Procedural Level Generation Using Occupancy-Regulated Extension

Peter Mawhorter, Michael Mateas

***Abstract*— Existing approaches to procedural level generation in 2D platformer games are, with some notable exceptions, procedures designed to do the work of a human game designer. They offer the usual benefits and disadvantages of AI applied to a cognitive task: they can work much faster than a human level designer, and are in some cases able to explore the design space automatically to find levels with desirable qualities. But they aren't able to capture the human creativity that produces the most interesting level designs, and they are usually very specific to their particular domain. This paper introduces occupancy-regulated extension (ORE), a general geometry assembly algorithm that supports human-design-based level authoring at arbitrary scales.**

I. Introduction

A good procedural level generator must satisfy complex constraints in an aesthetically pleasing manner. Existing approaches to procedural level generation systems for platformers often focus on the constraints involved, and thereby greatly limit the variety of their output. They instantiate processes which emulate a human designer in order to produce a level, using algorithms that by their nature create playable levels, with extra processes dedicated to adding interesting features that make the levels enjoyable. These processes are usually domain-specific, which allows them to generate level designs that take advantage of the interesting mechanics exclusive to their domain. The algorithm presented here, however, takes the opposite approach. It is designed to be general, in the sense that it generates geometry without knowing about the mechanics of the game (and therefore could reasonably be used with a variety of different games), and it is designed with variety rather than playability as the primary goal. To generate enjoyable levels without reasoning about mechanics, occupancy-regulated extension (ORE) relies on pre-authored chunks of levels as raw material, operating in a manner similar to case-based-reasoning. This reliance imposes some limitations on the system, but also gives it some unique advantages, including the ability to tap into the creativity of a human level designer.

Occupancy-regulated extension works by assembling a level using chunks from a library. The algorithm uses positions that the player might occupy during play to anchor each chunk, and these potential positions are also used as the extension points for a partial level. This process greatly narrows the space of levels that can be produced from the given chunk library, mostly by excluding various incoherent and unplayable levels. By regulating the incremental extension of the level using occupancy, the level is allowed to be quite complex without devolving into a chaotic jumble of unrelated components.

Although occupancy-regulated extension itself is domain-independent in that it specifies how geometry should be assembled without knowing what that geometry is, its use in a specific domain does require some adaption: a domain-specific concept of chunk compatibility must be established, and domain-specific post-processing is also usually desirable. Of course, a chunk library must also be created in order to use the algorithm. We have implemented ORE in the *Infinite Mario* engine, in order to enter the level generation contest at the 2010 Computational Intelligence in Games (CIG) conference.

II. Related Work

Games like *Infinite Mario* and *Spelunky* are examples of procedural content generation in playable systems [1], [2]. Both take a domain-specific chunk-based approach to generation: in order to generate a level, they split it into chunks, and generate each chunk using a relatively simple algorithm. For example, in *Infinite Mario*, the generator iteratively builds level sections of random length until it reaches the desired level length. Each section constructed is built using one of 5 template/algorithms: 'straight', 'hill', 'tubes', 'jump', and 'cannons'. These algorithms are pretty simple, building a specific type of terrain with a bit of random variation thrown in. In a similar fashion, levels of *Spelunky* are split into sixteen 10x8 rooms arranged in a four by four grid. Each room is randomly selected from a list of pre-authored segments, but the pre-authored room segments each contain a mixture of static terrain and hooks for random generation of sub-components. Finally, a separate algorithm interfaces with the generation of room details and adds enemies and treasure.

Both of these algorithms at their core generate "good" geometry, which they then decorate and modify, within strict limits. *Spelunky* is a bit more permissive in this respect, but also includes game mechanics for destroying terrain, which make the playability constraint easier to satisfy. The focus on playability limits how creative these algorithms can be, and in the case of *Infinite Mario*, it greatly limits the variety that is generated.

In both games, of course, new bits of content (either template/algorithms in *Infinite Mario* or room templates and sub-components in *Spelunky*) could be authored to increase the variety present in generated levels, but neither game supports combining geometry to create variety. In *Infinite Mario*, each template/algorithm works in its own space, so emergent effects are limited to the borders between sections, and are minimal. In *Spelunky*, components are combined hierarchically, which results in limited emergent geometry, but there is no interaction between geometric components that could produce truly novel architecture. On the other hand, *Spelunky*'s separate object generation does react with the geometry to produce emergent variation, which gives the game good replay value. It's also worth noting that by using specific room variants, *Spelunky* can incorporate large-scale features like shops and bosses that would be difficult to incorporate in a fine-grained generator.

