - c^2}{2ab} \tag{4.5}$$

Applying formula 4.5 to the upper triangle, we compute the first angle:

$$\cos(\theta_1 - \theta_T) = \frac{(L_1)^2 + (\sqrt{X^2+Y^2})^2 - (L_2)^2}{2L_1\sqrt{X^2+Y^2}} \tag{4.6}$$

$$\theta_1 = \arccos\left(\frac{(L_1)^2 + X^2 + Y^2 - (L_2)^2}{2L_1\sqrt{X^2+Y^2}}\right) + \arctan(\frac{Y}{X}) \tag{4.7}$$

Then we apply formula 4.5 again to compute the second angle:

$$\cos(180^\circ - \theta_2) = -\cos\theta_2 = \frac{(L_1)^2 + (L_2)^2 - (\sqrt{X^2+Y^2})^2}{2L_1L_2} \tag{4.8}$$

$$\theta_2 = \arccos\left(\frac{(L_1)^2 + (L_2)^2 - X^2 - Y^2}{2L_1L_2}\right) \tag{4.9}$$

4.1.4 Full-body iterative IK

If the IK chain we want to pose is longer than two bones, or if the joints have more degrees of freedom than one, the analytic solution described above cannot be used. Some implementations exist that compute closed-form solutions for two bones with more degrees of freedom [11], but at least for longer chains iterative numerical solutions have to be used.

For the needs of human IK, we hit these limitations if we need the character to able to extend the legs or hands further that they can reach, and therefore be able pull the torso towards the leg or hand (full body IK).

Solutions to the full body IK problem include the Jacobian inverse method, the CCD method (Cyclic-Coordinate Descent) and a method called FABRIK (Forward And Backward Reaching Inverse Kinematics). We will go over the ideas behind these methods.

4.1.4.1 Jacobian inverse method

The Jacobian inverse method uses the *Jacobian matrix*, which is defined as the matrix of all first-order partial derivatives of a vector-valued function. To solve the inverse kinematics problem using the Jacobian, we first define θ as the vector of the current joint angles of the bone chain, $\dot{\theta}$ as the current velocities of those joint angles, and V as a vector of velocities of the end effector at the tip of the bone chain (containing both linear and angular velocities). [30]

$$\theta = [\theta_1, \theta_2, \theta_3, ..., \theta_n]^T \tag{4.10}$$

$$\dot{\theta} = [\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, ..., \dot{\theta}_n]^T \tag{4.11}$$

$$V = [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]^T \tag{4.12}$$

We set V to such velocity that it takes the end effector from its current position to the target position. Now our problem is to find the respective joint velocities $\dot{\theta}$ to move the end effector according to the velocity vector V. Using the Jacobian, we can relate the two velocities:

$$V = J(\theta)\dot{\theta} \tag{4.13}$$

From the definition of the Jacobian matrix, J(θ) is then:

$$J(\theta) = \begin{bmatrix} \frac{\partial v_x}{\partial \theta_1} & \frac{\partial v_x}{\partial \theta_2} & \cdots & \frac{\partial v_x}{\partial \theta_n} \\ \frac{\partial v_y}{\partial \theta_1} & \frac{\partial v_y}{\partial \theta_2} & \cdots & \frac{\partial v_y}{\partial \theta_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial \omega_z}{\partial \theta_1} & \frac{\partial \omega_z}{\partial \theta_2} & \cdots & \frac{\partial \omega_z}{\partial \theta_n} \end{bmatrix} \tag{4.14}$$

Our solution then requires us to find $\dot{\theta}$. If we multiply both sides of Equation 4.13 by $J(\theta)^{-}1$, we get:

$$\dot{\theta} = J^{-1}(\theta)V \tag{4.15}$$

Which is the solution. However, this requires us to invert the Jacobian matrix, and this can only be done if it is a square matrix. For most systems it will not be square, so we have to use alternative methods. One is the *pseudo-inverse* $J^{+}(\theta)$ of the matrix:

$$\dot{\theta} = J^{+}(\theta)V \tag{4.16}$$

$$J^{+} = (J^{T}J)^{-1}J^{T} = J^{T}(JJ^{T})^{-1} \tag{4.17}$$

This yields an approximation of the inverse. A simpler method is to straight up replace the inverse with the transpose J^{T}, but this will result in a lower-quality approximation [7]. Other possible methods for computing the pseudo-inverse are the Damped Least Squares method and the Singular Value Decomposition method [7].

An important fact is that the inverse Jacobian only provides the joint velocities for that specific instant of time. This means that we won't actually reach the goal position of the end effector by a single big change to the joint angles, but we actually have to perform multiple small rotations to converge to the goal position. After each rotation, the Jacobian matrix has to be recalculated to represent the new velocities. This makes the method iterative in nature. [30]

The Jacobian inverse method in this form doesn't constrain or prefer any joint angles over others, so for the human skeleton it may result in unnatural poses. It is possible, though, to augment the method with a control expression that biases the solution towards specific joint angles, and fixes this problem. [30] Style-based IK [16], discussed in Section 4.2.2, is one example of such control expression.

4.1.4.2 CCD method

The CCD (Cyclic-Coordinate Descent) method [43] is quite intuitive to understand geometrically. In this method, each joint is rotated towards the target position, one joint at a time. The CCD method begins from the tip of the bone chain, and computes the vector P_{1c} between the tip of the bone chain and the first joint starting from tip of the chain. Then the vector P_{1d} is computed, between the first joint and IK target position. The joint is then rotated such that P_{1c} and P_{1d} overlap. [44]

CCD then continues to rotate the joints further up in the same fashion. Vector P_{2c} is computed between tip of the chain and the second joint, and P_{2d} between the second joint and the IK target. Figure 4.4 shows the iteration step of CCD for the i:th joint, where P_{ic} is the vector between the i:th joint and the tip of the chain, and P_{1d} is the vector between the i:th joint and the IK target position. When the last joint in the chain is rotated, the algorithm starts from the tip bone again. The iteration terminates when the distance between the IK target and the tip bone is shorter than an acceptable value of error. [44]

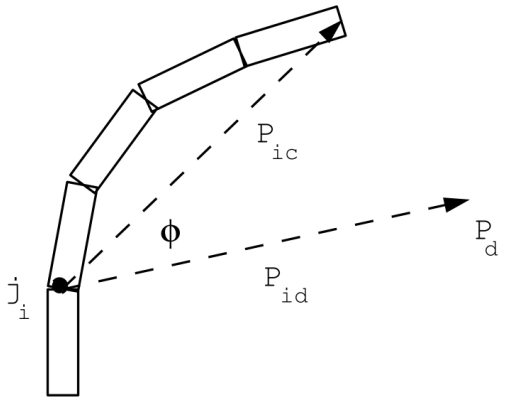


Figure 4.4: One iteration step of CCD [44]

4.1.4.3 FABRIK method

The FABRIK (Forward And Backward Reaching Inverse Kinematics) method [3] is similar to CCD in that one joint is considered at a time and rotated so that the position of the bone chain will converge towards the IK target position.

Figure 4.5 shows the complete iteration of the algorithm. We start from the tip of the bone chain (here denoted with p_4), as in CCD, but instead of rotating the bone segment towards the IK target, we temporarily move and rotate the bone so that it is both connected to the IK target, and lies in the line between the IK target and the joint p_3. This is shown in Images a–c. To reconnect the previous bone in the chain to the tip bone, it is also moved and rotated so that it lies in the line between p'_3 and p_2. This is repeated for all the bones until we reach the root joint, causing the root joint to be dislocated from its original position (see Image d). [3]

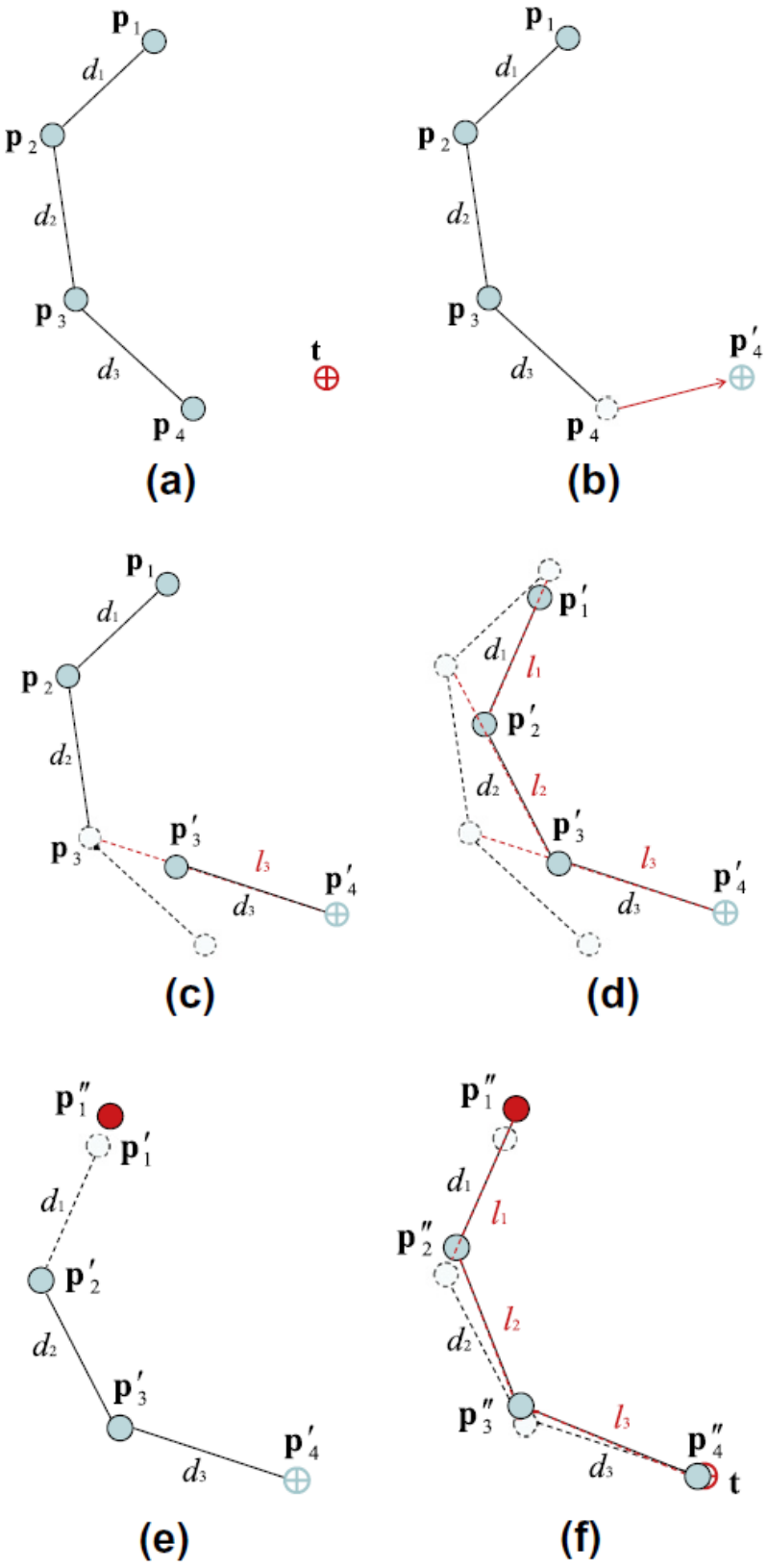


Figure 4.5: Complete iteration of FABRIK [3]

Then in Image e we repeat the algorithm, but in the opposite direction. We move the root bone in the chain so that it is again connected to its original position. We move the next bone so it is connected to the root bone. This is repeated until we reach the last bone in the chain, and move it so that the tip of the bone is not exactly in the IK target point, but should be closer than in the initial configuration of the chain (see Image f). The algorithm is repeated until the tip of the end bone is sufficiently close to the IK effector. Compared to CCD, the FABRIK iteration seems converge to the solution in fewer steps. [3]

4.1.4.4 Comparison of methods

The paper by Aristidou that proposes the FABRIK IK method [3] contains a comparison between FABRIK, CCD and the Jacobian inverse methods. Figure 4.6 contains a comparison of how different methods converge to the target. In the charts, remaining distance to the target is plotted against the number of performed iterations. We can see that here FABRIK outperforms the other algorithms in efficiency, though the difference to CCD is negligible. Jacobian methods (using various ways to invert the matrix) use approximately 40 times more iterations to reach similar results.

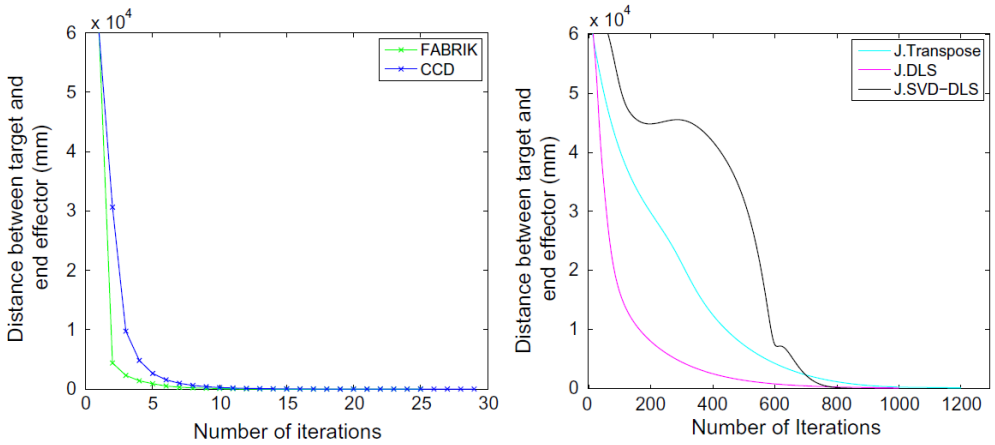


Figure 4.6: Different IK methods converging to the target [3]

Run on a Pentium 2 Duo 2.2 GHz, the author claims that the average time per iteration step of FABRIK is 0.86 ms, while it is 4.69 ms for CCD. The durations of the full iteration are 13 ms for FABRIK, 124 ms for CCD and around 10 seconds for the Jacobian methods, so overall FABRIK is clearly the fastest, and Jacobian methods are definitely too slow for real-time simulation.

