# Using Machine Translation to Convert Between Difficulties in Rhythm Games

**Kevin Gold**
Rochester Institute of Technology

**Alex Olivier**
Wellesley College

### Abstract

A method is presented for converting between *Guitar Hero* difficulty levels by treating the problem as one of machine translation, with the different difficulties as different "languages." The *Guitar Hero I* and *II* discs provide aligned corpora with which to train bigram-based language models and translation models. Given an Expert sequence, the model can create sequences of Hard, Medium, or Easy difficulty that retain the feel of the original, while obeying heuristics typical of those difficulties. Training the model requires a single pass through the corpus, while translation is quadratic in the length of the Expert sequence. The method outperforms a recurrent neural network in producing sequences that match the hand-designed levels. The method may make it easier for amateurs to produce content for the Rock Band Network.

## Introduction

With the new Rock Band Network program, rhythm game maker Harmonix has given the general public access to the tools necessary to author new *Rock Band* content. But these amateur authors may have little interest or experience in hand-coding note sequences that are easy enough for novice players. The game has four different difficulty settings, and the exercise of making four passes through the song to decide which notes to remove or shift is tedious at best. Moreover, the independent author may not take the time to playtest the song with players of various skill levels. The present paper provides a method by which independent authors might author a single Expert track, and then "translate" the track to lower difficulty levels, using a machine translation algorithm trained on Harmonix's earlier games, *Guitar Hero I* and *II*.

Rhythm games give players the illusion of playing actual instruments by playing recordings of electric guitar, bass, and drums as players hit the color-coded instrument keys in time to the music. The mapping of actual musical notes to colored gems is many-to-one; hitting a single note often results in a whole chord or arpeggio being played. Players are required to hit fewer notes on easier difficulty levels. Experienced players typically play on the "Expert" difficulty level, which tries to reproduce the rhythms of the original notes as faithfully as possible and requires the player to slide the hand up and down the plastic guitar controller. Casual players typically play on "Easy" or "Medium," which typically go no faster than eighth notes and do not require moving the hand (Figure 1). There are only a few hard-and-fast rules for authoring on the easier levels – no chords and only 3 note colors on Easy, only 4 note colors on Medium – and the rest is left to the author's taste, balancing difficulty with capturing the feel of the music.



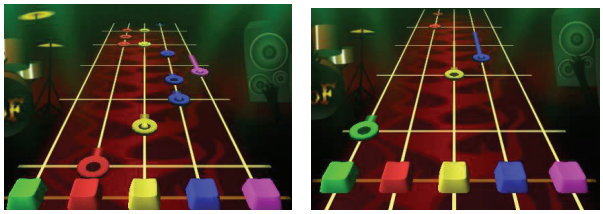

Figure 1: Left: A sample screenshot from the *Guitar Hero II* Expert version of "Search and Destroy" in the *Frets on Fire* emulator. Right: The official Medium version of the same segment. Notes are omitted to make the song easier, while retaining the melodic feel.

The key idea in the present paper is that the different difficulty levels can be treated as different "languages," and the existing songs on the official *Guitar Hero* discs provide an aligned corpus with which to do machine translation. The probabilistic translation incorporates factors such as the probability of transitions between chords, the probability that a note will be omitted if it does not fall on the beat, and the probability that a big jump in the Expert sequence will be translated into a smaller jump on the easier difficulty levels.

### Related Work

The idea of "machine translation for game content" is new to this paper, but there is ample prior work on generating game content that achieves some target difficulty. On the game side, there have been several approaches to automatically generating or modifying game content to achieve an optimal difficulty. The idea of creating challenges that are both interesting and of the appropriate difficulty was pursued using

