# Applications of Planning and Procedural Music: A Designed Musical Experience

Brillan Morgan and Adam De Broeck

## 1 Introduction

Can a planning AI architecture predict player behavior? Jeff Orkin posed this question at the Game Developers Conference in 2006. We were motivated to explore this novel idea in tandem with the concept of creating a more granular musical interpretation of feedback in relation to player prediction. Therefore, our aim was to create an experimental proof of concept for the pipeline of predicted player actions to audible player guidance. To that end, we created a system for procedurally generated music to supplement an AI planning architecture.

For choice of planning architecture, we opted for a hierarchical approach since it more closely resembles the way that humans solve problems. Accordingly, for the PCG music, we decided to take a top-down approach to more favorably riff on existing input music, rather than generate music from scratch. This is because generating music from scratch is more constraint-based and does not give enough control to musicians and composers.

## 2 Planning AI Architectures and Predicting Player Behavior

In the early 2000s, existing AI architectures such as Finite State Machines (FSMs) were limiting the ability of game developers to build robust, scalable AI. Several solutions were developed independently to address this need. Damian Isla developed the Behavior Tree architecture for Halo 2. Jeff Orkin based his GOAP planning architecture on the STRIPS planning system from academia and was the first to use planning in a video game [Orkin 06]. Orkin's work with GOAP was the precursor for both classical and hierarchical planning architectures used in various games over the years.

### *2.1 Differences Between Classical and Hierarchical Planning*

Both GOAP and STRIPS are considered "classical" planning architectures. They run a search to find a sequence of actions that change the initial world state to the desired goal world state. In contrast, a Hierarchical Task Network (HTN) starts with a compound root task, matches it to a suitable method, decomposes the method's tasks recursively, and gives a sequence of directly actionable steps that are equivalent to the root task.

While the search techniques of classical planning and hierarchical planning differ, they ultimately both yield a sequence of actions that satisfy the given goal or accomplish the root task. They operate on an abstraction of the game world called "world state." Both classical and hierarchical planning start with the parameters of the initial world state. They validate the preconditions of actions against the current modelled world state and maintain

this model by applying the effects of planned actions to the world state model. This is a notable difference between planning AI architecture and other architectures such as Behavior Trees. Planning can consider the compound expected effects of planned actions when choosing further actions [Humphreys 13].

### *2.2 Our Puzzle Demonstration App*

For our exploration of predicting player behavior with planning, we favored the hierarchical planning approach, as it has more similarities with the way that humans solve complex problems: by breaking them down into smaller, more manageable parts.

To demonstrate HTN planning in action and explore its player-predicting potential, we devised a simple top-down puzzle where the player moves an agent through a series of rooms with the goal of reaching the designated exit. A series of locked doors block the way through the puzzle. Each door has a color and matches a corresponding pressure plate. When the player moves the agent over a pressure plate, the door of the corresponding color unlocks and opens. If the player opens an appropriate sequence of doors, the agent can be moved to the exit and the puzzle is solved.