Beyond games like *Infinite Mario* and *Spelunky*, there have been several research projects aimed at applying procedural generation methods to platformer levels. For example, Compton and Mateas have proposed a framework for rhythm-based procedural generation of platformer levels [3] which was extended by Smith et al. [4], [5], and an alternative challenge-based approach has recently been proposed by Sorenson and Pasquier [6]. Other projects have used procedural generation as a means of studying the effects of level design on player behavior [7], or as the basis of a dynamic difficulty adjustment algorithm [8]. For the most part, the constraints that these efforts have adopted in order to facilitate reaching their goals have drastically limited the design space within which their algorithms work. For example, out of the citations above, only Compton and Mateas' initial framework allows for having two separate routes through the level stacked on top of one another. This is an infrequent occurrence in actual Super Mario levels, but it can be used to great effect by a human designer. Of course, limiting the generative space in these ways has allowed these projects to achieve other goals, like rhythmic pacing, and has enabled them to meet strong playability constraints. In contrast, because it uses human-designed chunks and arbitrary extension, occupancy-regulated extension is able to reproduce the full range of levels that a human might design, but does not guarantee playability.

An example of a less-limited system is Laskov's work on reinforcement-learning-based level generation [9] which treats level generation as a sequence of choices undertaken by a level-building agent and uses reinforcement learning to create a level-building policy that is executed to create a level. Laskov's system uses 3x6-tile chunks of level components as its actions, and places them from left to right at different heights within the level, creating a system of explicit branching paths. It uses a simple algorithm for tracing reachability during this process, which allows it to guarantee playability. Because Laskov's generation domain only includes one obstacle type, one enemy type, and one treasure type, it's difficult to assess the generality of the learning-based algorithm (or whether it can generate geometry that makes clever use of game mechanics), but the ability to use custom action types is promising.

Besides work in procedural level generation, ORE is inspired by (but only loosely based on) the reasoning strategy known as case-based reasoning [10]. Although the 'cases' used are level geometry rather than situations, and the full case-based reasoning cycle of recall and adaption is not used, the principle is the same: iteratively make a decision about what to do based on past actions. Accordingly, questions of how to generate and maintain a case library are relevant.

III. Occupancy-Regulated Extension

The core concept of occupancy-regulated extension is occupancy, which is expressed as potential positions ('anchors'). Anchors are locations within the level which the player might occupy during gameplay. Accordingly, the raw material of ORE consists of a partial level and a set of chunks of level geometry, both annotated with potential positions. The algorithm then works by adding chunks to the partial level in a manner that preserves anchors, iteratively expanding the existing content. To generate a full level, the initial content should just be a starting area with an anchor for the player's starting location, and a library of chunks (the size and nature of this library are discussed in section IV-C; our current implementation uses a library of 42 chunks, the largest of which is 10x10 tiles).

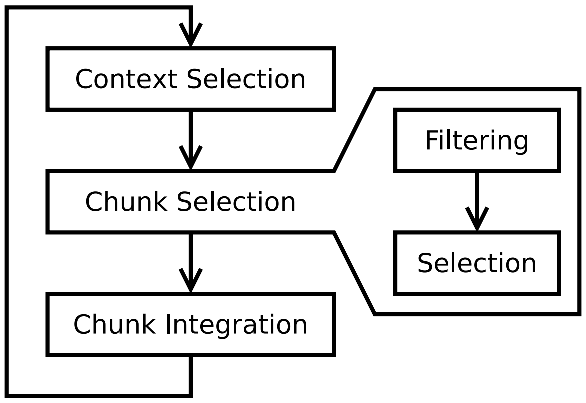

Fig. 1. The steps of the occupancy-regulated extension algorithm.

The ORE algorithm iterates using three steps (see figure 1). First, a context is selected, which will be used to expand the current content. Second, a chunk is selected from the library that matches that context. Finally, the chunk is integrated with the existing content.