As for the quality of the reached postures, FABRIK and Jacobian methods both seem to produce smooth and natural arcs, but CCD can produce unrealistic shapes [3] . It is mentioned that the Jacobian method works very naturally with systems with multiple end effectors, such as in the full-body human IK setting. FABRIK can seemingly be applied to this problem as well, as there is a commercial full-body IK implementation based on it available [25], so it seems well-suited for purposes of real-time IK.

4.1.5 Keyframe animation and motion capture

Let us now discuss about how one would actually produce animation. Using the skeletal rig rotated with forward or inverse kinematics, an artist can produce hand-crafted animation by posing the character in the key points of the desired movement. The animation is quantized into frames at a certain rate, and the frames that have a pose set are called key frames. The final animation is then produced by interpolating between the poses defined in the key frames. Different methods of interpolation can be used, from simple linear interpolation to fitting a spline (e.g. Catmull-Rom) that goes through the key points. The latter has the advantage that the movement velocity is continuous (this will be discussed further in Section 4.1.7).

Keyframe animation can also be produced with motion capture techniques. This means that the movement of a human actor is captured with special equipment. Popular way to do this is to have the actor wear a suit containing special markers, capture a video of the motion, and then use image recognition techniques to determine the positions of the markers. A skeleton of a virtual character can be then rotated to match the marker positions.

4.1.6 Procedural animation

For our dancer character, we want to produce animation that matches the desired way to play an arbitrary dance game stepchart. Thus we cannot, for example, play a complete pre-made animation for every stepchart. For individual moves, though, we can employ animation clips. The problem then becomes how to combine the clips for continuous motion.

Procedural animation is a type of animation controlled in some way by a computational model [30]. Figure 4.7 shows one example of procedural animation in the growth of foliage. In our system, we want to produce animation using some kind of procedural animation model to be able to play arbitrary content. We could combine animation clips according to some heuristic. This is feasible if the number of the required clips is manageable. This will be analyzed later in Section 5.1. Alternatively, we could compute

the trajectories for the character's points of interest, such as feet, body, and hands, and drive them with real-time IK control.

While passable motion with IK is easy, synthesizing realistic motion is difficult due to number of subtle motions present in actual human movements. Often inverse kinematics is rather used to guide (re-target) existing animations to reach procedural goals. For example, a character may have a animation clip for grabbing objects in the scene. To account for the position of the target object, and the character, the grab animation can be blended from the animation to an IK pose, where the character is grabbing the object. Another example for the use of IK is re-targeting the position of the character's feet to the ground surface, when walking on an uneven terrain.

Figure 4.7: Fractal weeds generated using an L-system [46] are one example of procedural animation

4.1.7 Animation continuity

Continuity of motion is an important concept when we are trying to judge whether a particular animation is smooth. In real life, moving an object from point A to point B always takes some amount of time. Therefore the position curve of any movement in respect to time will obviously be continuous. This should be then the basic requirement for animation too – it shouldn't look like as if an animated object instantly warps to a new location, or just

changes its rotation without any frames in-between. Of course, an animation is quantized to a certain frame rate, and thus all movement takes at least the duration of one frame, and is discontinuous between the frames. But if the animation is defined such that the position $P(t)$ or rotation $R(t)$ of the animated object is known for all the values of time t, continuity of the motion is possible. And even in case of keyframe animation, in practice we interpolate the motion between the keyframes, and the interpolating curve can be analyzed for continuity.

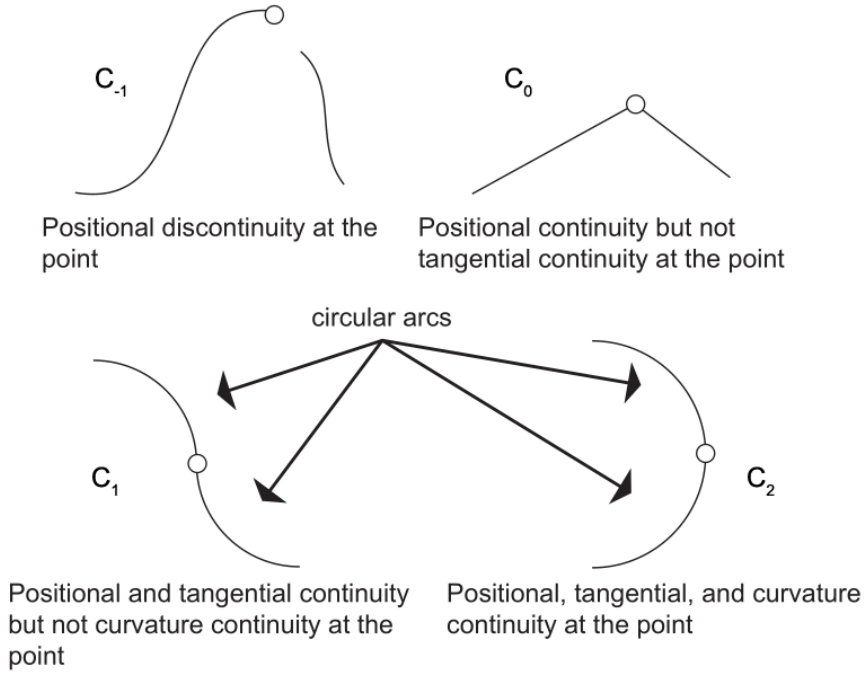


Figure 4.8: Types of curve continuity [30]

The *parametric continuity* of a curve can be defined in the following way [30] (see Figure 4.8 for examples of position curves with varying continuity):

- No continuity (C_{-1}) implies that there are two curve segments that are not joined (often the case), or there is a single point where the function has a discontinued value.
- Zero-order (C_0) continuity implies that the curve itself is continuous.
- First-order (C_1) continuity implies that the first derivative of the curve is continuous, or equivalently the tangents of the curve are the same at left and right limits of any point.

- Second-order (C_2) continuity implies that the second derivative of the curve is continuous. This means that the curvature is continuous, or equivalently that the change rate of the curve tangent is continuous.

When this is applied to a function of position over time, the continuity classes can be applied to familiar kinematic concepts. C_0 implies a continuous position curve, C_1 implies a continuous velocity curve, and C_2 implies that the acceleration curve is continuous.

4.1.8 Splines and easing

Splines are widely used in interpolating between keyframe poses in computer animation. A spline is a function that is defined by joining pieces of polynomial functions together. Using splines for interpolation instead of a singular polynomial function of high-order avoids the Runge's phenomenon [32], that is, the tendency of such functions to oscillate near edges of their interval.

There are several desirable properties for a given type of spline. The curve should be smooth, at least C_1 continuous, which is the case with all common spline types. C_2 continuity can be harder to achieve. The next useful property is if the curve has local control, meaning that altering the shape of a single curve piece doesn't affect the shape of rest of the curve. Finally, it is important how the shape of the curve is defined, and whether it passes through (interpolates) its control points. [42]

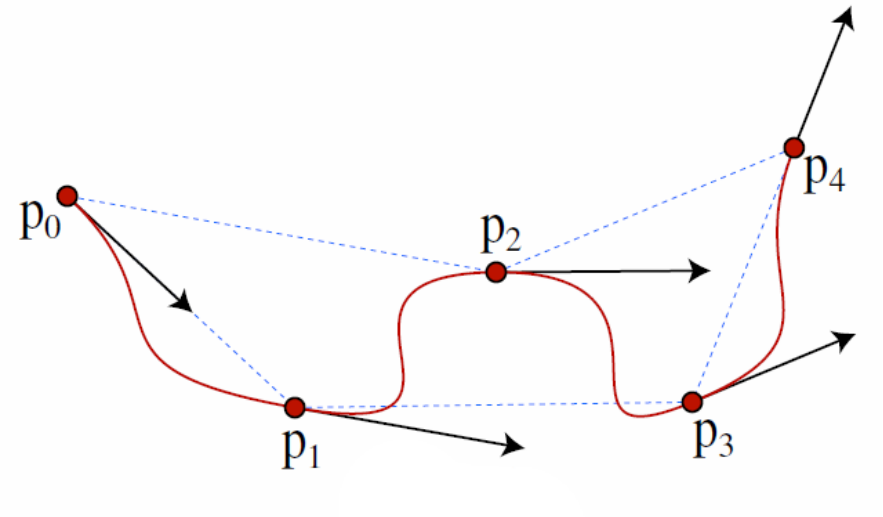


Figure 4.9: A Catmull-Rom spline interpolates all its control points [39]

For interpolating between keyframe poses, a spline such as the Catmull-Rom spline is appropriate. This spline passes through all its control points, and is fully defined by those points (see Figure 4.9). [39]

For creating paths of motion by hand, a spline whose shape can tweaked with additional control points, is useful. High degree of continuity (C_2) can also be important – this can be achieved with a B-spline [26].

The Table 4.1 compares the commonly used types of splines. The splines are compared for the maximum degree of continuity that is reasonably achievable, if they have local control, if they pass through (interpolate) their control points, and if they have additional parameters or control points for defining the shape of the curve.

Spline type	**Continuity**	**Local control**	**Interpolates**	**Shape ctrl.**
Catmull-Rom [39]	C_1	yes	yes	no
TCB [22]	C_1	yes	yes	yes
Bézier [42]	C_1	no	yes	yes
B-spline [42]	C_2	yes	no	yes

Table 4.1: Types of splines compared

Splines can also be used to control the velocity on the traversed path, as suggested by Bartels and Hardtke [5]. But for single motions, simple polynomials can be used to apply slow in and out and other *easings* to the motion. The term seems to be popularized by Robert Penner [31], who assigned names for multitude of functions for motion easing. Several of them are shown in Figure 4.10.

Here *ease in* means a motion that starts slowly and accelerates towards the end. *Ease out*, in turn, means a motion that starts fast, but decelerates towards the end. The combination, *ease in-out* then means that the motion starts and ends slowly, and the maximal speed occurs in the middle. Cubic, quadratic, quartic, quintic and other specifiers in the name then refer to the type of the curve. The ones listed refer to degree of a polynomial function used for the curve, but Penner also defines curves such as the easeInElastic (Figure 4.10), with more complex curves. We use several polynomial easing functions in the implementation of the dancer in Chapter 5, and refer to their type (in or out) and degree there.

For simple linear interpolation, in computer graphics, we often use the term *lerp*, because this is such a common operation. Likewise, the use of linear interpolation can be called *lerping*. We note this, because linear interpolation is used quite heavily in Chapter 5, and we use this shorthand in the text.

Figure 4.10: Several easing functions [33]

4.2 Procedural animation techniques

In this section, we will discuss methods present in the literature that can improve the quality of our animation compared to simply generating "ideal" trajectories of motion with IK. In Chapter 5 we then present a design and implementation of animation for the dancer character using some of these methods, and justify why we chose to use those methods.

4.2.1 Adding inertia with PID control

A PID controller is a feedback-based control system used mostly in industrial processes. It, however, has properties that make it interesting for smoothing out and adding inertia to procedurally generated motion. Let us review the structure of a PID controller (based on introductory paper by Tehrani and Mpanda [35]), and then consider its applicability for animation.

The PID controller consists of three parts: proportional (P), integral (I) and derivative (D) control. Each of the parts act based on feedback provided by the controlled system, more accurately, based on the error or difference between the desired value a controlled parameter and the actual value that the parameter has. Often the controlled parameter is a position or an angle, but it can be something else entirely.

Consider a PID-controlled system at a point in time t. The desired value we have for the system at that time is called the setpoint (SP). The actual value of the system is called the process variable (PV). Then the error in the process variable is $e(t) = SP - PV$. Now the PID controller output is $u(t)$, and can be interpreted as the rate with which the process variable of the system should be changed at the time instant t. The output is defined as:

$$u(t) = K_p e(t) + K_i \int_0^t e(x)\,\mathrm{d}x + K_d \frac{d}{dt} e(t) \tag{4.18}$$

Where the three terms correspond to P, I and D parts of the controller, and K_p, K_i and K_d are tuning parameters that determine how much each part of the controller affects the output value.

The proportional term of the controller, $K_p e(t)$, simply outputs the value of the error. If the output is used to be the acceleration or force applied towards reaching the target position, a system with only proportional control is equivalent to a physical spring where $F = kx = K_p e(t)$. In PID controllers, the P term usually contributes bulk of the overall value. But if the P term is only used, the controller will never fully converge to the target value, but rather oscillates around it.

The integral term of the controller, $K_i \int_0^t e(x)\,\mathrm{d}x$, is the accumulated error from time 0 to time t, proportional to both magnitude and duration of the error. Note that the error is signed, this makes it so that the integral term will finally converge towards 0 when the system reaches its steady state. The integral term will help in eliminating the steady state error, but it will make the response of the controller slower, and may cause it to first overshoot the target value.