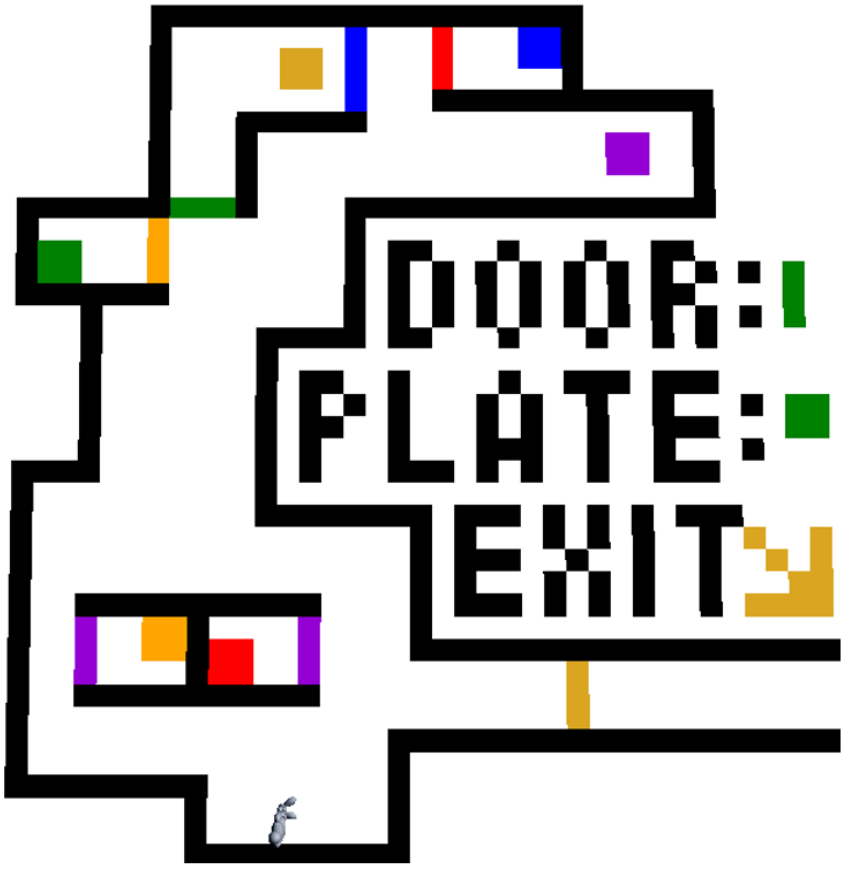


Figure 2.1 The initial state of the puzzle of our demo project

### *2.3 How HTN Planning Works*

HTN planning operates on a stack of tasks, starting with just one task: the root task. For our puzzle, the root task for the planner is ReachExit. There are two types of tasks: compound tasks and primitive tasks. Compound tasks, like ReachExit are not directly actionable by an agent, so they cannot be added to a final plan. Compound tasks must be decomposed into a series of more granular tasks. A compound task might be decomposed into only primitive tasks, only compound tasks or some combination of both. To decompose a compound task, the planner searches the list of available methods to find a match for the task at hand. The preconditions of each method are compared against the current world state until a method is found for which all preconditions are met.

Next, the decomposed composite task is popped from the task stack and replaced with the tasks specified by the selected method. This is a direct replacement of the popped task with tasks that, taken together, are equivalent to that popped task. Note that when a composite task is decomposed, neither the world state model nor the plan is modified, only the task stack.

A primitive task is a task that is directly actionable by an agent; it also cannot be decomposed into more granular tasks. In our puzzle demo, all primitive tasks are MoveTo actions that have a parameter of which grid space the agent should move to. When the planner pops a primitive task from the task stack, it does not push anything back onto the task stack. Instead, the effects of the action are applied to the world state model and the action is appended to the end of the plan.

If the planner empties the task stack, this means that the plan contains exactly the primitive tasks that are equivalent to the original root task. In our puzzle demo, a finished plan is made of only MoveTo actions that, when executed by the plan runner, will move the agent from its initial position to plates and through doors until finally arriving at the exit.

### *2.4 Planning for AI Agent Control vs. Planning for Player Prediction*

In our demo, at startup, the agent is under direct control of the player. Every time the agent moves from one grid square to another while under player control, a re-plan is initiated, and the resulting plan is printed to the console window. We argue that this plan is a plausible prediction of player behavior. Under what conditions does our planner effectively predict player behavior? When does it not?

The planner and world state model that we designed finds one solution to the puzzle that minimizes the number of doors the player must open to reach the exit. If the demo is changed to planner control of the agent at any point, the plan runner executes the steps of the generated plan and guides the agent to the exit. This is literally AI control of the agent.

What is the difference between using planning to choose AI behavior and using planning to predict player behavior? Consider the most straightforward scenario: the player has full knowledge of the game world, understands how to win, and executes perfect play. If the planner can find an optimal solution to a scenario, then the likelihood of the planner predicting the player's behavior should depend only on how many different optimal solutions there are starting from the given initial world state. Recall that initial world state refers to the modelled world state that is the input to the planner at the beginning of a plan or a re-plan; it is not necessarily the initial state of the game world in general.

What if the player cannot or may not have perfect knowledge about the game world? Then the domain of the planner would need to operate on a restricted set of possible methods for some or all tasks. Some way of determining what a player can know or likely knows must be used to restrict which methods are matched to a given task at the time of compound task deconstruction.

In a more complex game, what if the player is not skilled at the game? Perhaps they have some other priority than winning in the most efficient way possible? These factors would also need to be reflected in the availability of methods that the planner can match to tasks. In a more complex game, Player Modelling techniques could be used to track player traits, and method availability for tasks could be determined based on those traits.