Context selection consists of deciding which part of the existing geometry to expand. ORE itself is agnostic to the context selection method, but it must select a single anchor at which to expand the level. The simplest possible context selection procedure just picks an anchor at random from the existing anchors within the generated content. This often picks positions at which no further expansion is possible given the chunk selection algorithm and chunk library, however, so a slightly smarter routine is used: select anchors from existing content in random order, but don't re-use any until all of them have been used. Much more intelligent context selection criteria are of course possible, and this is one area that warrants future study.

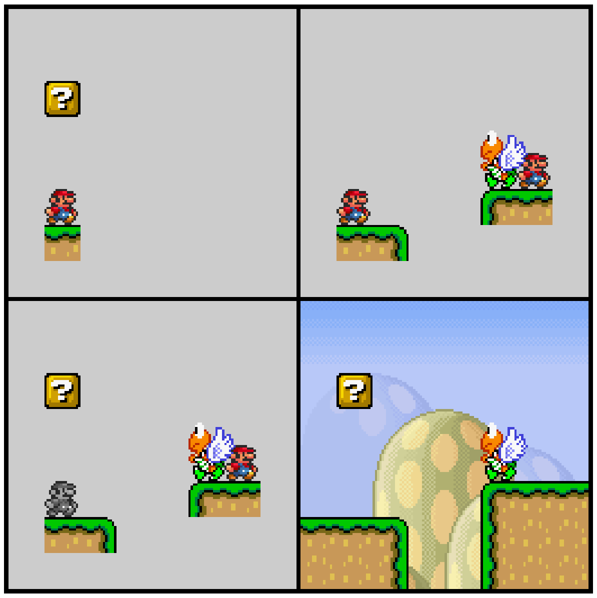

Fig. 2. A dramatization of chunk selection. In the upper left, we see existing content, including an anchor. The upper right shows a matching chunk (based on geometric filtering). The lower left shows the resulting geometry (the original anchor is now marked as used), and the lower right shows the level segment after post-processing.

Following context selection, chunk selection consists of a filtering step followed by a selection step (see figure 2). First, the chunk library must be filtered to select a list of compatible candidates, and then a single chunk must be chosen from this list for integration. Once again, ORE is agnostic to the actual filtering and selection methods used, except that they should produce a single chunk with which to extend the level. It is important that the selection process produce a compatible chunk, however; otherwise ORE would happily construct a completely incoherent level. This notion of compatibility, of course, depends on the domain in which generation occurs, and on what exactly a chunk contains.

For *Infinite Mario*, we use the various sprites (such as normal ground, a pipe, or a Koopa) as components, along with some extra information (like which direction(s) a platform should extend in). Chunks are sets of these components placed in relation to each other, with anchors mixed in as just another component type. The chunk library encodes these component sets in a library file, annotated with chunk properties as in this example chunk, which consists of a pipe on flat ground that contains a piranha flower (the `@` characters represent anchors; the frequency tag here means that this chunk will have a weight of 0.6 during random selection rather than the default of 1):

```
frequency: -0.4
   @  \
  [[  \
  [[  \  f
@ [[ @\
++++++\
```

The filtering algorithm in our system iterates over all of the chunks in the chunk library and checks each for compatibility with the chosen context at each of the anchors within the chunk. The filtering algorithm uses the following steps to produce a matching chunk given a test chunk from the chunk library (along with a particular anchor within it) and the chosen context (an anchor within the 'query chunk', which is all of the components within the level so far):

1. Align the test chunk with the query chunk using their anchors.
2. For each component in the test chunk:
   a. If it duplicates an existing component, eliminate it and continue to the next component.
   b. If it is otherwise on top of an existing component, reject this test chunk.
   c. If it is not a platform, for each of its up to 8 adjacent neighbors in the query chunk:
      i. If the neighbor is duplicated by a component in the test chunk, continue.
      ii. If the neighbor matches the test component, eliminate the test component and continue testing the rest of the chunk.
      iii. Otherwise, reject the test chunk.
   d. If it is a platform, for each of the up to 25 neighbors in the query chunk within 2 units of the component:
      i. If the neighbor overlaps a matching component in the test chunk, continue.
      ii. If the neighbor isn't a platform and directly blocks the test component from extending as specified, reject the test chunk.
      iii. If the neighbor is a platform, reject the test chunk unless it is 'compatible'[1].
   e. Eliminate any remaining components in the test chunk that extend beyond the edges of the level.
   f. If the test chunk was not rejected, return all of the components not eliminated during testing. These form the matching chunk.