The derivative term $K_d \frac{d}{dt} e(t)$ outputs the rate of change of the error. This makes the controller to somewhat predict the system behavior, and compensates the overshoot and increase in settling time caused by the integral term. In industrial processes where PV is measured from the system, the derivative term is prone to noise in the measurement, and a low-pass filter applied to PV for derivative control to reduce the noise. Derivative control has variable impact on system stability in real-world applications, and due to this is used less often than the P or I components [2].

The Table 4.2 summarizes the effects when increasing the value of K_p, K_i or K_d:

In industrial processes, the PID controller is used adjust motors to drive physical systems reliably to the desired goal. One of its first uses was the

Term	Rise time	Overshoot	Settling time	Steady-state error
K_p	Decrease	Increase	Small change	Decrease
K_i	Decrease	Increase	Increase	Decrease significantly
K_d	Minor decrease	Minor decrease	Minor decrease	No effect in theory

Table 4.2: Effects of terms in the PID controller

automatic steering of ships [6]. And it can also be adapted to situations where the setpoint is too a result of measurement in the system.

For example, in the Embedded Systems course held by the Computer Science department of Aalto University, students implement control for a robot car, that follows a track on the ground recognized by sensors in the car. A PID controller is used to steer the car. When one of the eight sensors in the bumper of the car (see Figure 4.11) registers the tape that marks the center line of the track, its displacement from the center point of the bumper is used as the value of error for the PID controller.

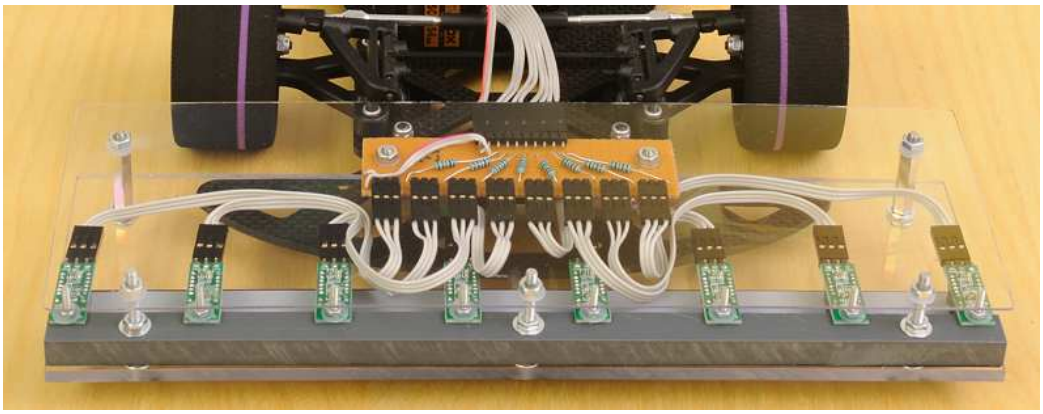


Figure 4.11: Front view of the car used in Embedded Systems course at Aalto University, showing the sensors used for track recognition. [18]

In procedural animation, we can use PID control to add physical characteristics to the motion. First, we generate a motion for an action of a character we want to animate. Then we drive the character such that the PID controller setpoint is determined by our generated motion, the actual position is affected by a force supplied by the output of the PID controller, and the value of error in the PID controller is the difference between those positions. We deliberately tune the controller so that the I term is large enough to cause noticeable overshoot in the motion.

Now the motion looks smoother, more physical and organic, because of

how human motion tends to overshoot (see 2.5.5) and because the controller also causes the motion to have slow in and out velocity (see 2.5.7), or inertia. If the transition to the next action would have continuity issues, PID control can also help to smooth them out.

4.2.2 Style-based IK

Style-based IK is a system to constrain IK poses of a character based on learned model of human poses. It is proposed by Grochow, Martin, et al [16]. If an IK system is only constrained by limits of how the human joints can move, much work has to be done to manually tune trajectories of multiple IK effectors to produce realistic poses and animation. Therefore, a system such as style-based IK could significantly improve the quality of procedural IK animation. In the paper, it is claimed to be useful in animation where an animator sets keyframes for the IK effectors, and the poses of the character are determined by style-based IK. It can be also in other applications, such as computing locations of missing markers in motion capture data, posing characters interactively and determining a likely pose of a person in a 2D image.

Because style-based IK is an advanced technique for procedural animation and various other usages, we introduce it here rather than in the computer animation fundamentals section (4.1). Let us summarize how the system works. Only an overview is given, as the required math is somewhat involved.

The main idea of style-based IK is to build a probability distribution function from poses obtained from motion capture data. This function will then tell us how likely a particular pose is when the IK effectors are set in our desired configuration. The most likely pose should then have a style of motion similar to the training data, and look natural.

For each pose in the motion data, a vector $\mathbf{q_i}$ is built, containing joint angles, and the position and orientation of the root transform (a total of 42 elements). Here i is an index over the training poses. For each vector $\mathbf{q_i}$, a *feature vector* is also constructed, this contains the data that we want the system to learn from. The feature vector consists of:

- The joint angles from the $\mathbf{q_i}$ vector (but not the root transform)
- Vertical orientation of the character, this is represented by part of the rotation matrix of the root transform
- Velocity and acceleration of the joints and the character root

The feature vectors are then used the construct the function which assigns a likelihood for a pose to be of the same style as the training data. The probability distribution function is constructed using the Scaled Gaussian Process Latent Variable Model (SGPLVM), described in [27]. Various optimization algorithms can be then used to find an input pose for the probability distribution function, that is close to the required IK pose, and for which the function yields a high value of probability.

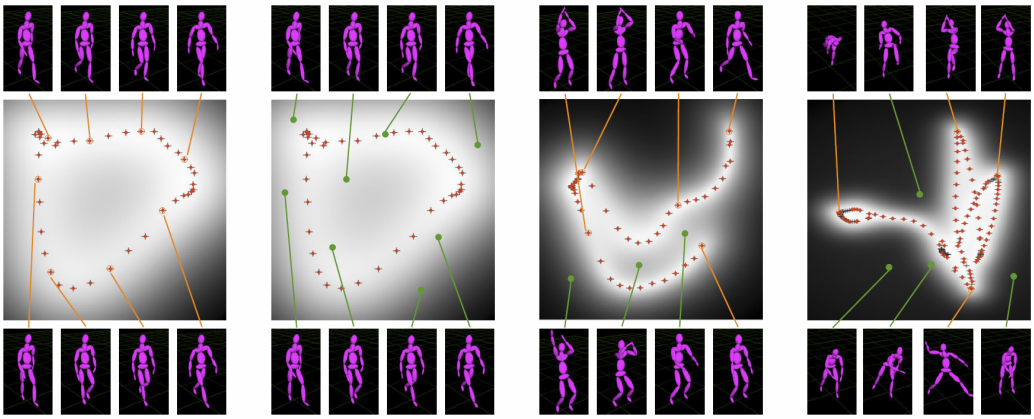


Figure 4.12: Style-based IK builds a probability function, which tells how likely a pose belongs to the same style as the training dataset. The probability distribution can be visualized using a height map. Orange points mark original poses in the training data. Some of the poses are shown explicitly.

4.2.3 Move trees and motion graphs

Another way of procedurally generating animation is to combine existing motion data (created by an animator or recorded with motion capture) using some heuristic.

Move trees [28] are widely used in the game industry – they are a structure which connects an animation and all the animations that may be played next with a directed graph (see Figure 4.13 for example). The graph is constructed manually by the game programmer or an animator. At runtime, the followup animation is then chosen by game logic and/or player input. The problem is that between clips there are usually a lot of possible transitions that can be taken, and it is often impossible to make animations that naturally blend in a seamless fashion. But if the animations are reasonably similar, this can be solved by linearly interpolating the control over from the current animation to the next when the transition occurs. It is suggested that approximately

10 frames as the duration of blending between the animations works in most cases [28].

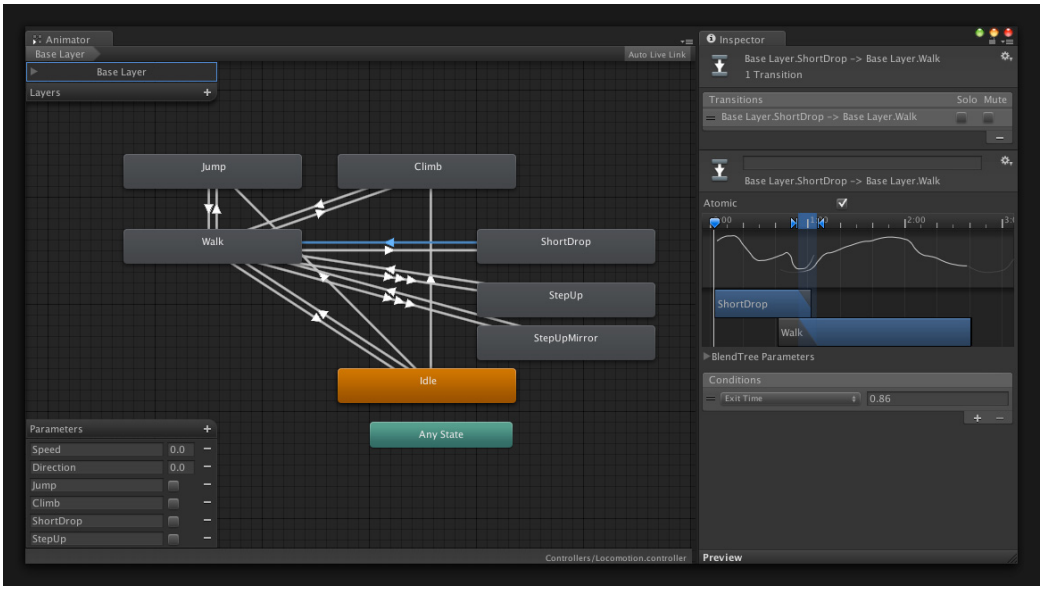


Figure 4.13: Unity3D's [40] Mecanim animation system is an example of a move trees implementation

If the set of motions that can be performed is large, manual graph construction is not feasible. In the case of our dance game motion synthesis, we have quite a few motions the player can take (analyzed more accurately in Section 5.1), and then we have the constraint that these motions should take an exact, arbitrary duration depending on the stepchart. To achieve the move of a required duration, we might be able to blend between a slower and a faster move – given enough sample moves of different duration, this could look natural.

A more automated approach is presented by Kovar et al [24]. Their paper describes *motion graphs*, a structure which describes how an arbitrary continuous set of motion capture data can be re-assembled to produce new motion. It is similar to move trees in that it's a graph describing transitions between moves, but the graph is assembled automatically according to heuristics, and the individual moves from the motion capture data don't have to be manually cut as separate clips.

In the motion graph system, the first task is to identify portions of the input animation data that are sufficiently similar so that straightforward blending between them would generate valid transitions. Like with the style-based

IK system, position, velocity and acceleration between different postures of the character are compared. Instead of comparing the joint angles, the paper notes that the skeleton is only means to an end, and the polygon mesh is what is visible to the viewer, so a select set of points from the mesh is compared instead. These have to be selected in such way that they adequately represent the full set of vertices of the mesh.

Before the motion graph is generated, we select the number of frames that should be used for transition between two motions, and denote it with k. Then when we consider the similarity between two frames A and B, we also account for similarities in the k frames after A and k frames before B that would be blended together, if this transition was performed. According to the paper, the comparison of multiple frames additionally incorporates information about the velocity and acceleration of the points to the metric.

Squared distances between pairs of frames from A and B are then compared, and a weighted sum is used as a metric of similarity. Additionally, the point clouds are rotated such that the rotation minimizes the distance between the points. This rotation can be computed analytically – the solution is detailed in the paper. The rotation enables us to find motions that are similar, but occur in different orientations.

Once the motion graph is generated, we can then produce motion by traversing it. A walk of the graph will take the character from a pose to another. We can then search the graph for walks that satisfy our desired end pose. The naive solution of enumerating all the graph walks and choosing the

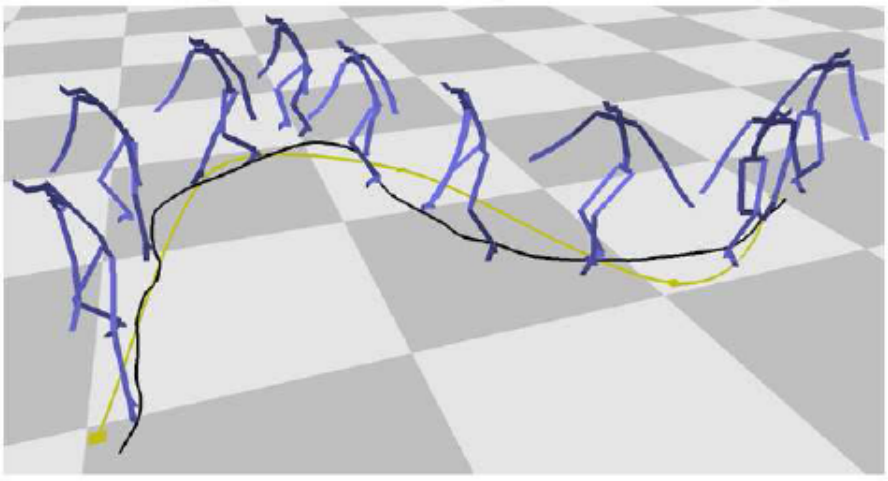


Figure 4.14: Motion graphs are a good solution to synthesizing motion following a set path [24]

best one is definitely too expensive, so we compute the difference between a graph walk and an "ideal motion", using a case-specific error function supplied by the user. We try to find a walk for which the error is under an acceptable threshold, and terminate searches for any paths that exceed this error threshold.