neural networks and evolutionary algorithms in a *Pac-Man* game, with feedback provided by built-in metrics for diversity and challenge (Yannakakis and Hallam 2004). A neural network has been trained to predict player fun, challenge, and frustration in *Super Mario Bros.* levels, based on the number, width, and variation of pits, and whether the player needed to change direction (Pedersen, Togelius, and Yannakakis 2009). A dynamic difficulty adjustment system for *Half-Life* used concepts from economics to identify when the player's inventory of resources such as health or ammunition were encountering shortfalls, which could be used to supply such resources dynamically within a level (Hunicke and Chapman 2004). Most famously, the recent game *Left 4 Dead* placed zombies dynamically in response to a model of the player's level of arousal (Booth 2009). These dynamic approaches to difficulty balancing can make sense when modeling a living, changing environment, but there is also something to be said for creating a set of static challenges which the player can learn to master. In particular, the rehearsal of content that remains the same from playthrough to playthrough more closely resembles learning to master a musical piece; this is the reason for the existence of musical games' "rehearsal modes," in which the player can practice specific difficult passages at slower tempos.

The idea of probabilistically replicating musical events from an established corpus to emulate a particular composer was used in David Cope's famous EMI system (Cope 1990). A recurrent neural network was trained on Sonny Rollins' saxophone improvisations to reproduce his style (Franklin 2001). The present work might be seen as similar to these, in that it is reproducing the work of human "composers" on the easier difficulty settings, but also bears some similarity to the work on AI for musical performance, since the system is choosing which notes are most important to express. Approaches to learning from established performances have included recurrent neural networks (Bresin 1998) and case-based reasoning (Arcos and de Mantaras 2001). The challenges of the present work are somewhat different from any of these; the highly constrained choices for the algorithm at once greatly reduce the search space, while at the same time forcing more and different tradeoffs, since with only three notes, the sequence often runs out of room to ascend or descend.

The following work is novel in that it is the first to apply machine translation techniques to a non-linguistic corpus of video game content. It is also the first to specifically address the problem of automatically generating rhythm game note content. While the model could readily incorporate audio track information into the decision process by conditioning its probabilities on volume and inferred pitch, the current work omits this step, so as to keep the focus on the idea of simply translating game content from one "language" to another.

## Methods

The techniques used here are borrowed from statistical natural language processing; see (Manning and Schütze 1999) for a more thorough introduction. The "languages" that we are translating between are the different *Guitar Hero* difficulty levels. We wish to maximize the probability of the easy message E, given the expert message X. Using Bayes' rule, this becomes:

$$P(E|X) \propto P(X|E)P(E) \tag{1}$$

$P(X|E)$ is given by the *translation model*, and $P(E)$ comes from the *language model*. These two factors provide the balance between faithfulness to the original song (translation model) and faithfulness to the target difficulty (language model).

The model can be further broken down into note transitions, note translations, and timing. Each component will be described in detail before the search algorithm is described.

### Language model: Note and chord transitions

The simplest component of the model is the language model for the colors used in notes and chords in the target language. This is simply a bigram model, in which the probability of each note or chord depends only on its predecessor:

$$P(E) = \prod_{i=1}^{|E|} P(E_i = c'|E_{i-1} = c) \tag{2}$$

Though there are technically $2^5 - 1 = 31$ different possible chord or note transitions, in practice, no difficulty level lower than Expert uses more than two notes in a chord. Thus, to represent this component of the model, a $15 \times 15$ table $T$ is built for all possible transitions between notes or 2-note chords.

The language model ensures that the ease of transitions between notes and chords is appropriate for the difficulty level, and that more difficult notes and chords are less frequent for easier difficulty levels.

### Translation model: Note movement and chord translation

The overall translation probability is the product $P(X|E) = P_c(X|E)P_\chi(X|E)$, where $P_c(X|E)$ is the probability of a particular color translation for the melody, and $P_\chi(X|E)$ is the probability of a particular translation of the supporting chords.

The primary goal in translating the notes is to ensure that notes maintain their direction of movement as much as possible. To this end, pairs of notes on the expert difficulty are matched to pairs of notes on the easy difficulty:

$$P_c(X|E) = \prod_{i=1}^{|E|} P(X_i = c', X_{i-1} = c|E_i = \gamma', E_{i-1} = \gamma) \tag{3}$$

This equation assumes that the expert and easy sequence are of the same length, which is never actually true; but the "Timing" section below will deal with the problem of dropping notes.