Once a set of matching chunks has been found, the selection algorithm is invoked. This selection algorithm can be used to control the generator output, and tuning it is important for producing fun levels. The currently implemented selection process involves computing a weight for each chunk and picking one at random with probabilities proportional to their weights. Weights are computed using a combination of a per-chunk weight specified in the library, the number of times that chunk has been used before, and whether or not the chunk is "precise" as specified in the library. Of course this particular selection process isn't part of ORE; the core ORE algorithm only requires that the chunk selected be compatible with the chosen context (as ensured during the chunk matching step).

Given a context and a compatible chunk, the final step is

[1]Depending on their relative positions, neighboring platform components might be compatible if they extend towards each other, if they form an L shape (i.e. both extend towards or away from a common point in orthogonal directions), or if they extend away from each other with a gap in between.

simply to integrate the new chunk with the existing geometry. This is achieved by pasting in the matched chunk at its specified anchor. During this step, the potential position that is used to anchor the new chunk is marked as used to inform the next context selection step, and any other potential positions in the incoming chunk are added to the existing content just like the other pieces of the chunk. Because of this, new chunks will grow out of the chunk that was just placed, eventually forming a complete level.

A. Edge Cases

The three steps of ORE can be iterated to build an entire level, but the algorithm can run into trouble in two ways. First, it's possible that by using chunks that include only a single anchor, the total number of available anchors in the content being generated will reach zero. Second, it's possible that when attempting to find a compatible chunk for a given context, no chunk will match. The first problem is solved by simply resetting the list of used anchors, so that each of the original anchors throughout the existing content may be checked again for chunk placement. But of course it could be the case that no chunks can fit at any of these anchor points. The solution to the second problem fixes this issue as well, though: when no matching chunks are found for a context query, the algorithm extrapolates a new anchor several units to the right of, and possibly a few units above, the current anchor, as long as this new anchor wouldn't overlap existing geometry. Presumably, this new position will permit further generation, and in the worst case, iterating the addition of these free anchors should eventually lead to a suitable place for further generation.

B. Post-Processing

Even after enough components have been placed to form a complete level various extra mechanisms are necessary. For example, many components indicating extension in a particular direction will not necessarily have neighbors immediately adjacent to them. To fix this, a post-processing algorithm goes through the level and expands all components that are marked for extension, forming solid blocks of ground from sparse platforms (this is illustrated in the final panel of figure 2).

A second post-processing step takes raw platforms and gives them specific sprites. For example, each of the edges of a platform must be set to the correct edge sprite depending on its facing (figure 2 uses these edge sprites to indicate extension constraints on the components involved, but the correct sprites actually aren't known until after the expansion step). This cleanup step is relatively uninteresting, except when an invalid sprite is placed (in *Infinite Mario*, this can happen when ground is placed that is only one tile thick: there is no sprite for ground that has edges on two sides). In these cases, the post-processing step does one of two things: it either replaces the offending sprite with a similar sprite that fits (in the case of thin ground, it uses solid blue blocks), or it removes the offending sprite entirely, effectively modifying the level. Given the fact that invalid sprites are mostly produced by quirks in the generation algorithm, this domain-specific post-processing can correct for some edge cases.

The final post-processing step applies some simple global constraints to the level. This addresses the inability of the simple iterative algorithm to respect global constraints. For example, traditionally in Mario levels, the distribution of powerups is regular: they are distributed along the x-axis such that they occur infrequently, often exactly twice in a level. Because this constraint is tied to both the x-position of the powerup box and the existence of other powerup boxes, it would be difficult to implement as part of the chunk selection process. Instead, the global post-processing step iterates over the level from left to right, using simple probabilities to replace most powerup boxes with coin boxes. The algorithm uses a threshold that increases with each iteration, and leaves a powerup in place if a random number is below the threshold, resetting the threshold when it does so. By tuning the initial threshold and the increment, an even but random distribution of powerup boxes can be achieved, assuming the chunk placement algorithm places powerup boxes regularly. This exact same step is also performed separately for enemies, winged enemies, and enemy spawn points, although each of these features use a much faster increment than powerups do. For these other features, if the probability check fails, the feature is simply removed. These post-processing steps allow more exact control over the output than chunk selection weights would alone, especially with regards to difficulty.