## 3 Procedurally Generated Music as a Means of Player Feedback

In order to visualize how we can use planning to predict player movements and behaviors we first need a method of feedback that will work well with our system. To keep the theming of AI, we chose to create and implement a form of procedurally generated music. By doing so, we can tweak values of how the music is generated as a means of guiding the player based on the predicted player movements and behaviors. For example, if the player chose to go in the opposing direction of the currently decided subgoal for completing a puzzle that they had been given, the music might decide to change its position/volume, how active it is (total activity in terms of playing notes) or even its nature (major/minor/key). Being able to control these parameters on the fly makes auditory feedback simple to program while still being flexible enough to use in a wide variety of situations.

How does this differ from traditional techniques though, and why go through the effort of writing an entirely new system instead of using established techniques? A very common approach to music design in the game industry is a combination of two methods of musical composition. The first being horizontal re-sequencing, a method in which pre-composed sections of music are cut and pasted in different ways in order to create dynamic looping tracks. The second method is vertical re-orchestration where selected elements/instruments in the composition are added or removed on the fly to change the total "busyness" of the track or to set a different undertone. These techniques are widely used in games like *Halo* and even *Mario Kart* to either fit a finite track into an undeterminable amount of time — common in large battle scenes across the *Halo* games — or to change the overall tension or excitement in the music to match a scenario in the game — this can be seen in *Mario Kart* during the level, character and kart selection screens; as the player gets closer to starting a race, instruments are added to increase intensity.

These techniques are great at creating music that can adapt to a given situation (hence why they are called adaptive music techniques) but do not offer the granularity required for something like instantaneous player feedback. Instead, we need to be able to alter not just the sequences/clips of music tracks but rather the notes themselves. The technique that we came up with borrows a lot of concepts from adaptive music and uses them with other PCG approaches in order to create something that would both sound reasonable and work as instantaneous player feedback.

### *3.1 Choosing an Approach*

We identified two distinct approaches to creating procedurally generated music: top-down and bottom up. Both approaches have their own pros and cons that need to be considered in order to decide which method is more applicable. Creating music entirely from scratch in a bottom-up approach allows for much more variance in the outcome. However, this type of music generation relies entirely upon "levers and knobs" in order to tweak the output into something that is not only usable but also thematically relevant — this is on top of the fact that creating something that sounds musically ideal in the first place is no easy feat. Using a bottom-up approach would require setting a significant number of constraints and tweaking many settings to get the desired results. This doesn't even consider the practical applications and how difficult it would be to use from a music pipeline point of view (I.e., musicians and composers don't want to tweak hundreds of variables in order to make music).

The other approach, top-down, is significantly more usable by musicians/composers as it tries to replicate and/or "riff" on existing music that it is given. This not only makes the technique more in line with existing adaptive music but also happens to be significantly easier to program. By choosing to replicate existing music, we can cut out the entire process of creating/managing constraints needed to reign in randomly generated music to our theme. We can also write existing music that we want the application to sound like (thematically) and directly use that in order to generate new music, something that is much more akin to adaptive music.

Given that the application needs to be able to also play music and not just generate it (after all, we are trying to achieve auditory feedback) we had to choose a method of playing music. Since we planned on playing individual notes and not clips of music, recording each individual note and how they play — aka sampling — would be extremely tedious. On top of this, the ability to do exactly that is already something that has been available for over 40 years. Naturally, the best option for audio playback in this scenario would be MIDI. Most systems already have virtual MIDI outputs that play audio directly to the system's audio service. Not only would this make playing generated music easier, but it would also allow for better scalability of the tech that we were designing. In the future, one could expand the MIDI capabilities of the generator and change out even the instruments used in order to achieve any type of audio they so desired. For this purpose, MIDI was a clear choice (we chose to use RtMidi which is a MIDI general purpose library for C++ for our demonstration)

### *3.2 Implementation*

It was briefly mentioned that our implementation would use a top-down approach. This would help create replicable examples of audio that could be used for future audio generation. To do this, our audio "scene" — for lack of a better word — would need to have a hierarchical structure to it. For our demonstration, we chose to keep things simple and used a 16-bar chord cadence as our input audio to be riffed on. At the top layer would be patterns

that represent 4 measure chord cadences — each pattern is 4 measures, hence the 16-bar chord structure. Then, for the resulting 4 measures in each pattern, each note would be generated to match the similarities in an existing input pattern.