Unlike the language model, the translation model does not keep track of every possible set of chord transitions; since the size of this probability table grows as the fourth power

of the number of symbols, using even just two-note chords would give a $15^4$ = 50,625 entry table, which we did not think the corpus could support. Instead, these translation probabilities use just the top note of each chord, producing a $5^4 = 625$ entry table.

Chord types are translated independently of color:

$$P_\chi(X|E) = \prod_{i=0}^{|E|} P(X.chord_i = \chi'|E.chord_i = \chi) \quad (4)$$

The chord types here are classified by their intervals, and not the specific notes; the types are singleton, 1-2, 1-3, 1-4, and 1-5, with 3-note chords lumped with 1-5 chords as difficult, Expert-level only chords. The language model will take care of removing awkward chord transitions in the target language; this term simply ensures that chords retain roughly the same feel as the expert chords.

Notice that in all these cases, the conditional probabilities are perhaps the reverse of what is intuitive, because the use of Bayes' rule to incorporate the language model forced us to turn around the probabilities; these are the probabilities of the expert sequence given the easy sequence, when the ultimate goal is to produce an optimal easy sequence.

## Timing

Easier sequences tend to drop notes and chords that occur in rapid succession or do not fall on the beat. While it might seem natural to treat the "null note" as just another symbol in the above models, this would make the bigram models less helpful than they could be, since the color of a note depends heavily on the color of the previous note, regardless of how long ago it was played.

Instead, the choice of which notes to drop is performed independently before color translation. The chance of dropping a note depends on how long it has been since the last easy note, as well as whether the note falls on the beat. Let K be the sequence of length $|X|$ that is 1 if the corresponding expert note is kept for translation, and 0 if it is dropped. The probability of keeping a note depends on the time since the last voiced note on the easy sequence ($T(K_{1...i})$ in Equation 5), and whether the note under consideration is on the beat ($b(X_i)$):

$$P(K) = \prod_{i=1}^{|X|} P(K_i|T(K_{1...i}), b(X_i)) \quad (5)$$

The *Guitar Hero I* and *II* corpora consistently use 480 midi ticks to a beat, which makes the problem of calculating and storing these probabilities from their frequency in the training corpus relatively straightforward. However, sometimes a particular time between notes is never observed in the training corpus. In such cases, the algorithm treats the probability of keeping the note as $e^{-4}$, and the probability of dropping it as $e^{-b}$, where $b$ is the number of beats since the last note. While these do not sum to 1, this has the desired effect of making dropping the note more likely than keeping it for uncommon intervals less than a measure long (since they are probably awkward), while exponentially decreasing the likelihood of dropping a note after a long delay (since the player is probably getting restless).

If a note is kept, its duration is copied directly from the Expert sequence.

## Tree search algorithms

Building the probability tables above requires only a linear time pass through the training corpus, a huge advantage in computation time over neural network training; but searching for an optimal easy sequence given an expert sequence requires a bit more finesse. The brute force approach of calculating the probabilities of all $16^{|X|}$ sequences of zero, one, or two notes is not feasible for expert sequences of 1000 notes, so it is rather important to use a polynomial time algorithm. The methods we shall describe below take $O(|X|^2)$ steps to choose the notes to keep, and $O(|E|^2)$ steps to choose the optimal chords and notes.

The search algorithm for the optimal timing relies on the following insight. Suppose $P(K_{1...j}) > P(K'_{1...j})$ for two sequences $K$ and $K'$, where $P(K_{1...j})$ is the probability given by multiplying the first $j$ terms of $P(K)$. If $K_j = K'_j = 1$ (both subsequences end with the same note being kept), then $K'$ will never have a chance to "catch up" to $K$, because any sequence that could be added to $K'$ could be added to $K$ to produce a higher probability. We can say that $K'$ is *dominated* by $K$. In iterating through $X_i$ to produce new possible sequences, the search does not need to grow exponentially at all, because for any set of sequences ending with a note kept at time $i$, one will dominate all the others.