The use of domain-specific post-processing techniques highlights some of the weaknesses of the occupancy-regulated extension algorithm on its own. Although ORE is agnostic to chunk content, it relies on smart chunk selection and post-processing algorithms to create interesting levels. To use ORE in a new domain, it would be necessary to define a new compatibility algorithm and post-processing procedure.

IV. Discussion

A. Characterization

Without the opportunity to perform experimental playtesting on the output of our system, we rely instead on critical analysis of the generator output to judge the quality of our system as a procedural content generator. Eventually, a more formal analysis including metrics for desired output properties should be performed, as has been done with other systems [11].

B. System Output

The occupancy-regulated extension algorithm generates content within a large space of possible variations. Even given a small fixed chunk library (there are 42 chunks in the library used to generate the examples shown here), because the space of possible levels includes all tilings of those chunks in the plane (constrained by potential positions and compatibility), the generative space is quite large. A small sampling of this space is enough to demonstrate that it is

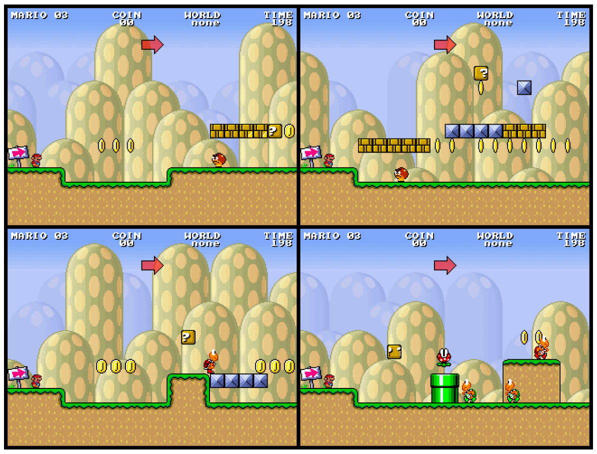
Fig. 3. The beginnings of four different levels. The arrow behind Mario and the initial downward curb are part of the fixed starting platform that forms the basis for generation.

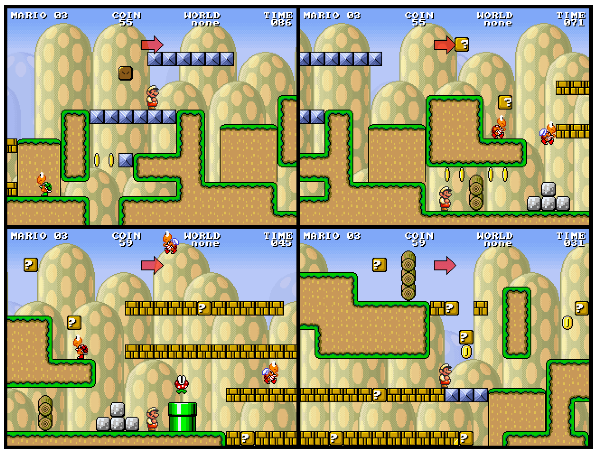
Fig. 4. Four screens of progress through a complex generated area. Not all levels contain structures this complex, but neither are they extremely rare.

also an interesting one: figure 3 shows just the first screen of four different generated levels (each panel contains between about four and ten chunks). These four levels were generated one after the other, and used identical generation parameters and the same chunk library. From examining these examples, some of the output patterns become clear.

Given the starting components and the chunk library, levels where the player starts near the top of the screen will be rare. It's also the case that much height variation within the first bit of the level is uncommon. However, unlike levels of the original *Infinite Mario*, the placement of enemies, coins, and blocks is quite unpredictable. At the same time, we can already see that because of the chunk library, the generator does have some idioms that it uses: the walking Goomba below a line of blocks appears in two of these levels, as does the line of three coins above open ground (both of which are chunks in the library). Because these idioms combine with surrounding geometry, they don't become boring: the line of coins is more interesting with blocks directly above it, or when it gets placed at the edge of a platform. Beyond idioms, there's also a basic logic to the way that components are placed: there is lots of variation, but it isn't just chaos. By maintaining the fundamental continuity of potential positions, ORE achieves high variability while remaining coherent. In some cases, it does produce odd architecture, like the open space under the line of blocks in the bottom left example, but this odd architecture does not dominate the levels, and it sometimes crosses the line from odd into interesting.