This means that we would need 2 layers of procedural content generation: one to generate patterns, and another to generate individual notes in those patterns. Since patterns could be mostly entirely random with a couple of given constraints, we chose to go with something very simple, filtered random. Note generation on the other hand, would need to match existing note patterns in the music input. Luckily, there happens to be a nifty technique for doing something quite similar — ngram generation. By assuming that all notes in our input data are "history" and all notes in the selected pattern for future note generation are "recent history" we can come up with a data driven method of generating new notes based on examples of previous notes.

### *3.2.1 Using Filtered Random Pattern Generation for Chord Cadence Flow*

As mentioned previously we chose to use filtered random for generating new patterns. These patterns represent the order of 4 measure cadences of chords and by randomizing their order we come up with a similar idea to horizontal re-sequencing — the only difference is that instead of looping in a specified order, we make up the order as we go. Generating completely random patterns isn't necessarily a good idea though. Two ending sequences of chords back-to-back wouldn't sound very good and likewise, neither would two beginning sequences. There are also some transitions between patterns that just do not sound very good. In order to account for these problems with random pattern generation we can use filtered random.

In the context of pattern generation for music, filtered random works like so: Generate a new pattern and check the pattern against the history of previously generated patterns (see Figure 3.1). If the pattern breaks any pre-defined rules about which future patterns are possible, ignore the new pattern and generate another new pattern. Repeat this operation until a valid pattern is found. The pseudo code for this looks like:

Listing 3.1 Function to generate new filtered patterns.

```
int[] GeneratePattern(int length, int range,
                      int[] prevPattern)
{
    int[] result;

    // Generate 'length' new patterns to add to this one
    for(int i = 0; i < length; ++i)
    {
        int newPat= 0;

        // Check if prospective Pattern is valid
        int timeout = range * 2;
        do
```

```
        {
            newPat = random(0, range);
            --timeout;
        }
        while (!checkValid(prevPattern, newPat) &&
               timeout >= 0);

        // If pattern is valid, add to result
        result += newPat;
    }

    return result;
}
```

This works well for the most part, however there's a minor pitfall that will cause this generator to get stuck in an infinite loop. If our possible patterns for generation are too small and there exist too many rules, it is possible to create a situation where no pattern that is generated will be a valid pattern. To fix this problem, we can implement a timeout counter that will force a pattern to be chosen if more than x number of tries has happened. This value will have to be hand tuned as too high of a value could incur performance penalties and too low of a value may be too lenient (if there are many rules). In our demonstration we found that a value of twice as many attempts as the number of possible generated values works well — this is entirely arbitrary, however. The fix looks like this:

Listing 3.2 Fix to infinite loop in pattern validation.

```
// Check if prospective Pattern is valid
int timeout = range * 2;
do
{
    newPat = random(0, range);
    --timeout;
}
while (!checkValid(prevPattern, newPat)
       && timeout >= 0);
```