Thus, even though there are $2^i$ possible sequences of $K$ up to note $i$, there are actually only $i$ sequences that are "viable" up to that point – one for each possible time of last kept note. The timing algorithm, Algorithm 1, makes use of this fact by building a tree of possible sequences. Each node in the tree corresponds to a kept expert note, and each path from the root to a leaf or interior node corresponds to a note sequence. Each node also stores the log probability of the sequence that leads to that node. (Log probabilities are used to prevent underflow.) When incorporating a new note from the expert sequence $X$, it only needs to be added in one place on the tree, where the resulting sequence's probability is higher than anywhere else, because one sequence ending in that note dominates all the others. All other nodes on the tree are updated with the conditional probability of "no note," because they can only remain viable by not adding the note there; they represent the optimal subsequences with no notes kept after their own. When the algorithm finishes processing the sequence $X$, the tree will have exactly $|X|$ nodes, with the path to each node giving the optimal expert note sequence that ends with that node's note as the last note of the song. It is then a simple linear time search to find the node that has the highest log probability, and return the notes on the path from the root to that node.

The time required to build the timing tree is $O(|X|^2)$, since each note in $X$ must be tried against each previous note's node (including interior nodes) to find the sequence

**Algorithm 1** The algorithm for construction of the timing tree.

```
for i = 1 to |X| do
  candidateList ← {root}
  bestLogProb ← −∞
  bestParent ← root
  while candidateList ≠ ∅ do
    n ← candidateList.pop()
    lp ← n.lp + lgProbNoteOn(X_i.time, n.time)
    if lp > bestLogProb then
      bestParent ← n
      bestLogProb ← lp
    end if
    candidateList.push(candidate.children)
  end while
  for all nodes n in tree do
    n.lp ← n.lp + lgProbDrop(X_i.time, n.time)
  end for
  bestParent.children.push(newnode(X_i, bestLogProb))
end for
return pathToMaxLogProb(root)
```

that ends in $X_i$ that dominates the others.

The sequence $K$ of boolean values can then be used to filter the expert sequence $X$ into a shorter sequence, $X'$, that is the length of the target easy sequence $E$.

Once the optimal selection of notes to translate has been found, the same approach can be used to build a second tree, this time for colors and chords. The same insight that justified Algorithm 1 applies – namely, that for any set of sequences ending in a particular note or chord, one will dominate all the others, because any future subsequence added to one of the dominated subsequences could be added to the dominating subsequence to produce a higher probability. When translating the chord at time $i$, it is only necessary to add one node to the tree for each possible one or two-note translation, where the resulting probability is highest. Thus, the tree gains at most 15 nodes for each step: one for each single note or two-note translation. The full algorithm is omitted for space, but is similar to Algorithm 1. The primary difference, besides using the language model and translation model probabilities instead of the timing probabilities, is that nodes are no longer allowed to be attached anywhere, but must be attached to a node representing the previous note in the sequence. (This means some branches of the tree will become dead; a further optimization could prune them so that they are not traversed during the search.)

The algorithm can be further sped up if one assumes that chords are only ever translated to chords of their own size or smaller, so that a maximum of 5 nodes are added for singleton expert notes. Notes and chords that are prohibited on the target difficulty level are not considered at all.

By a similar logic to the first tree, building the tree requires $O(|E|^2)$ time, and finding the optimal sequence once it is built is linear in $|E|$.

## Experiments

The free open source software *Frets on Fire* (Kyöstilä et al. 2006) was used to extract the note sequences from the *Guitar Hero I* and *II* Playstation 2 discs. A model for each difficulty level was built using a 21,536 note corpus of 25 songs from *Guitar Hero I* and *Guitar Hero II*. The algorithm then translated six songs that were not in the training corpus.