Besides variation, ORE is also capable of producing complexity. This is something that the original *Infinite Mario* lacked, and something that is easy to neglect when a generator has other specific goals. Figure 4 illustrates a complex area. Although this is a hand-picked example, such complex areas aren't rare, and portions of levels that involve multi-path structures are quite common. Again, this complexity is not just chaos: the constraints of potential positions and compatibility give rise to recognizable paths through these areas: they are complex in an enjoyable way. It's also worth pointing out that the largest chunks in the chunk library for these examples are no larger than 10x10 tiles (in the lower-right panel of figure 3, the ledge at the right of the screen, along with the two coins above it, the ground below it, and the three enemies near it, is a single chunk), and the smallest chunks are around 2x3 tiles (the lines of brick blocks in figure 4 are generated by a series of overlapping chunks each of which contains a pair of bricks). Although complex areas could be added via large hand-authored chunks, they also arise algorithmically from the interaction of many small chunks.

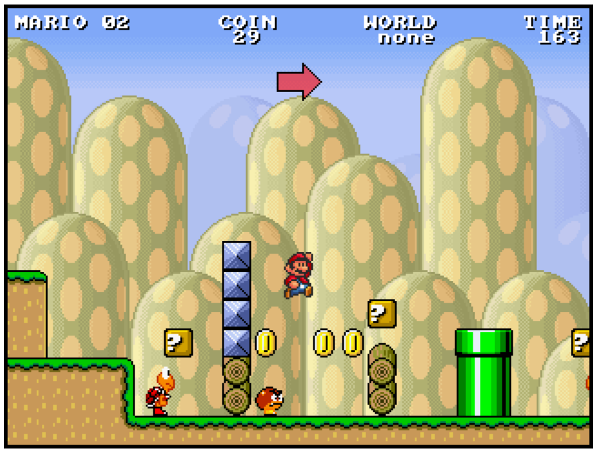

Fig. 5. An emergent bit of level design: the block above the logs contains a powerup, but it can only be reached if the player doesn't already have a powerup (or is very skilled).

Variation and complexity are certainly desirable properties, but they might just result directly from hand-authored chunks. If this were the case, then ORE would be uninteresting as a creative system, even if it were an excellent tool for producing levels. But examination of generated levels shows that this is not the case: the random combination of compatible chunks sometimes results in interesting use of game mechanics that was not present in the chunk library. Figure 5 shows an example of this: two chunks, one involving a stack of logs and the other involving a powerup block, have been superimposed. The resultant geometry makes it difficult for the player to obtain the powerup unless they don't currently have one (getting a powerup increases your size, making it difficult to fit below the powerup block). These sorts of emergent structures are fairly rare (this one occurs once in perhaps several dozen levels) but they show that the system does have some creativity to it. Because clever combinations of elements are something that players look forward to in human-designed levels, their presence in a machine-generated level is encouraging.

C. The Chunk Library

None of this analysis of variance and complexity has addressed how the chunk library influences generation. The existing chunk library contains 42 chunks split into two groups: an initial library of 22 chunks, and an extended library of 20 additional chunks. The initial library is more general, while the extended library contains more varied components.

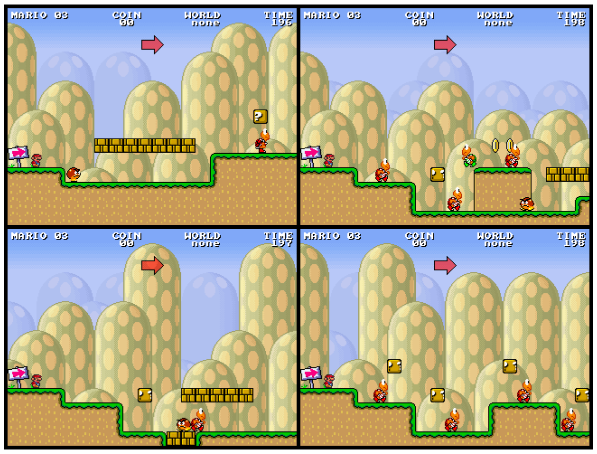

Fig. 6. The initial screens of four levels constructed using only the initial chunk library. Note that at least one chunk (a red Koopa under a question block) appears in all four of these, and is used several times in panel 4.