The motion graphs are suggested as solution for the problem when we have a set path which a character should traverse, and we want to generate suitable motion to do that (see Figure 4.14). For dance game motion synthesis, the system would have to additionally be constrained so that the character would only step once for one tap note, and jump once for two notes – no steps in-between should be allowed. This constraint could perhaps be incorporated into the motion search, but the authors of the paper mainly use the system for path synthesis, so it's not clear if this would work.

Chapter 5

Designing the dancer character

In this chapter we will describe the how the tutor dancer can be animated. First, we analyze the complexity of the problem: how many states can there be in the animation? Based on this, we will propose an initial design, and then improve on it. We then explain further how the individual moves are animated in the final design. Finally we cover other animation details, such as how to make the dancer look at the screen, and how to simulate movement of the dancer's ponytail. Our final implementation is shown in Figure 5.1.

5.1 Problem complexity

If the number of possible moves that the dancer can do is small, the animation could simply be implemented by having an explicit animation clip for every move, possibly having different animations based on which move lead to this particular move, making the transitions between the two moves seamless.

Let us consider the how many basic moves there are, and leave tapping with hands and evading mines to the center panel out of the picture. In the 4-panel dance game, we can be on the dance pad in $4 * 3 = 12$ different positions (first choose the panel for the left leg out of 4 options, then choose the panel for the right leg out of remaining 3 options). Out of any of these positions, we can step with the left leg into 2 other panels, and similarly with the right leg. We can also tap the same panel with either leg. So the number of different steps we can do at any position is 6.

At any position, we can also jump into any position. So number of possible jumps at a position is 12. This makes the total number of possible moves at a position $6 + 12 = 18$. And therefore the number of moves that can be done overall is $12 * 18 = 216$.

If we don't consider the previous move in the animation, this number of

moves is feasible to record with motion capture. But if we want seamless transitions, then we have to transition from any move to the next move, yielding $216^2 = 46656$ required animations, which is not feasible to capture, and quite likely too many animations to load into memory. In the future, we also want to extend the dancer to be able to play the double game mode, in which the player uses two dance pads simultaneously. This would increase the required animations even further. With symmetry considerations (the player uses two identical pads) the animation count doesn't increase quite so drastically, but there are quite many moves for transitioning between the pads.

In the single pad case we can also use symmetry to optimize the number of moves. Intuitively this should halve the number. So then the required number of animations would be $(216/2)^2 = 11664$, which is still too many to capture. One idea is if we considered motions in the player's local coordinates instead. Then majority of those motions we required above would be classified as same. But this is true only if feet motion alone is considered. The player will look at the screen, so movement of the upper body differs, and with different upper body movement presumably uses different hand movement.

Also we probably want different animations for moves with different durations. To achieve the exact duration, we can interpolate between the moves, but this still increases the number of moves required by a factor of $2 \ldots 4$.

We can conclude that using captured or hand-animated clips as a part of the system is definitely feasible, but implementing the system by only playing back clips in sequence is not.

5.2 Initial design, choice of platforms

As the trivial just-animation-clips solution was deemed unfeasible, the project plan was to first animate the dancer in a rudimentary way, using inverse kinematics. We would move the legs in arcs, and keep ground plane position of the body in-between the positions of the legs. Because it was quite clear that this would not result in realistic animation, additional techniques would then be implemented to improve the animation if the time permitted this.

The platform, a 3D graphics engine, also had to be chosen. The dancer AI already existed in a Java-based codebase, along with an editor software for our game, Dance Tricks. The codebase for the actual game (Figure 2.1), uses an open source dance game engine Stepmania, implemented in C++. The choice of the graphics engine then was influenced by several factors. First, a solution that could integrate with our existing codebase would be desirable.

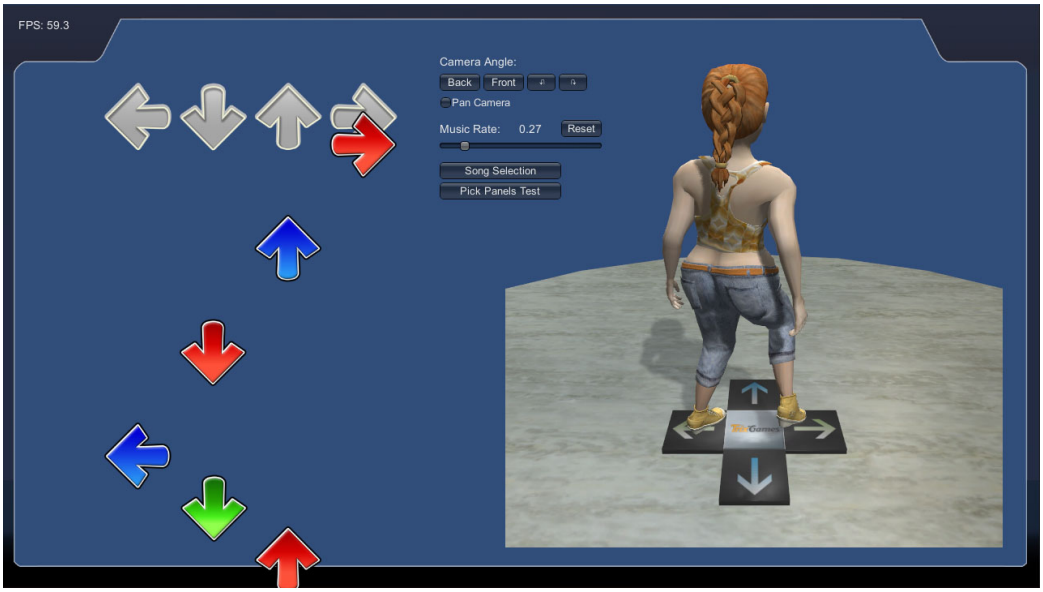


Figure 5.1: The gameplay screen from our finished dance game tutor system. The stepchart is automatically played, and Elizabet the dancer demonstrates how to play.

Second, some system that implemented skeletal animation out of the box would surely be required, as writing one from scratch would take more time alone than was allocated for this project.

The Stepmania engine actually has some support of dancing characters (performing random predefined dance moves rather than actually playing the game, see Figure 5.2), but this implementation is old, and it was very likely that we would run into many limitations if we tried to adapt this system into actually demonstrating gameplay of the game. In interest of unifying our platforms, it would of course be good to implement the dancer on top of Stepmania, or with Java.

If we were to use Stepmania, a separate 3D graphics engine could be integrated and its rendering composited into the game. But this would also probably prove to be a difficult task. Ultimately, the compatibility with our existing solutions had to be sacrificed in the interest of easy development for this project, and Unity3D was chosen as the graphics engine. It supports skeletal animation and is widely used so that enough documentation is available. Additionally, in Unity3D, the programs are written with the C# language, which is very similar to Java. The part of the Java editor codebase which implements gameplay of a dance game would minimally have

Figure 5.2: There is support for dancing characters in Stepmania, but they cannot actually play the game, and are meant more as background visualizations [10]

to be ported to Unity3D, but similarity between Java and C# makes the conversion straightforward.

In fact, automated solutions for porting Java to C# exist. Most of them are based on translation of the code, but this does not yield idiomatic C#, and hand tuning is required [13]. The most promising solution was an implementation of a Java VM on top of Mono called IKVM.NET [14]. This could theoretically let Unity3D run Java code, as its runtime is based on Mono. Unfortunately as of Unity 4.3, the heavily customized version of Mono used by Unity would not successfully run our code, even though it ran fine on vanilla versions of Mono. As the automatic solutions weren't feasible, subset of the editor codebase that implemented the dance game gameplay was then ported by hand to C#.

The AI code could also have been ported over to Unity3D, but that would have required a considerate time investment. Instead, a deserializer for plays generated by the Java AI was written, as the could already be serialized into a JSON-based format at the Java side. Another upside of this decision was that no new bugs would be introduced by a new port.

In the end, the initial implementation consisted of an Unity3D-based dancer character, using skeletal animation driven by the two-bone IK implementation built into Unity. It would read the play of a stepchart generated with the Java-based AI code, and control the dancer according to the play. This resembled movement of an actual player, but the motion was quite robotic and not smooth, as predicted. There was some time to improve the system, so additional techniques were put to use.

5.3 Design based on animation blending

For the improved design, we tried to use animation clips of stepping and jumping for individual moves, combined with IK re-targeting animation that doesn't perfectly match the desired move. The newly-added animation solution in Unity 4, called Mecanim, also allowed for parametric interpolation between two animations, so different step and jump animations could have been blended according to the change in position and rotation of the dancer's legs. Mecanim uses move trees [28] for keeping track of the animation state, and also interpolates between the animations when transitioning from one animation to another.

The problem for this implementation was that it is difficult to enforce strict timing for animation within Mecanim. The transition period between animations is mandatory, and seems to always take at least one frame, but sometimes several. This makes it hard to match state of our own control system with Mecanim state. Exact times for the state transitions are hard to set, as Unity only provides a slider for controlling the transition period in the GUI of the move tree system. It is not clear whether the transition times between two animations actually increase the combined duration of those animations.

Also, changing the speed of an individual animation programmatically is not supported, only the speed of the whole system can be altered. This makes it impossible to play several animations (for the upper and lower body, for example) at different speeds. The animation clips can be blended between themselves to alter speed, but the blend is non-linear, making it hard to match the desired animation duration. Finally, the API is limited in its ability to give any overview of the state of the system, making debugging harder.

Because timing the animations was so unreliable and difficult with Mecanim, we ended up only using animation clips for a base idle animation played while IK controls the actual movement, and for basic animation of hands. The idle animation has no timing requirements, and they are more relaxed for the hand animation, so this works well timing-wise. But the IK animation was still stiff; how this was improved is described next.

5.4 Final design

The final design was then an effort to introduce more detail and realism to the animation, still using only IK to drive the legs to ensure accurate timing of the moves. We, however, switched to use full-body IK (see Figure 5.3).

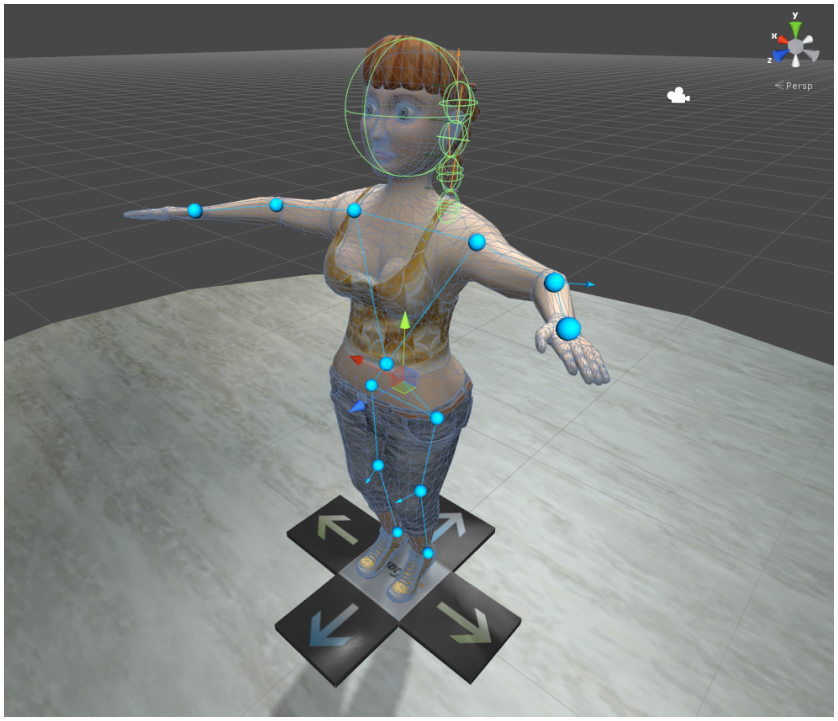


Figure 5.3: Our dancer was almost exclusively animated using full-body IK animation. The IK effectors (feet, hands, body, shoulders and thighs) are shown as blue spheres in the picture. The elbows and knees aren't effectors, but the IK library [25] marks them with blue regardless.

We considered the use of motion graphs or style-based IK, introduced in Chapter 4, but it seemed that there would not realistically be enough time to implement either technique. The downside with motion graphs also was that it was unclear whether we could implement proper constraints for this system for searching motion from the graph: the legs should end up in specific positions, the move should have a certain length, and no in-between steps should be allowed. Style-based IK seemed fitting, but also complicated to implement.

As per recommendation from the instructor of this thesis, we then ended up using PID control to smooth out the rigid movement and adding inertia (overshoot) for motion of the upper body. More specifically, the IK effectors in thighs, torso, shoulders and hands of the character would not be directly controlled, but the desired position would be passed as the setpoint value to a PID controller, and then the output of the PID controller would be taken as the velocity changing the position of that IK effector. We would then tune

the gains of individual controllers to achieve desired overshoot and movement smoothing. In practice this meant that the inertial gain was 4 times larger than the proportional gain, at minimum, and the derivative component of the controller was left unused in this application.