For a baseline comparison, a simple Elman recurrent neural network (Elman 1990) was trained on the same corpus of 25 songs, using the PyNeurgen library (Smiley 2008). The output layer consisted of 5 nodes, one for each color, representing the possibility of producing any combination of notes or no note at all for each chord in the Expert sequence. (The network was trained with +1 representing a note and -1 representing no note, and the test output was thresholded at $> 0.2$ to produce a note.) The hidden layer contained 6 nodes. The input layer consisted of 13 nodes – 5 representing the 5 notes, 6 that were copies of the hidden layer from the previous time step, and two inputs to represent timing. One timing input represented how strongly the input note or chord fell on the beat: 1 for being on the beat, 0.5 for a syncopated eighth note, 0 for a triplet, -0.5 for a sixteenth note, and -1 for all other notes. The other timing input represented the number of beats $b$ that had passed since the last Expert note, and was a floating point input of $\min(b-1, 1)$ to give a range between -1 and 1. This recurrent neural network was trained for 50 epochs on the training corpus, with a learning rate of 0.3, resulting in a mean-squared error that changed only 0.00002 between epochs 49 and 50.

In order to assess our model's songs qualitatively, five subjects of varying skill level were recruited to play-test the sequences produced by our model. Subjects were asked to play two versions each of two different songs using *Frets on Fire*. In one comparison, subjects played an Easy song produced by RNN and an Easy song produced by our model. For the other comparison, subjects played an original Guitar Hero song of the difficulty of their choice, and our model's version of the same song at the same difficulty. Subjects were given a practice song before beginning the experiment, and were given each comparison in balanced presentation order. They were then asked to provide feedback on perceived difficulty, perceived entertainment value, and how well the notes played matched the melody of the song, and then select the song they preferred (if either).

## Results

Figure 2 compares the output of our algorithm and the recurrent neural network to the note sequences on the original *Guitar Hero* discs for the Hard, Medium, and Easy difficulties.

The algorithm differences from the "official" versions can be broken down into two categories: Timing and Color/Chord. For timing, we can define precision as the proportion of notes produced that fell on the same beat as a note in the official sequence, and recall as the number of notes in the official version with corresponding notes at the same time in the algorithm output. (Note that "recall" is being used in a technical IR sense; the algorithm never saw the

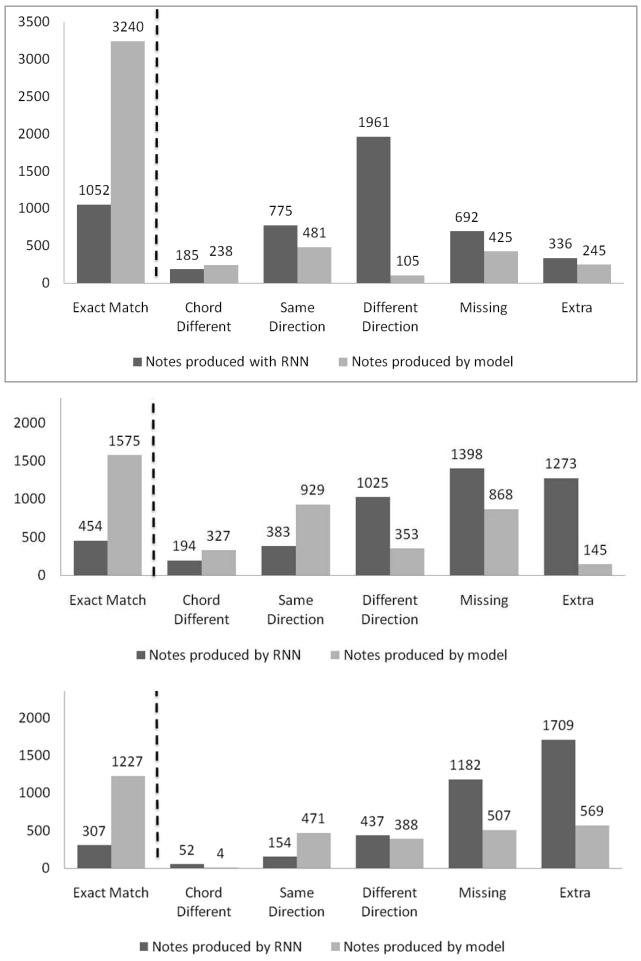


Figure 2: Comparing the algorithm and RNN sequence output to the original songs on the *Guitar Hero* discs, for the six songs in the test set. Differences (right side of dotted line) are arranged from left to right according to decreasing desirability. From top to bottom: Hard, Medium, and Easy.