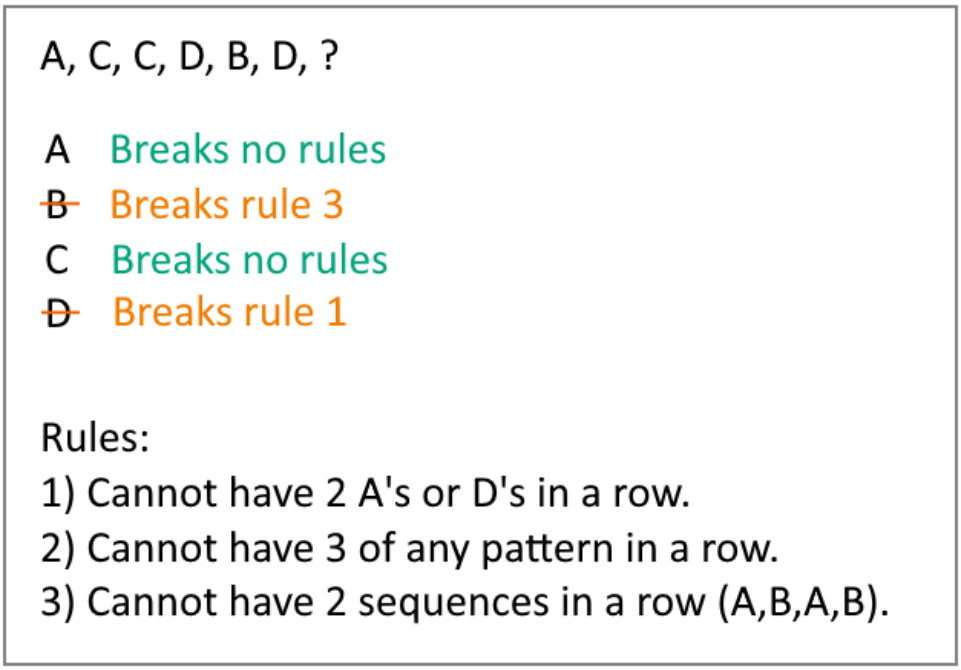


Figure 3.1 An example of a pattern sequence and valid/not valid new patterns.

*3.2.2 Creating Melodies and Chords with Ngrams*

Now that we can come up with valid patterns that will sound musically pleasing, we need to fill those patterns with notes. To do this we opted to use Ngrams, there's a couple reasons for this. Firstly, ngrams work great at detecting the occurrence of a specified pattern in a previous history of existing patterns. By assuming that the set of all possible notes from our input are valid history during a generation call, we can match our note history with said notes and come up with a reasonable method for calculating future notes. Of course, we would prefer to generate future notes that are similar to the upcoming pattern and not just any pattern (we want our music to have structure after all and not sound random). To do this we can give selected notes weights and calculate the probability of each, but more on the later.

Before we can generate new notes based on ngrams we need to be able to first specify how ngrams exist and how to load them into the program. To do this, we came up with a very simple file structure that is delimited by spaces for specifying existing note ngrams. There exists 1 file for each instrument. During file read, the program first takes in the number of styles that exist for a given instrument and then how many patterns exist for that style. Next is the number of notes/chords (chords and notes are indistinguishable, I.e., notes are just single chords) for each pattern followed by each specific note/chord. Notes/chords have 4 values, the number of notes played in the chord/notes, the duration of the chord/note, the number (MIDI pitch) of the note to be played and the velocity of the note press (similar to volume). All of this note data is loaded at runtime and used to compare against in order to generate new notes — it was written by hand from the original composition used as an example (see Figure 3.2).

```
2 // Styles
    4 // Patterns
        32 // Chords/Notes
            3 0125 051 052 // # of notes, duration, pitch, velocity
                   057 064
                   036 064
            1 0125 000 000
            1 0125 051 052
            1 0125 036 064
            2 0125 051 052
                   042 064
            1 0125 036 064
            3 0125 051 052
                   037 064
                   038 052
            1 0125 000 000
```

Figure 3.2 An example ngram file to be loaded and used to generate new notes.

Now that we've loaded the data, we need in order to generate new notes we have to come up with a method for deciding which notes to use. We want the possibility of using notes that aren't from the pattern we specify during generation —this is for a couple of reasons. Firstly, for added variance when coming up with new notes but secondly, and arguably more importantly, because we don't want to use the "best" result every single time — this would make the music predictable and stale. Instead, we can check all the possible notes and assign them with "weights". These weights will dictate how likely the note will be chosen. By assigning all notes weights and putting them into a sorted table, we can choose notes that are relevant to our pattern a large majority of the time but not always. This also allows us to use notes from other patterns when there are no notes from the pattern we want that match our note history. By changing how notes are weighted we can come up with a few parameters that change the behavior of the note generation — similar to the bottom-up approach, but much more constrained to our input music.

The key factors that determine note weight are whether the note in question is from the specified pattern being searched for and how many ngrams it matched when compared to the history provided. We can check unigrams, bigrams, trigrams, and quadgrams and for each successive match we increase the not weight further (a match for higher tier grams results in a larger weight increase). In order to select a note from the list of candidates (note weight table) we calculated the total weight of all notes in the table and selected a random number between 1 and the maximum note weight. Then we can just go down the table of notes subtracting each note's weight until our calculated weight is less than or equal to zero. At this point the randomly selected note is the note that resulted in a negative/zero calculated weight. An example of this can be seen in Figure 3.3 below.