Several improvements were made for the actual control, too. When stepping, the head would now bob towards the leg that the dancer raises, and the shoulder from that side would be lowered a bit. With jumps, we fine-tuned anticipation and follow-through phases of the jump so the movement would look reasonable with different jump durations. Also, we computed the vertical position in the jump according to gravity (the ground plane position would be linearly interpolated). The angles of the body and feet would now be interpolated between the target angle for the next move and their previous angle.

Finally, our dancer character had a braided hair, so its movements were simulated using the NVIDIA PhysX physics engine [29] employed by Unity.

5.5 System structure and operation

Let us then cover the structure and operation of the dancer's animation system in detail, in the final design. The system is split into two main components: the AI controller component and the movement component. The movement component contains the animation logic for different types of moves. The AI controller iterates over the play generated by the AI, and signals the movement component to perform the required moves. This separation enables the character to be driven by other sources of movement than the AI play. For example, our system contains a test that allows the user to manually move the dancer by clicking on dance pad panels with the mouse.

Control flow of the animation system is displayed in Figure 5.4. We will next give a detailed explanation of the components in the system.

5.5.1 AI controller component

The AI controller performs actions with the blue background in Figure 5.4.

When the system is started, the stepchart to be played is loaded an passed to the controller, along with the AI play computed for that stepchart.

Then, at the first frame, we start to iterate over the AI play. The play dictates in which state the legs (and hands, in case of hand notes) of the dancer are during a particular note row in the stepchart. We keep track of the current state, the next state, and the state after that, to be able to

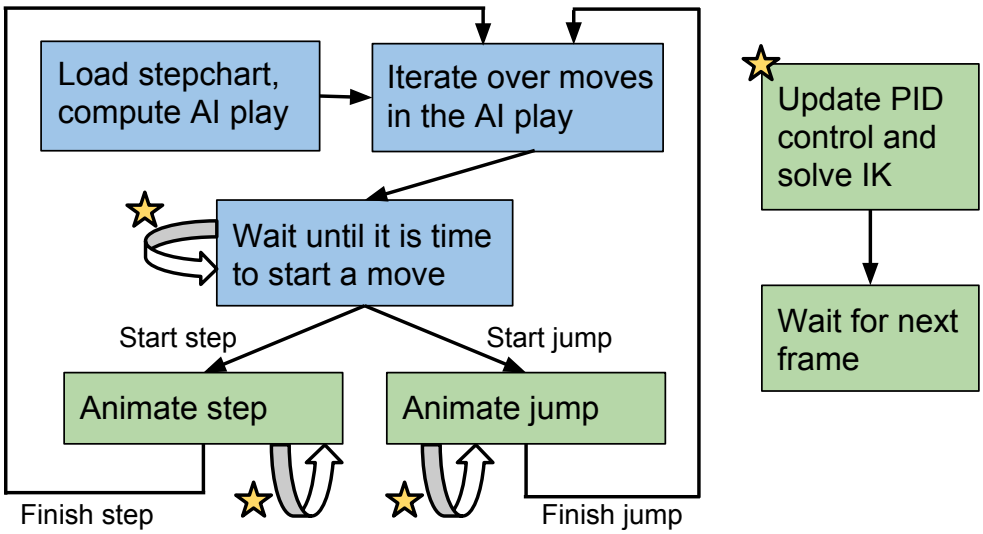


Figure 5.4: Control flow of the dancer animation system. Blue nodes are actions taken in the AI controller component, and green nodes are actions taken in the movement component.

calculate durations of the moves and the follow-through movement. Duration of the next move will be the time between the current and the next move. If the move would be longer than 500 ms, it is limited to that duration (Lasseter suggests 420 ms as duration of the longest individual move [26], so this should be enough). If the move duration is now 500 ms or less, the move is started immediately; else we wait for the remaining time before the move. Similarly, the duration and start time for the move after this move is computed.

If the move is a step, we signal the movement component do perform the step, passing the target panel of the moving leg and the duration of the move as arguments.

Similarly, if the move is a jump, we signal a jump should be performed, and pass target positions of both legs and duration of the move. After the jump duration has passed, the dancer has moved to the target stance, but has not yet done the follow-through movement after the jump. The duration of the follow-through has to be limited also. If move following the jump is a step, we limit the follow-through so that at least 150 ms is left for the duration of the step. If the follow-up move is a jump, we require at least 350 ms as its duration. Between fast consecutive jumps, this then completely cuts the follow-through of the first jump, and merges it with anticipation of the next jump, so the dancer will only crouch once between the jumps.

Due to time constraints, we deliberately ignore moves where the dancer should evade a mine, and notes that should be pressed with hands. Mines are ignored because the existing AI implementation only evades them by moving the endangered leg to the center panel. In practice, though, the mine should be evaded by just lifting the leg up, unless there are mines bound for both panels where the player is standing, in which case the dancer should jump to the center panel. Hand tap animation, in turn, requires an additional move to be animated, and is no small undertaking either.

When a move has started, the AI controller component then waits for it to finish. Then the next dancer state in the AI play is considered. When all the states have been processed, the controller terminates.

5.5.2 Movement component

The movement component performs actions with the green background in Figure 5.4.

For animating the moves, we use a third party full-body IK library for Unity called Final IK [25], as Unity does not have a native full-body IK implementation (it contains a two-bone one). Using full-body IK we can specify effector positions for thighs and shoulders in addition to feet and hands for use in our animation. It also allows the aforementioned hand tapping animation to be created as future work. This IK implementation is based on both two-bone analytic IK and the FABRIK algorithm. It first computes positions for feet and hand using analytic IK, and then applies iterations of FABRIK (a comment in the library source justifies this with: "Trigonometric pass to release push tension from the solver"). The library also provides a component for look-at IK, i.e. for rotating the body, head and eyes towards a target of interest. This is used to focus the character's attention to the front in the scene, where the game screen would be.

The movement component is called every frame. First, if there is an ongoing move, it is animated – the animation routine alters the setpoint values for the PID controllers. Then, regardless of there being ongoing moves or not, the velocity values of IK effectors are read from the PID controllers. The change in position is computed from the velocity multiplied with the delta time for that frame, and is applied to the IK effector position. Computing IK at all times ensures that the overshoot effect of the PID control continues even when there is no actual move in progress.

After the IK effectors have been updated, the full-body IK system is solved and the solution is written to joint transforms of the character skeleton. Finally, the look-at IK system is solved and the solution is applied on top of the animation IK solution.

5.6 Animation of moves

We shall then consider how the moves are animated in the movement component.

5.6.1 Step animation

First, we analyze the implementation of the step animation. For the resulting animation, see Figure 5.5.

(a) (b) (c) (d) (e)

Figure 5.5: Five frames from the dancer's step animation

The step routine gets the leg to move, target panel of the leg and the move duration as an argument. It is ran every frame before the PID controllers are updated, until the move duration has passed. In the routine, we first compute the target position of the leg. The position will be between the panel center and the inner edge. This is to simulate faster moves being more constrained in their length; in real play, the players tend to step closer to the inner edge to conserve energy. The destination position is then determined by interpolating between the edge and the center, according to the move duration. The interpolation is non-linear: the reduction in foot travel distance starts slowly, but its rate increases towards the end (a cubic reduction curve).

Next thing we do is to compute the rotation of the stepping foot at the end of the motion. This is done by first taking the vector between both legs in the end position, and taking the vector perpendicular to that in the ground plane (let us call this vector the feet normal). The end rotation of the stepping foot will then be a halfway lerp (linear interpolation) between the rotation of the feet normal vector, and rotation of the other, supporting

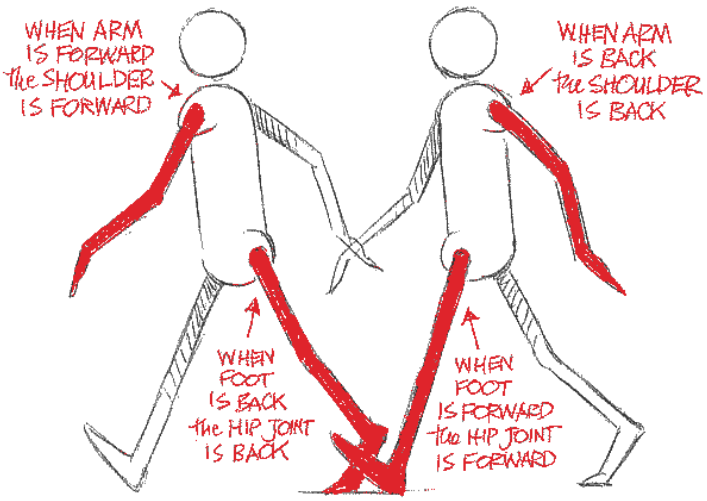


Figure 5.6: The basic human walk cycle [47]

foot. This is to make it so that both legs will try to face at roughly the same direction, but often not exactly, avoiding the twins pose (see 2.5.11). We also compute the height of the foot position at the vertical peak of the move. For this, we define the height that is used for the slowest move (500 ms) and choose the used height by interpolating between this maximum and half of the maximum. A cubic reduction curve is used here as well, so that the reduction in height starts slowly.

At this point we start a basic arms swing animation clip, in which the arm at the side of the supporting leg will swing in the direction of the stepping leg, and the arm at the side of the stepping leg will swing in the opposite direction. This follows the movement in the basic human walk cycle (Figure 5.6).

Now we can play the stepping animation. During the move, we lerp the ground plane position of the foot IK effector between the start and end points. The vertical position of the effector will follow a sinusoidal path. We raise the body of the character half the amount the foot is raised, and also lower the shoulder IK effector of the stepping side by this amount. The heel of the foot will be rotated downward; the amount of rotation will depend on the vertical position of the foot. We use a function found by tuning for this purpose, which quickly changes the angle at the beginning so that the heel seems to stand on the ground for a while, then the rate of change for the angle decelerates sharply when the heel is lifted up.

We lerp the rotation of the body between its starting rotation, and the end rotation of the foot we computed earlier. Then we interpolate the rotation

of the foot between its starting rotation, and the rotation of the body we just computed, so it will lag a bit behind the body. This is in accordance with the follow through and overlapping action guideline (Section 2.5.5). The interpolation parameter used is the percentage of the move's completion, but with a cubic ease-out function, which makes the foot angle lag behind a little less prominently. We don't alter the rotation of the supporting leg, as it should stand firmly on the ground. In some cases, where the dancer turns a lot, we may end up with the legs in unnatural rotations that differ too much from each other. However, this occurs mainly with some spin motions, and the fact that we take the supporting leg's rotation into account in the end rotation prevents majority of the problems.

Currently we actually feed the ground plane position of the foot to the IK effector through a PID controller. The PID controllers are more useful with upper body movement smoothing, but this has the effect of reducing the foot travel with fast motions even further in combination with positioning the end point towards the panel edge. Only the ground plane coordinates are controlled this way; to maintain accurate sync to the music, the vertical position of the foot is controlled directly. The integral component of the controller also has to be zeroed when the foot hits the ground, so the foot remains stationary. In retrospect, this use of PID control doesn't add much to the animation that couldn't be achieved with better positioning of the step target point.

5.6.2 Jump animation

Now we analyze the implementation of the jump animation. For the results, see Figure 5.7.

The jump animation operates similar to the step animation. As arguments, we get the target positions for both legs, the move duration, and possible time limit for follow-through phase of the jump. The jump will consist of three phases: the anticipation phase where the dancer crouches before jumping, the actual jumping phase, and the follow-through phase where the dancer crouches again when landing. We compute durations for each of these phases, such that the anticipation phase will take 25% of the jump duration, the jump will take the remaining time, 75%, and the follow-through phase will take between 80–150 ms depending on the duration of the jump, but can be limited by the provided time limit (even omitted completely).

When there are fast motions in the stepchart with jumps mixed in-between, in reality the jump follow-up phase would be mixed with the dancer already starting to step towards the next note. In this implementation, this is not possible, and we have to compromise between a realistic-looking follow-

through phase or a fast response after the jump. Simultaneous action like this would be useful to implement as future work. We will propose how the overlapping action could be handled in Section 6.2.