easy versions of the target songs.) Under this metric, timing precision was 0.88, 0.64, and 0.70 for Hard, Medium, and Easy for our algorithm, compared to 0.60, 0.24, and 0.21 for the recurrent neural network. Timing recall was 0.93, 0.92, and 0.68 for our algorithm, compared to 0.75, 0.26, and 0.15 for the RNN. Thus, our algorithm was superior in precision and recall across all difficulties.

For the notes and chords that match in time, we can directly compare their colors, treating notes that move in the same direction and share the same chords as the original as correct responses. (Our algorithm was incapable of producing difficulty-inappropriate colors, so exact color is assumed not to matter much.) Color/Chord accuracy is then 91%, 78%, and 81% for our algorithm on Hard, Medium, and Easy, compared to 46%, 41%, and 49% for the RNN.

Chi-square tests confirm that our model is significantly more likely than the RNN to produce exact matches ($p << 0.0001$ for each difficulty).


Figure 3: The machine-translated version of the same passage as Figure 1, translating from Expert to Medium using a corpus that did not include the target song. Though different from the hand-designed "ground-truth," the song is comparable in difficulty while actually capturing more of the melodic movement.

Figure 3 shows some sample output for our algorithm on the Medium difficulty for the same passage as Figure 1. The figure demonstrates that being different from the ground truth does not necessarily make the sequence bad. If anything, the illustrated sequence captures more of the melodic movement than the official version, while remaining similar in difficulty; it contains more notes, but does not use the pinky to hit the blue key.

Though leave-one-out cross-validation was too computationally expensive for the RNN to be feasible over the large corpus, it was run on all 31 songs for our algorithm to ensure that the results were not an artifact of the particular songs chosen. All measures were within 3% of the reported values for the 6 target song case, except timing recall on Hard (85%) and Medium (87%). The full results are omitted for space.

Out of five participants, four preferred our model's sequence to the sequence produced by the RNN, while one subject had no preference. The subject that preferred neither version remarked that she felt unprepared for the difficulty of either song, which may explain her indifference. When asked to compare a sequence produced by our model with the original Guitar Hero sequence, four subjects reported that the quality of the sequence produced by our model was negligibly different from that of the official Guitar Hero sequence. Only one subject voiced a strong preference for the official sequence; this was on "Rock This Town," a fast-tempo song in which the model had included quarter notes that were too fast for the inexperienced subject to reliably play (see conclusions).

Figure 4 shows the "Percent correct" for each subject on the original version and the model-produced version of their chosen song and difficulty.

## Conclusions

Given the novelty of the approach, the authors were somewhat surprised at its efficacy. Designing the model to be descriptive but not too sparse was a careful balancing act, but

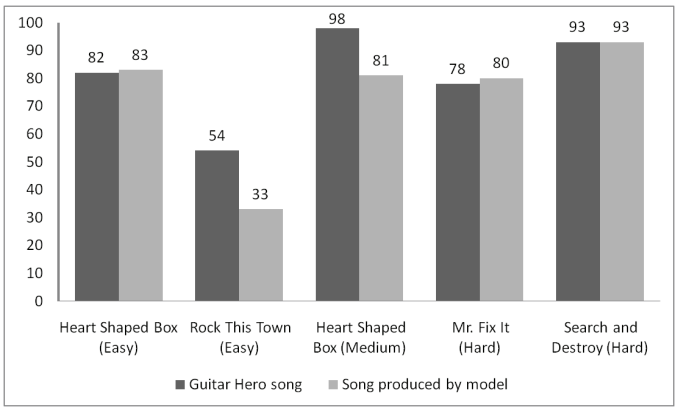


Figure 4: Subjects' percent correct on their chosen songs and difficulties, for the original song and the model-produced song.

the corpus proved to be large enough that chords and conditioning on the beat were added late in development with no problem. The primary stumbling block during development was how to account for a varying number of note events; our early approaches suffered from a bias toward short sequences. Once the timing step and color/chord step were separated, and the number of events in the timing equation fixed, the algorithm produced reasonable sequences that were relatively satisfying to play.