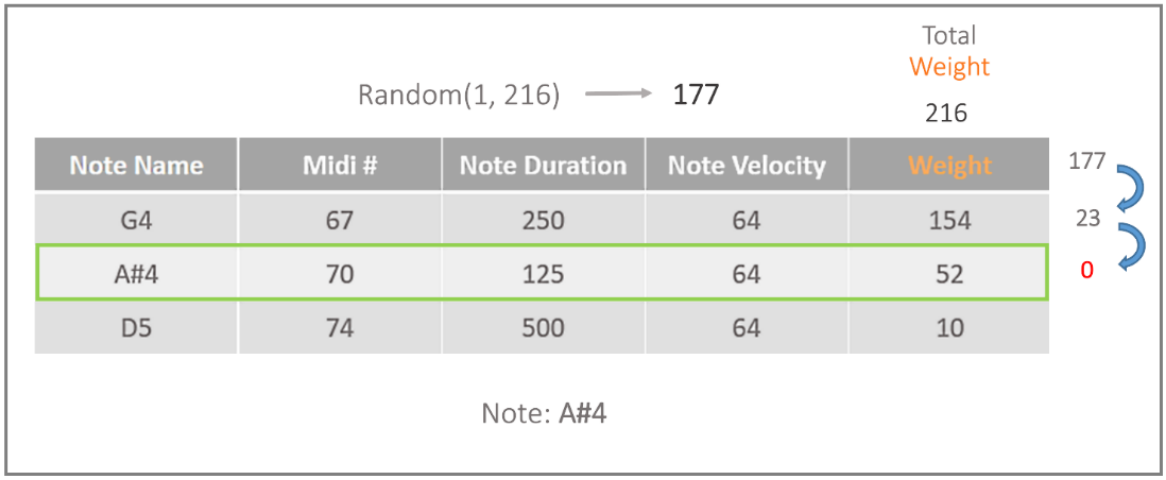


Figure 3.3 Determining a note based on its weight from a weighted table.

Listing 3.3 Code showing weighted table lookup.

```
// Calculate total weight
int totalWeight = 0;
for(Chord candidate : candidateChords)
{
    totalWeight += candidate.weight;
}

// Get a random weight index
int weightIdx = random(1, totalWeight);
int idx = 0;

// reverse sort candidate chords to get biggest in front
sortLargestFirst(candidateChords);

// Until the weight index is found subtract from it for each
// candidate chord
while(weightIdx > 0)
{
    weightIdx -= candidateChords[idx].first;
    if(weightIdx <= 0) break;
    ++idx;
}
```

Other factors that were experimented with during the creation of our demonstration included key signature validation. By assigning a key signature to a specific pattern we can check for accidentals (notes that do not follow the key signature) and try and correct/align them. Notes that did not follow the key signature were weighted less than those that did, effectively

coercing the decision to pick notes that were less likely to produce jarring sounds. Unfortunately given the small amount of variance in our input and the difficulty of choosing notes that matched the key signature while also being relevant to the melody, we opted to not use this feature in our final build of the demonstration.

### *3.3 Limitations and taking the next steps*

This approach has its limitations of course, most of which were with how it was designed — this is partially due to the nature of the demonstration not needing to be fully fleshed out. One of the biggest limitations is how we chose to use MIDI. We only used the note pitches and arbitrarily created durations of those notes in order to create music. While this did work, MIDI supports much more than we used it for. Some examples include instrument damping, usage of velocity in notes (all our notes had the same velocity and we changed instrument volumes using channels instead) and control changes for channels. These factors could even be used in ngram generation to create more varied results and more interesting note weighting.

Another big limitation of our demonstration is how we chose to store ngrams. While a text file with basic delimited structure works for what we needed, having to transcribe and modify the original music score by hand for every instrument including pitches, velocities and durations was extremely tedious, even for something as small as a 16-bar chord progression. Realistically this would be very bad for musician/composer pipeline in an actual project/game. A better way to store the ngram data would be directly as MIDI files because most musical notation software supports exporting to MIDI. This would significantly ease up on the amount of work that musicians/composers would need to do in order to implement their work into a project. This would require changing the internals of the ngram generator but would by extension support changes to the previous limitations while unlocking even more usability. This would also make using virtual instruments along with the MIDI easier too.

Further work on key signature validation (almost an entire research project by itself) would allow for better note composition on the fly. This could be combined with another technique: note interpolation. Instead of generating every single note, generate “key frame” notes and interpolate the notes that should occur between the key frames. By itself, such a technique would create odd sounding scales and out of place accidents. But when combined with an advanced form of key signature validation and some tuning values, such a method could create more melodic pieces of music (notes generally don’t “jump” around as much as our demonstration shows). If one were to take this idea a step further and add note extrapolation as well, even more melodic variance could be achieved.