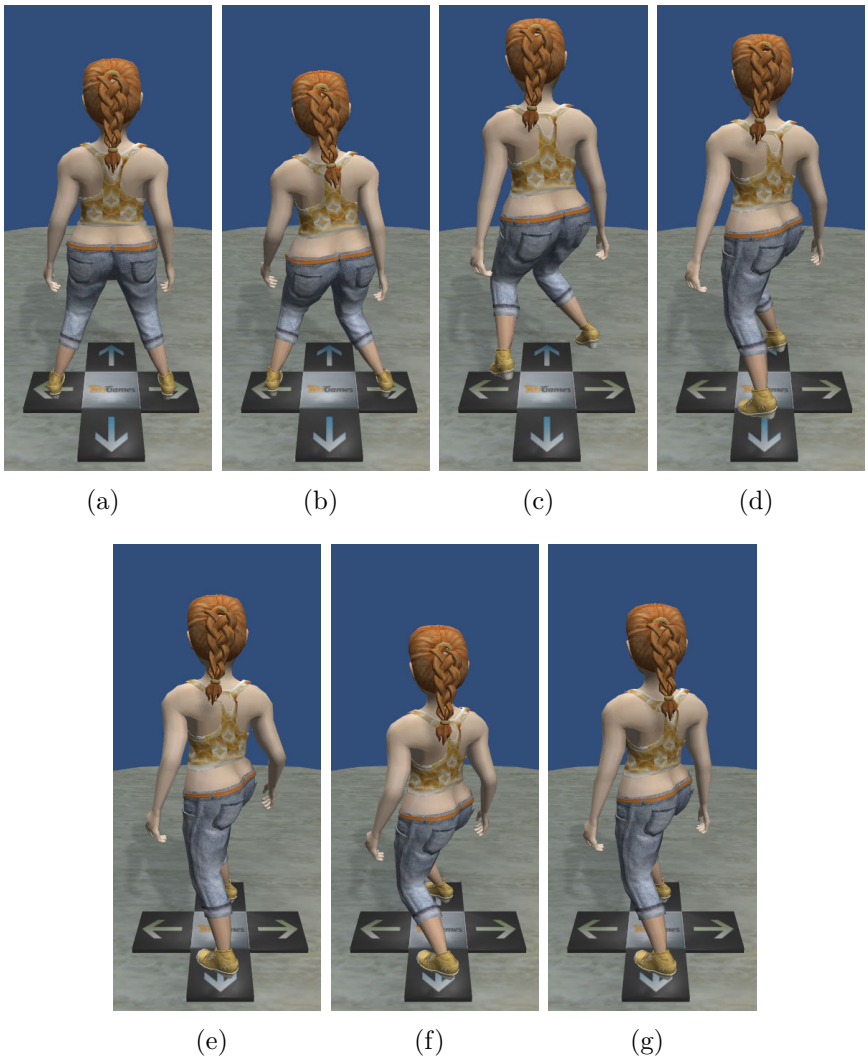


Figure 5.7: Seven frames from the dancer's jump animation, when jumping from the ← and → panels to the ↑ and ↓ panels. Frames (a) and (b) are in the anticipation phase, (c)–(e) in the jump phase, and (f) and (g) in the follow-through phase.

In calculating vertical position of feet in the jump, we employ equation of displacement in constant acceleration [48]. Here the acceleration is gravity g, which has the opposite direction compared to the initial velocity v_0. At time t the vertical position y of the dancer's feet is:

$$y = v_0 t - \frac{1}{2} g t^2 \tag{5.1}$$

We also want to compute such v_0 that the dancer has just grounded when the jump duration has passed. This is:

$$v_0 = \frac{g t_{jump}}{2} \tag{5.2}$$

5.6.2.1 Anticipation phase

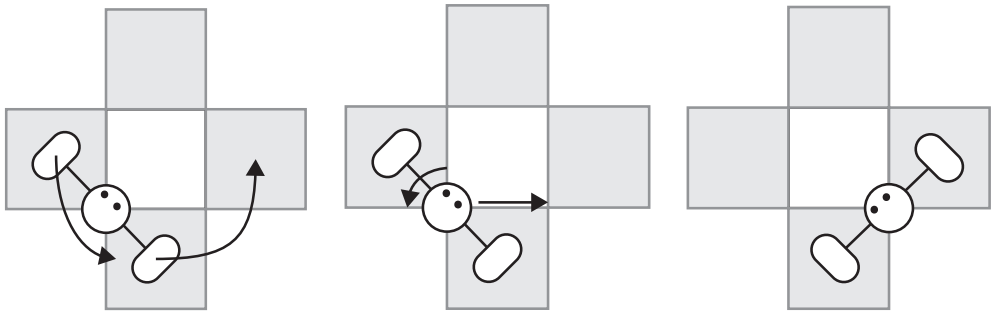
The jump begins with the anticipation phase, show in frames (a) and (b) in Figure 5.7.

In this phase, the dancer crouches and contracts her leg muscles to prepare for the jump. To implement the crouch, we adjust the root position of the dancer vertically, while having the legs stay on the ground using IK. We could just model the crouch using the negated value of Equation 5.1, and have anticipation and jump segments join with continuous velocity. The problem is that the crouch ends up being very short in this case. So we cheat a little and start the crouch with 1.5 times the velocity of the initial jump velocity, $v_{c0} = 1.5 v_0$. The deceleration $a_c = 2 v_{c0} / t$ will result in the character being in the starting position after the crouch duration (this is derived from Equation 5.2). The join point of anticipation and jump velocity curves will have a discontinuity now, but the smoothing effect of the PID controller makes it unnoticeable, and now the crouch is deep enough to look good.

5.6.2.2 Jump phase

After the anticipation phase, the jump phase begins, as shown in frames (c) through (e) in Figure 5.7.

Here, we use equations 5.1 and 5.2 to compute the vertical position of the dancer's root transform. In addition, we use a sine function from 0 to π to raise the IK effectors of the dancer's feet, so that it looks like the dancer is slightly pulling her legs up during the jump. The sine term is multiplied with such constant height that this leg pull is not very noticeable with longer jumps, but in fast, very low jumps it helps to emphasize the motion more.



(a) Interpolate feet pos. (b) Rotate root transform (c) Target pose

Figure 5.8: Two alternative methods, (a) and (b), of animating the ground plane position of the dancer's feet to the target pose (c).

The trajectories of the feet in the ground plane are bit more involved than with the step animation. We could implement the jump in the same way, so that we just linearly interpolate the position of the both feet between the starting panel and the goal panel. This is shown in Figure 5.8(a). But with some jumps the trajectories of the two feet would intersect. Consider for example a jump where the dancer is standing on the down and up panels, and switches legs between the panels. This of course looks bad, but also causes singularity issues with the foot normal angle; the angle instantly changes by 180 degrees when the legs cross.

Another way of creating trajectories for the feet could be to move and rotate the character's root transform to the point that lies between the target panels, shown in Figure 5.8(b). We would obtain this point by computing the vector between target panels and find its midpoint. The target orientation of the root would be the orientation of this vector. If the distance between the initial panels and the target panels differs, the dancer also has to adjust the distance of her legs from the body. This method works, but is not very natural either. It looks as if the dancer is rotated something twisting her around from the torso – exactly what is happening.

The best results are achieved with a combination of both techniques. The root rotation is lerped between the starting rotation and the rotation of the feet normal in the end, and the root position is lerped between the starting point and the feet normal midpoint in the end. The positions of the feet, however, are a halfway lerp of positions calculated with both techniques: a straight lerp from the start point to end point contributes half of the foot's position, and the position in which the leg would be when rotated by the

root transform contributes the other half. With this, the trajectory doesn't suffer from singularity issues, the feet don't overlap in any given time, and the motion looks like being driven by the legs.

As noted in the animation guidelines (Section 2.5.5), the hips should lead the movement and then the legs should then follow. We can achieve this by defining that the root rotation is interpolated during the whole animation (including the anticipation phase), but rotation of the legs around the root is interpolated only during the actual jump move. What this then achieves is that rotation of the legs starts only after the dancer has propelled into air, but the body turns the whole time. For both interpolations, cubic ease out is used instead of lerp, to make the bulk of the rotation occur in the beginning.

We then still need to specify the rotation of the feet. The heels of both feet are rotated in the same way as with the step motion. The rotation of the feet in the ground plane follows the same principles too. Here the target rotation of the feet is a lerp between the feet normal rotation in the end, and initial rotation of the leg, but this time with 75%/25% ratio. In jumps it seems to look better in most cases that feet face more towards the jump direction, though this can lead to the "twins pose" discussed earlier.

We also lean down and forward with the shoulder IK effectors during the anticipation and the jump phase. This makes the anticipation of the jump more prominent. The degree of lean is constant though, and looks a bit off for jumps covering a longer distance. The degree of lean should be improved to be a function of the jumped distance as future work.

Note that during the jump the control of the vertical position of all the IK effectors is interpolated between the PID controllers and direct control. The jump starts with PID control, transitions into 100% direct control in the middle, and fades back into full PID control; a smooth curve is used for the transition. This is done to make the jump more responsive to control vertically. Ground plane coordinates still are fully controlled by PID control.

5.6.2.3 Follow-through phase

The follow-through phase occurs after the dancer has landed from the jump, shown in frames (f) and (g) in Figure 5.7.

In this phase, we just repeat the crouch we did at the anticipation phase, though the timing and height of the crouch are different. As mentioned, this phase may be cut off completely or be limited in duration, if the next move occurs shortly after, especially if the next move is a jump. The timings are tuned such that a situation where the dancer lands crouching, gets up, and crouches for anticipation of the next jump should not occur. Instead a single crouch is done in-between the jumps.

As future work some hand animation should be implemented to the jump. Now only procedural inertia resulted from PID control animates hands in any way during the jump.

5.6.2.4 Gallop jumps

A preliminary support for gallop jumps is built into the system. The gallop, as introduced in Section 2.4.3, is a step pattern where two notes for different panels are relatively close in timing, so that it makes sense to jump on the arrows so that the other leg lands shortly after the first leg on the later note. The jump routine can be specified which leg is the one arriving later and the duration between the hitting time of the legs. Then we calculate additional jump velocity for that leg that causes it to land later, using Equation 5.2, and sum an additional Equation 5.1 term to position of the leg.

5.7 Looking at the screen

As the dancer is trying to simulate a real play of the game, her eyes should be kept focused at the screen where the game would be displayed. In our implementation, we don't have a visible display device prop in the scene so that different viewing angles aren't obstructed, but the dancer still looks to the front where the display would be (see Figure 5.9).

The look-at IK solution is computed and applied after the full-body IK solution that drives the animation has been applied. In the solution such world rotation is computed for parts of the body that the part faces directly to the specified target point. The torso, head and eyes are considered. After a rotation for the body part is computed, we multiply the rotation with a weight from 0 to 1. The eyes should be facing the screen most directly, then the head less directly and the torso with the largest difference in rotation. We should then assign the weights in this order; we use full weight of 1 for the eyes, 0.8 for the head and 0.7 for the torso.

We should also limit rotations of the body parts to avoid them turning too much, when the character's hips face considerably away from the target. We do this with clamp weights for every part. If the clamp weight is 0, the body part can be turned the full 180 degrees away from its identity local rotation; if it is 1, the body part is not permitted to turn at all. We use the clamp weight of 0.7 for all body parts, meaning that their local rotation is limited to ± 126 degrees.

Special consideration should be given to what happens when the character spins around. In real play, the player would try to keep looking at the screen

Figure 5.9: Our dancer demonstrating a crossover pattern, while still looking at the front, where a display device would be located in a real gameplay situation.

as long as possible when doing the spin motion, and then quickly turn the torso and head to the opposite angle at around the 180 degree rotation. In our system, the 180 degree angle is a point of singularity where the look-at angle changes sign: if the rotation of the torso was clamped at 126 degree angle, now it is -126 degrees. This transition should be interpolated over time, using an ease out type curve.

5.8 Ponytail simulation

To add some secondary action (Section 2.5.10) to the character, a ponytail (braid) is simulated using the PhysX physics engine integrated into Unity. The setup of the ponytail should be applicable to any rigid body physics engine.

In the character skeleton the ponytail is rigged with four bone segments. In the physics engine side, each of the segments is encompassed by a capsule collision volume (see Figure 5.10), which are linked together with joints that limit their rotation, and completely prohibit twisting rotation. This chain is then linked to the head. The head is also equipped with a spherical collision volume, allowing the ponytail to collide with it. This volume is kinematic,

that is, it participates in collisions, but does not receive any collision response. The hair volumes are dynamic, meaning that they do receive the collision response, in this case when they collide against the head, and are also affected by gravity. At runtime, the rigid-body physics engine then runs the physics simulation in discrete steps, resolving collisions and applying gravity to the hair collision volumes.

However, there were several problems with our implementation. Initially we had a collision volume in the character's torso too, so that the ponytail would collide against the back. This would cause the hair collision volumes to swing on top of the shoulders of the character, and lie there until displaced by a fast move. Setting more strict rotation limits for the joints would fix this problem, but make the ponytail seem too rigid. This problem was mostly caused due to design of the character mesh and length of the ponytail, but we wanted to keep the design as it was pleasing otherwise, and making changes at that point would have been a lot of work.

In some cases it seemed that the ponytail could actually intersect and get stuck in the shoulders. This would occur especially when the character spun around fast. We were then forced to just remove the collision volume in the torso, making the ponytail mesh sometimes swing inside it, but this does not happen to such degree that it would be distracting.

To make the simulation really stable, we also had to make it run with 120 Hz frequency. This is pretty high; a developer from the games company Tri-Ace tells that all games they have developed run with 30 Hz, and no more 60 Hz should necessary to use [34]. The instability in our system is probably due to the fact that we render the dancer animation with variable timesteps, but the physics use fixed timesteps. Because the animation and physics updates interleave in various ways depending on the frame rate, the simulation is not too robust.

Another problem we had is that because the bone transforms are updated by the transform hierarchy in the character skeleton, and the joints for the physics engine are a separate concept, having the hair linked with bones actually causes the physics joints not to exert any forces to the hair segments. When the character moves, the hair should exhibit inertia and drag behind, being pulled by the tension force of the hair strands. But if the position of the hair is just determined by the parent transform, the physics engine will not need to apply any force to drag it along. Our workaround for this was to just disconnect the hair bones from the head, so the linkage is enforced solely by the physics joint. It is unclear whether this is the intended way this should be done.