To get an idea of the effects of chunk library variation, figures 6 and 7 each show the first screen of four levels (like the earlier example, these levels were generated one after the other). The levels in figure 6 were generated using only the initial chunk library, while the levels in figure 7 were generated using only the extended chunk library. Immediately, the patterns in each group can be seen, and properties like the availability of chunks that represent changes in ground height are clear. These two example sets have noticeably different styles which are consistent within each set. The first set uses lots of Koopas on varying-height ledges, along with frequent blocks. The second set uses lots of vertical objects, like pipes and logs, along with special enemies like bullet towers, and it also has some blocks scattered in midair. While the ability to generate stylized levels using a custom chunk library is useful, over the course of an entire level the components used in the second batch might become boring due to uniform height. Because of this, that chunk library will also be unlikely to generate complex structures. The first set of levels holds more promise, since there are ledges and blocks that could contribute to complex structures, but it the uniform enemies and frequent blocks might become boring after a while.

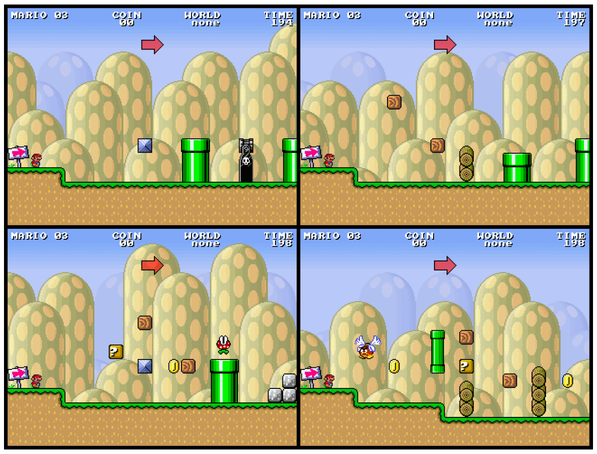

Fig. 7. Four more levels, this time constructed using only the extended chunk library. The small library is evident here as well: a chunk consisting of two wooden blocks appears in three of these four starting screens.

By looking at these smaller chunk libraries individually, it is evident that alone, they do not quite produce interesting variance. But using both libraries together can produce the levels shown in figures 3 and 4. This shows that a relatively small selection of chunks can create interesting levels, and by manipulating the chunks present in the library, stylistic effects can be achieved. It's also apparent that chunk libraries require certain kinds of chunks in order to generate complex and interesting geometry configurations (chunks that incorporate height variance are specifically useful for this). Unfortunately, the effects of larger chunk libraries and non-human-authored libraries have not been studied.

D. Strengths

One great benefit of ORE is that it incorporates human-designed elements into the generated levels. Its ability to use chunks of any scale allows for complex human-designed components to be used, and the iterative nature of the algorithm even permits joint design with a human, since ORE doesn't care how much of a level has been authored before it starts generating. It would be simple to design a mixed-initiative system where a designer built part of a level, asked ORE to generate some extra pieces, and then built more based on the results. The desirability of such systems has been established by Smith et al. in their continuing work on mixed-initiative rhythm-based generation [5].

By using human-designed elements as a base, ORE also achieves some measure of domain-independence. Although chunk selection and post-processing depend on the specifics of the domain, these don't require a complex player model or deep knowledge of game mechanics. Things like how far the player can jump, what geometry is suitable for which enemies, and the effects of powerups are all encoded in the chunk library, and aren't represented explicitly in the chunk selection or post-processing algorithms (except for the code that generates new potential positions when the algorithm gets stuck, which knows about jump distance). Using ORE with another 2-dimensional platformer would not be terribly complicated, and in fact, with more involved post-processing, ORE could in theory be applied to a 3-dimensional space. The chunk-based nature of the algorithm also means that level styles can be changed or mixed together simply by manipulating the chunk library.

The chunk-selection process is a natural place to tweak the generator output. For example, the current implementation adjusts difficulty in part by marking chunks with tags such as "precise" (meaning that traversing the chunk requires precise jumping). By using this information during chunk selection to compute weights, the difficulty of the level can be altered in very specific ways. Because the algorithm is relatively quick with a small chunk library (it takes around 15 seconds to generate a complete level using a library of 42 chunks), it might even be possible to perform dynamic difficulty adjustment using ORE.