One thing to note about our Procedurally Generated music is that because it was designed to work with a single 16-bar chord structure, it doesn’t perform well as “long term” music — it is very similar to minimalism however, and some people enjoy that. Listening to the generated music for more than 5-10 minutes would cause a loss of interest (although I guess that’s true for most pieces of music that are 3-5 minutes long). In order to generate variadic PCG music that changes enough to hold a user’s attention for longer, more layers of randomness in the hierarchy are needed — patterns and notes aren’t enough. Many more layers could realistically be added and there’s really no harm in adding more. Adding

another layer of randomness for “sections” (whole 16-bar sections) could add even more variance. Most musical pieces have defining and different sections (ignoring minimalism for now) that change throughout the course of the piece. Instead of using pattern generation for the next layer up, however, one could use Perlin Noise to also control the overall intensity/volume of the piece along with other tuning factors. Even just one additional layer provides a lot of extra “life” in the generated music much like emergent behaviors in AI.

Lastly, for some extra “pizzaz” and fun, implementing common sound effects such as horror shrills, triumphant fanfares, etcetera, could add to the music in an immersive way that normal sound effects cannot. By mixing in the sound effects with the procedurally generated music it gives the sound effects a sort of “blended” nature with the rest of the composition.

## 4 Applications of Procedurally Generated Music in the Context of a Planner

Now that we have procedurally generated music and a planning system that can be used to predict player movement/behaviors we can put them together and see a practical application of one amongst the other. For our demonstration we chose to create a top-down puzzle with a series of doors and pressure plates that are linked to open when walked over. Because multiple pressure plates were locked behind multiple doors, the planner could be utilized to identify subgoals required in order to reach the exit. That’s where the music comes in. When the player is in control of the agent that navigates the world, the planner can calculate the next subgoal and the procedural music system tunes variables in the music generation to change based on circumstances in the world to provide feedback to the player. When the player moves away from a subgoal, the music’s activity slows down and the volume drops — the pan position of the audio is also changed left/right in order to signify to the player where the subgoal is located.

As the player continues to reach subgoals and unlock progress to the next subgoal (and ultimately the final goal), the total activity of most of the instruments/channels being played increases — the drums for this are especially noticeable. During this, the low frequency part of the music track is constantly playing in the direction of the final goal, symbolizing the ultimate ***bass*** condition for completing the puzzle. Upon completing the final goal, the key signature of all the patterns in the music switch to playing a more major key to symbolize a triumphant victory.

This is a very rudimentary example of how one might use player prediction and PCG music together to complement existing gameplay in a fluid way. Some other ideas we tinkered with but ultimately scrapped due to limitations included changing the tonality of the music being played based on how much the players movement “agreed” with the calculated path towards their next subgoal — a sort of musical compass. Using procedurally generated music and player prediction doesn’t need to be limited to just feedback, however. One could fully implement an entire game mechanic with this idea and, while auditory feedback as a game mechanic has been done before, PCG music feedback as a game mechanic doesn’t appear to be nearly as prevalent in the gaming industry.

## 5 Conclusion

Whether it's sound effects or particle systems, feedback to the user is critical to software design — game design specifically is no exception. By providing a way to blend feedback in between two existing game system architectures, we can create a new way of visualizing — audiolizing? — events and game mechanics. Procedurally generated music and predicting player actions through planning are the tools that enable us to produce this new medium. While the demonstration we have implemented is limited, the techniques we discussed can be taken to the next — and more practical — level with relative ease. We hope that the combination of these two architectures will be used in the future to create some incredible new experiences.

## 6 References

[Humphreys 13] Humphreys, Troy. Exploring HTN Planners through Example. In Game AI Pro, edited by Steve Rabin. CRC Press, 2013, pp. 149-167.

[Nirwan 21] Nirwan, Debby. 2021. Total-order Forward Decomposition: An HTN Planner. https://towardsdatascience.com/total-order-forward-decomposition-an-htn-planner-cebae7555fff (accessed March 14, 2022).

[Orkin 06] Orkin, Jeff. 2006. 3 States and a Plan: The AI of F.E.A.R. Presented at the Game Developers Conference 2006.