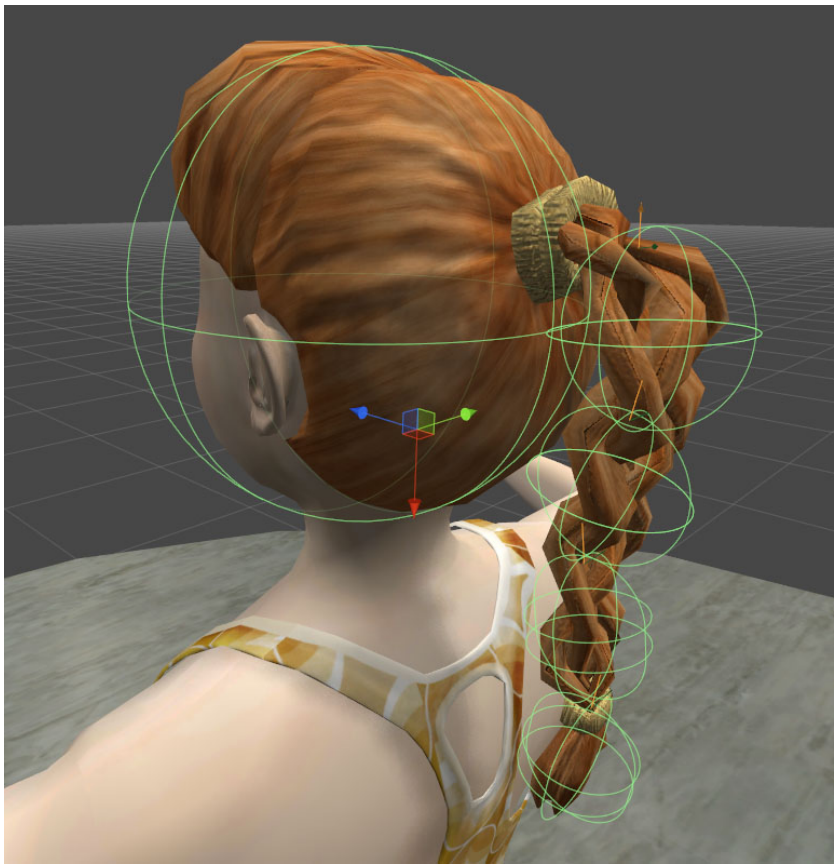


Figure 5.10: The collision volumes of our character's ponytail from close up

Chapter 6

Evaluation and discussion

In this chapter we will evaluate the quality of our dancer animation, and discuss what improvements could be made to the system as future work. A public survey and the author's personal experience with dance games will used for the assessment.

Figure 6.1: The main menu screen of the dancer demo, containing instructions for the demo and survey, and a selection screen for the included songs

6.1 Public survey

In the survey, we published a demo program of our dancer animation, along with three songs to test, in our company's website. Figure 6.1 shows the initial screen of the demo. The demo program was distributed for the Unity webplayer browser plugin, so it could be played directly from the web. The link to the page containing the demo and the survey was published in social media, IRC (Internet Relay Chat), and in discussion forums. These channels were related to either the dance game player community, or to Finnish game developers, so most if not all of the replies are from this demographic.

For people participating in the survey, we asked 10 questions regarding the quality of the animation. Each of the questions had the participant rate a specific quality of the animation. The answer options ranged from "very poor" to "very good" at a five point scale, barring a few questions which had context-specific descriptions of the scale. In total, 54 responses were given to the survey. Table 6.1 lists the questions and the mean values and standard deviations of the results. Detailed results and the survey form are available in Appendix A.

Question	**Mean**	**Std. dev.**
Did the animation look smooth	3.7	0.8
Did the animation look realistic and physics-based?	3.2	0.9
Did the dancer play similarly to a real player?	3.4	1.0
Rate the quality of step motions	4.0	0.8
Rate the quality of jump motions	3.0	1.1
Rate the quality of slow motion	3.8	0.9
Rate the quality of fast motion	3.2	0.9
Rate the quality of hand motions	3.1	1.3
Rate the quality of other secondary motions	3.1	1.1
Give an overall score to the animation quality	**3.5**	**0.8**

Table 6.1: Survey results summary

From the results, we can see that survey participants considered step motions and slow motion of the dancer to be the strong points of the demo, grading them with 4.0 and 3.8 points, on average. The animation was also thought to be fairly smooth, with 3.7 average points. Answers for each of these questions also had fairly low standard deviation, meaning that the participants mostly agreed on the answer.

Jump motions, hand motions and other secondary motions were considered to be the weak points of the animation, with averages of 3.0, 3.1 and

3.1 points. The participants agreed less in these questions, however, with standard deviations of 1.1, 1.3 and 1.1.

The participants gave an overall score of 3.5 to the animation, with a low standard deviation of 0.8. This means that the quality of the animation was thought to be between average and good.

We also asked each participant about their experience with dance games to see if familiarity with them correlated with the answers in some way. Out of the 54 total responses, only 4% (2 people) had not played at dance game at all. 13% had tried a dance game briefly, 24% had played on multiple occasions, and the majority, 37% had played quite a bit, but not competitively. 22% of the participants answered that they play at competitive, high levels of play.

If we only consider the last two groups, the people who have played quite a bit, or play at competitive level, and summarize their answers, we have the results in Table 6.2. The second column contains answer mean values for the experienced and competitive gamers combined, and the third column lists the same for competitive people only.

Question	**Experienced**	**Competitive**
Did the animation look smooth	3.9	4.2
Did the animation look realistic and physics-based?	3.3	4.0
Did the dancer play similarly to a real player?	3.7	3.5
Rate the quality of step motions	4.2	4.3
Rate the quality of jump motions	3.2	4.1
Rate the quality of slow motion	3.9	4.3
Rate the quality of fast motion	3.3	3.3
Rate the quality of hand motions	3.4	3.6
Rate the quality of other secondary motions	3.3	3.7
Give an overall score to the animation quality	**3.7**	**4.0**

Table 6.2: Survey results summary of experienced gamers' answers

All the scores increase when only the experienced dance game player demographic is considered, and the mean overall score is 3.7. Furthermore, if only competitive gamers are considered, some of the score averages are almost between good and very good, and the overall score average is 4.0. However, while these groups surely are better aware of what the dance game gameplay looks like, they might give extra points because they appreciate the idea of a tutorial character for the sport they like. So probably the mean scores of all the survey participants represent the overall quality of the system best.

The survey also had an additional feedback section, in which people could write free-form comments or questions. Over half of the participants wrote

additional comments to this section. Of the critique written here, most common was people noting that longer jumps looked floaty or non-physics-based, and that longer or heavier anticipation would be required for them. Many also noted that hands don't move much, and in reality the dancer would try to keep her balance by weight shifting with the hands and torso, and this doesn't happen here. One participant said that it was hard to connect the motion of the dancer to the stepchart while looking at them side by side.

Besides the critique, quite many participants said that they liked the idea of the tutor dancer, and the implementation of the system overall.

6.2 Personal assessment and future work

The survey received more than enough replies to have people notice all the obvious shortcomings in the animation. The limitations in hand animation are obvious, and at least more hand animation clips should added into a future version. Some kind procedural model of balancing weight with hands would probably be the most realistic-looking solution.

The problems in timing animation correctly with the Mecanim animation system in Unity3D should be resolved in some way, so that we could use full-body animation clips in combination with inverse kinematics to control the moves. This should enhance the quality of the criticized long jump animations, for example.

The limitations in the follow-through phase of the jump animation should be addressed. When the jump ends and the dancer bends her legs and crouches at the follow-through phase, if a next move is due shortly, it would be started during the follow-through. Our current implementation does not allow for this, but simultaneous action could be implemented as future work in the following way:

Do not directly control the height of the dancer's root position in the jump routine. Instead, just store the desired position in a field. Allow a new move to be started during the follow-through phase of the jump, and have it store its desired height of the root position in a field, too. The used height is then interpolated between the two values. If this implementation does not result in smooth enough motion, we could pre-calculate the desired height values from key points of the animation, and interpolate between them with a spline. The problem with this is, though, that we cannot then easily force the gravitational acceleration for the dancer in the actual jump motion.

In the gameplay screen, we currently use the side-by-side view, where the stepchart scrolls in the left, and dancer is in the right side of the screen. We should research if a scene, where the dancer looks at a television where the

gameplay is composited, would be more intuitive to follow. Such a scene was used for testing in early stages of the development, but was dropped from the later versions. The main problem with it was that the dancer easily obstructs the view to the television, as she will be standing between it and the camera. A bigger screen could perhaps be used, but it might look unrealistic.

As an added feature, we should definitely implement the double play mode support for the tutor dancer, too. Play using two dance pads is harder to follow than normal play, so the dancer could be very helpful for people trying to learn patterns in double mode stepcharts.

Overall, we are quite satisfied with the dancer implementation despite the shortcomings, and it seems that if they are addressed, this system could very well be put to real use.

Chapter 7

Conclusions

In this master's thesis, we designed and implemented a character that simulates the behavior of a real person playing 4-panel dance games. This character is used to both aid the player of a dance game to learn the patterns in a new song, or help the creator of those patterns, the stepartist, to validate their intended play.

To facilitate the design of the dancer, we first introduced the game mechanics of dance games, and stated what requirements the dancer should fulfill. Then we designed an artificial intelligence that reads a stepchart, and outputs the moves that should be performed to play the chart as efficiently as possible.

Once the prerequisite knowledge of computer animation techniques had been introduced, we proposed several designs for the dancer, and further described the final design we settled upon. This design uses a combination of inverse kinematics and PID controllers in producing procedural animation.

Finally, we evaluated the quality of the animation by publishing a demo of the system, and running a public survey. The demo and animation were generally well-received, and once some shortcomings in the motion are fixed, the dancer is quite likely ready for real use.

Bibliography

[1] Andamiro. Pump it up. http://piugame.com/. Accessed 13.5.2014.

[2] Ang, K. H., Chong, G., and Li, Y. PID control system analysis, design, and technology. *Control Systems Technology, IEEE Transactions on 13*, 4 (July 2005), 559–576.

[3] Aristidou, A., and Lasenby, J. FABRIK: A fast, iterative solver for the inverse kinematics problem. *Graphical Models 73*, 5 (2011), 243–260.

[4] Autodesk. FBX file format. http://www.autodesk.com/products/fbx/overview. Accessed 11.5.2014.

[5] Bartels, R., and Hardtke, I. Kinetics for key-frame interpolation, 1987.

[6] Bennett, S. Nicholas minorsky and the automatic steering of ships. *Control Systems Magazine, IEEE 4*, 4 (November 1984), 10–15.

[7] Buss, S. R. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation 17* (2004).

[8] Catmull, E., and Rom, R. A class of local interpolating splines. *Computer aided geometric design 74* (1974), 317–326.

[9] Crytek. Cryengine. http://cryengine.com/. Accessed 11.5.2014.

[10] Danford, C. e. a. Stepmania. http://www.stepmania.com/. Accessed 11.5.2014.

[11] Diankov, R. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[12] Epic Games. Unreal engine. `https://www.unrealengine.com/`. Accessed 11.5.2014.

[13] Faber, R. e. a. Where can i find a java to c# converter? `http://stackoverflow.com/questions/168837/how-should-i-convert-java-code-to-c-sharp-code`. A question in a Q&A-style discussion forum. Accessed 12.5.2014.

[14] Frijters, J. IKVM.NET. `http://www.ikvm.net/`. Accessed 12.5.2014.

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[16] Grochow, K., Martin, S. L., Hertzmann, A., and Popović, Z. Style-based inverse kinematics. *ACM Trans. Graph. 23*, 3 (Aug. 2004), 522–531.

[17] Harmonix. Dance central. `http://www.dancecentral.com/`. Accessed 13.5.2014.

[18] Hirvisalo, V. T-106.5300 embedded systems. `https://noppa.aalto.fi/noppa/kurssi/t-106.5300`. University course. Accessed 12.5.2014.

[19] id Tech. MD3 file format. `http://en.wikipedia.org/wiki/MD3_(file_format)`. Accessed 11.5.2014.

[20] Kavan, L. *Real-time Skeletal Animation*. PhD thesis, Czech Technical University, 2007.

[21] Khronos Group. Collada file format. `https://collada.org/`. Accessed 11.5.2014.

[22] Kochanek, D. H. U., and Bartels, R. H. Interpolating splines with local tension, continuity, and bias control. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 33–41.

[23] Konami. Dance dance revolution. `https://www.konami.com/ddr/`. Accessed 13.5.2014.

[24] Kovar, L., Gleicher, M., and Pighin, F. Motion graphs. *ACM Trans. Graph. 21*, 3 (July 2002), 473–482.

[25] Lang, P. Final IK, 2014. `http://www.root-motion.com/final-ik.html`. Accessed 23.4.2014.

[26] Lasseter, J. Principles of traditional animation applied to 3D computer animation. In *ACM Siggraph Computer Graphics* (1987), vol. 21, ACM, pp. 35–44.

[27] Lawrence, N. D. Gaussian process latent variable models for visualisation of high dimensional data. In *Nips* (2003), vol. 2, p. 5.

[28] Mizuguchi, M., Buchanan, J., and Calvert, T. Data driven motion transitions for interactive games. In *Eurographics 2001 Short Presentations* (2001), vol. 2, p. 6.

[29] NVIDIA. Physx. `http://www.geforce.com/hardware/technology/physx`. Accessed 18.5.2014.

[30] Parent, R. *Computer Animation: Algorithms and Techniques*, 3 ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[31] Penner, R. *Robert Penner's Programming Macromedia Flash MX*. McGraw-Hill, Inc., 2002.

[32] Runge, C. Über empirische funktionen und die interpolation zwischen äquidistanten ordinaten. *Zeitschrift für Mathematik und Physik 46*, 224-243 (1901), 20.