There are two directions in which this work could be pursued. One is to incorporate more music-specific information to achieve a better translation. The most noticeable errors in the current approach occur when a sustained note is omitted, often because it occurs off the beat. The algorithm could probably achieve better performance if the timing step took into account the duration of a note, in deciding whether to keep it. The algorithm also does not currently make use of any information at all in the accompanying sound file; it would be a natural next step to add the relative volume or energy of a note to the decision process. Tempo probably should also play a role in the timing algorithm; the results currently vary most from the official version on fast songs like "Rock This Town," since the algorithm does not take into account the fact that notes are significantly more difficult when the tempo is sped up. To achieve the best performance, these additional factors will have to be weighed against the curse of dimensionality that they introduce; some may not be worth the additional sparsity in the data.

It would be interesting to see whether a model with more long-distance structure, such as a probabilistic context-free grammar (PCFG), could achieve better performance than the bigram model presented here. The current model does not necessarily use a consistent sequence for a repeated passage, which goes against the Harmonix authoring guidelines. If simple memoization of previously translated subsequences achieves this effect, more complicated approaches may be unnecessary.

Musical structure aside, the second interesting possibility is applying machine translation to other kinds of game content. It is common for fan-made content for first-person shooter, real-time strategy, and roleplaying games to be adapted to work with a game's sequel when it comes out; but finding equivalent units and terrain types must be done by hand, and much content remains "untranslated." Given some examples, could machine translation convert *Starcraft* scenarios to *Starcraft II* scenarios, or *Doom* WADs to *Doom 3* content, or *Neverwinter Nights* content to *Neverwinter Nights 2*? Translation of a linear genre, such as the sidescroller or shooter-on-rails, should be very similar to this work, while translating 2D and 3D "languages" would pose an interesting research problem.

## Acknowledgments

This work was supported by the Norma Wilentz Hess fellowship of Wellesley College.

## References

Arcos, J. L., and de Mantaras, R. L. 2001. An interactive case-based reasoning approach for generating expressive music. *Applied Intelligence* 14(1):115–129.

Booth, M. 2009. The AI systems of *Left 4 Dead*. *AIIDE '09* Invited Talk(October 15, 2009).

Bresin, R. 1998. Artificial neural networks-based models for automatic performance of musical scores. *Journal of New Music Research* 27(3):239–270.

Cope, D. 1990. Pattern matching as an engine for the computer simulation of musical style. In *Proceedings of the 1990 International Computer Music Conference*, 288–291. San Francisco, CA: International Computer Music Association.

Elman, J. 1990. Finding structure in time. *Cognitive Science* 14:179–211.

Franklin, J. A. 2001. Multi-phase learning for jazz improvisation and interaction. In *Eighth Biennial Symposium on Art and Technology*.

Hunicke, R., and Chapman, V. 2004. AI for dynamic difficulty adjustment in games. In *Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*.

Kyöstilä, S.; Inkilä, T.; Kerttula, J.; and Korkiakoski, K. 2006. Frets on fire. http://fretsonfire.sourceforge.net.

Manning, C. D., and Schütze, H. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Pedersen, C.; Togelius, J.; and Yannakakis, G. N. 2009. Modeling player experience in *Super Mario Bros*. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*.

Smiley, D. 2008. Pyneurgen neural network library. http://pyneurgen.sourceforge.net.

Yannakakis, G. N., and Hallam, J. 2004. Evolving opponents for interesting interactive computer games. In *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior '04: From Animals to Animats 8*, 499–508. MIT Press.