[33] Sitnik, A. Easing functions cheat sheet. `http://easings.net/`. Accessed 28.4.2014.

[34] Spiro, L. What's a reasonable [physics] timestep. `http://www.gamedev.net/topic/634778-whats-a-reasonable-timestep/#entry5003396`. Discussion forum posts: #2 and #4. Accessed 12.5.2014.

[35] Tehrani, K. A., and Mpanda, A. PID control theory. *Introduction to PID Controllers - Theory, Tuning and Application to Frontier Areas* (2012).

[36] Thomas, B., and Walt Disney Productions. *Walt Disney: the art of animation*. Golden Press, 1958.

[37] Thomas, F., and Johnston, O. *Disney Animation – The Illusion of Life*. Abbeville Press, New York, 1981.

[38] Torus Knot Software. Ogre3d. `http://www.ogre3d.org/`. Accessed 11.5.2014.

[39] Twigg, C. Catmull-rom splines. *Computer 41*, 6 (2003).

[40] Unity Technologies. Unity3d. `http://unity3d.com/`. Accessed 11.5.2014.

[41] Valve Software. Source engine. `http://source.valvesoftware.com/`. Accessed 11.5.2014.

[42] Vuylsteker, P. Computer graphics: Spline, 2004. `http://escience.anu.edu.au/lecture/cg/Spline/index.en.html`. Accessed 28.4.2014.

[43] Wang, L.-C., and Chen, C. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *Robotics and Automation, IEEE Transactions on 7*, 4 (Aug 1991), 489–499.

[44] Welman, C. *Inverse kinematics and geometric constraints for articulated figure manipulation*. PhD thesis, Simon Fraser University, 1993.

[45] Wikipedia. In the groove 2. `http://en.wikipedia.org/wiki/In_the_Groove_2`. Accessed 13.5.2014.

[46] Wikipedia. L-system. `http://en.wikipedia.org/wiki/L-system`. Accessed 11.5.2014.

[47] Williams, R. *The Animator's Survival Kit: A Manual of Methods, Principles and Formulas*. Faber and Faber, 2001.

[48] Young, H., Freedman, R., Sears, F., and Zemansky, M. *University Physics*, vol. 1. Pearson Education, 2006.

Appendix A

Survey form and detailed results

The form used for the survey and detailed summary of the results are included in the following pages. For each question, a graph showing the distribution of the answers is presented.

Edit this form

Dancer feedback

Once you have listened to all the songs, provide feedback of the dancer's animation here!

* Required

Background info

Which of the following best describes your experience with dance games? *

By dance games we mean games played with a dance pad, such as: Dance Dance Revolution (Dancing Stage), In the Groove, Stepmania, Pump It Up, iDance, Dance Tricks (beta). Your feedback is valuable even if you haven't played a dance game before.

- ○ I haven't played any dance game
- ○ I have tried a dance game briefly
- ○ I have played on multiple occasions
- ○ I have played or play quite a bit, but don't compete
- ○ I play at competitive, high levels of play

Quality of animation

10 short questions. Please take the time to answer all of them.

Did the animation look smooth? *

	1	2	3	4	5	
Not at all	○	○	○	○	○	Very smooth

Did the animation look realistic and physics-based? *

	1	2	3	4	5	
Artificial	○	○	○	○	○	Realistic

Did the dancer play similarly to a real player?

If you're not familiar with dance games, you can leave this unanswered.

	1	2	3	4	5	
Not at all	○	○	○	○	○	Very much like a real player

Rate the quality of step motions *

When stepping on a single arrow.

1	2	3	4	5

Very poor ○ ○ ○ ○ ○ Very good

Rate the quality of jump motions *
When jumping on to two arrows.

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Rate the quality of slow motion *
(Present in song K4)

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Rate the quality of fast motion *
(Present in songs Sushi Challenge and Lunar Ripple)

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Rate the quality of hand motions *
Does the character move her hands in a natural way when playing?

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Rate the quality of other secondary motions *
Such as movement of the body, shoulders, head and the ponytail when stepping and jumping

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Give an overall score to the animation quality *
Summarizing the previous points

1 2 3 4 5

Very poor ○ ○ ○ ○ ○ Very good

Hardware details

These are not mandatory, but please answer at least to the first question. Answering them all will

help me to optimize the program.

What was your average FPS during the demo?
This is visible in the upper left corner. With decent hardware this should cap to 60 FPS (your monitor refresh rate, unless your monitor supports a higher rate).

Are you using a desktop computer or a laptop?

- ○ Desktop
- ○ Laptop

What operating system are you on?

- ○ Windows
- ○ Mac OS X

What is the version of your operating system?
Such as: Windows 8, Windows 7, Windows Vista, Mac OS X 10.9

What is your CPU?
Such as: Intel Core i7-2600k. In Windows 7 you can find this in Computer (right click) -> Properties -> Processor.

What is your GPU?
Such as: NVIDIA GeForce GTX 560 Ti. In Windows 7 you can find this in Computer -> Properties -> Device Manager -> Display adapters.

Additional feedback

Freeform comments or questions
Write additional comments or questions for me here. These can be about the animation quality, technical problems, or something else.

Contact information for replying
If you want me to reply to your question, provide your e-mail, name or an IRC nick. The survey is processed anonymously whether you provide contact information or not, and the information is only used for the purposes of contacting you.

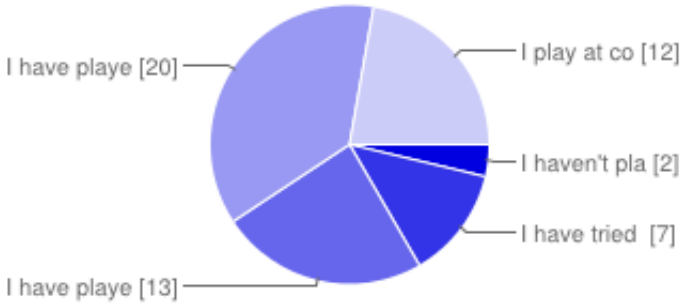
54 responses

View all responses Publish analytics

Summary

Background info

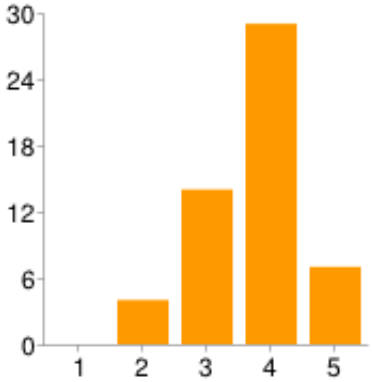
Which of the following best describes your experience with dance games?



I haven't played any dance game	**2**	4%
I have tried a dance game briefly	**7**	13%
I have played on multiple occasions	**13**	24%
I have played or play quite a bit, but don't compete	**20**	37%
I play at competitive, high levels of play	**12**	22%

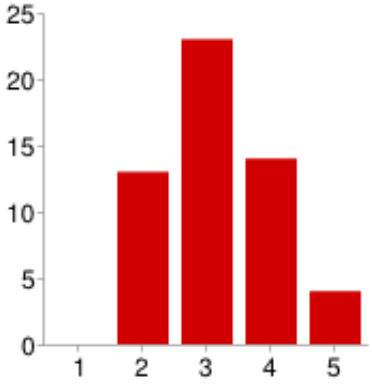
Quality of animation

Did the animation look smooth?



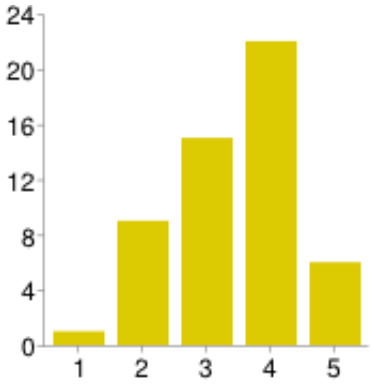
1	**0**	0%
2	**4**	7%
3	**14**	26%
4	**29**	54%
5	**7**	13%

Did the animation look realistic and physics-based?



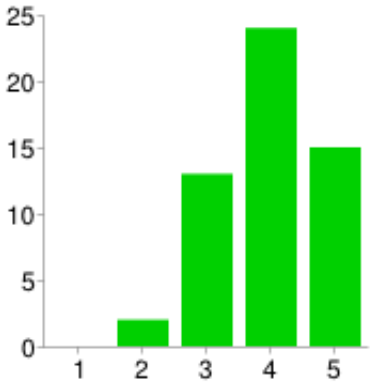
1	**0**	0%
2	**13**	24%
3	**23**	43%
4	**14**	26%
5	**4**	7%

Did the dancer play similarly to a real player?



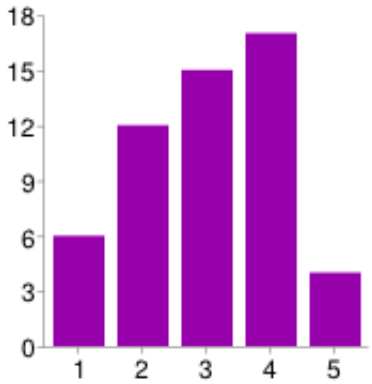
1	**1**	2%
2	**9**	17%
3	**15**	28%
4	**22**	42%
5	**6**	11%

Rate the quality of step motions



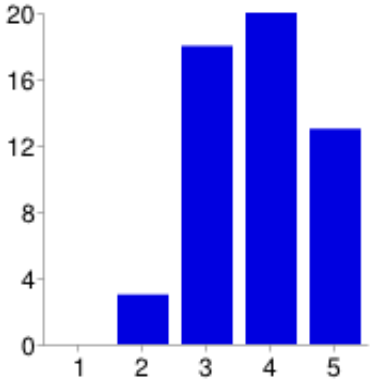
1	**0**	0%
2	**2**	4%
3	**13**	24%
4	**24**	44%
5	**15**	28%

Rate the quality of jump motions



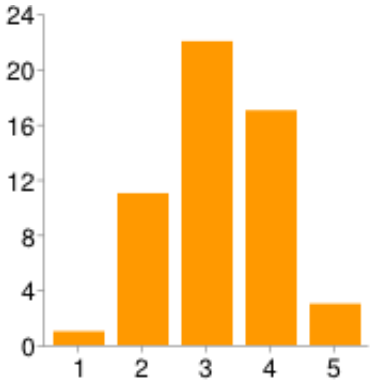
1	**6**	11%
2	**12**	22%
3	**15**	28%
4	**17**	31%
5	**4**	7%

Rate the quality of slow motion



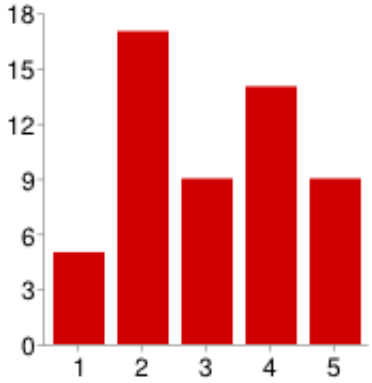
1	**0**	0%
2	**3**	6%
3	**18**	33%
4	**20**	37%
5	**13**	24%

Rate the quality of fast motion



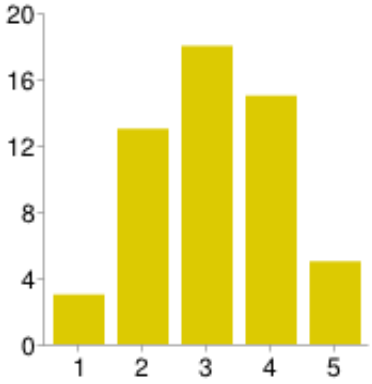
1	**1**	2%
2	**11**	20%
3	**22**	41%
4	**17**	31%
5	**3**	6%

Rate the quality of hand motions



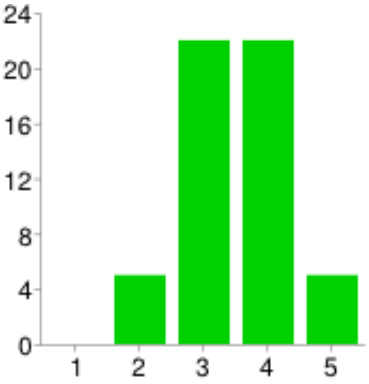
1	**5**	9%
2	**17**	31%
3	**9**	17%
4	**14**	26%
5	**9**	17%

Rate the quality of other secondary motions



1	**3**	6%
2	**13**	24%
3	**18**	33%
4	**15**	28%
5	**5**	9%

Give an overall score to the animation quality



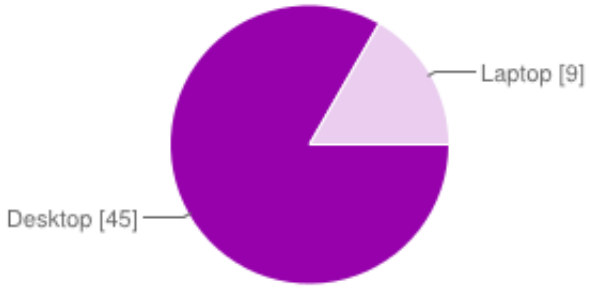
1	**0**	0%
2	**5**	9%
3	**22**	41%
4	**22**	41%
5	**5**	9%

Hardware details

What was your average FPS during the demo?

38 40 58.8 59.9 28 60 FPS 20 ~100 59,8 59,9 120 ~50 59
58 60 48 144 around 60 142 50

Are you using a desktop computer or a laptop?



Desktop	**45**	83%
Laptop	**9**	17%

What operating system are you on?