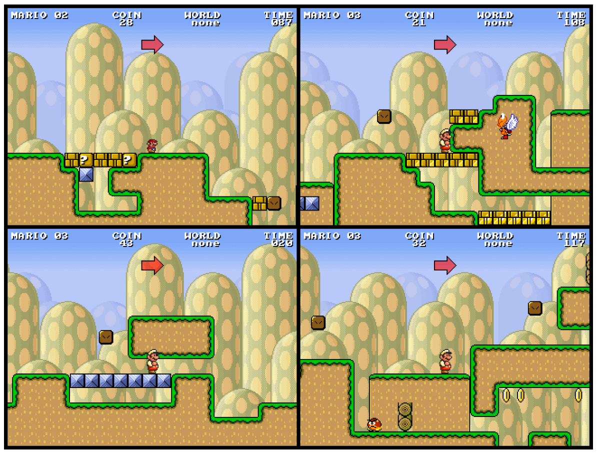

Fig. 8. Four examples of generation mistakes: an unreachable area, a trapped enemy, a location where the player can get stuck, and a graphical artifact (the stack of logs is cut off at the top).

E. Weaknesses

Naturally, many of the decisions made in the design of ORE involved trade-offs. In particular, using a chunk library for generation necessitates the generation and maintenance of said library. Also, there are aspects of the algorithm that depend on the nature of the chunks in the chunk library (as discussed briefly above in the context of post-processing). As a generative algorithm, ORE is uninteresting to the degree that it depends on a very specific type of chunk library. The examples above that show generator output under variations in the chunk library are somewhat reassuring, but further experimentation is necessary to validate this. The fact that ORE produces interesting output at very small chunk library sizes is encouraging, however, because even if it requires a hand-tuned chunk library, it could still be useful in the context of a mixed-initiative generator that specifically incorporated chunk generation as a task.

Another weakness of ORE is that by prioritizing variety, it fails to guarantee playability. Given only the constraints that arise from the anchor points, it's possible to build an unplayable level, because interactions between chunks placed can violate global playability. For example, after placing several chunks that create new anchor points successively farther to the right, the algorithm might place a chunk that creates an anchor point above an existing one. This could be followed by placing chunks back towards the left, which might almost (but not quite) overlap with the existing chunks that were placed going to the right, forming an insurmountable cliff. Another possibility is that if there are "decorative" chunks in the library that contain powerups or enemies but no geometry, these might be placed end-to-end to form an unsurmountable gap. The current implementation does not address this weakness, and it does sometimes create unbeatable levels. This is currently a rare phenomenon (in part because of the nature of the chunk library), but steps to address it are important directions for future work.

Finally, ORE does require a robust compatibility algorithm. Without care, interactions between the chunk selection and later post-processing steps can produce undesirable situations, as seen in figure 8. In most cases, these don't have a major impact on gameplay (for example, to get the avatar stuck in that third panel, a very specific action sequence is necessary), but ideally they could be avoided by using some more advanced compatibility algorithm. Future work on ORE will address issues like these.

V. Future Work

A. Mixed-Initiative Design

One of the most promising directions for future work would be to incorporate occupancy-regulated extension into a mixed-initiative design tool. Smith et al. [5] have motivated the creation of such tools, and their existing work shows promise. ORE is already well-suited to the task because it works iteratively: it doesn't care whether existing geometry was specified by hand or by previous iterations of the algorithm. Furthermore, in a mixed-initiative tool, the use of chunks would provide expressive power beyond just the ability to hand-author segments of the level. By labeling chunks in the library with various properties, a designer could specify regions in which certain properties were preferred. This preference could then be realized by manipulation of weights during the chunk selection process. The post-processing steps could also be manipulated by the designer (and of course, hand-designed content could be excluded from them), allowing for statements like "this area should have a lot of enemies, but after it there should be some tricky platforms with few enemies," to be encoded as a combination of property specifications and post-processing customizations.

B. Library Improvements

The exact performance of ORE under variations in the chunk library is not known. Experimenting with different chunk libraries would better characterize the algorithm, and using ORE with machine-generated chunk libraries would be an interesting goal. Such a chunk library might be automatically extracted from a level by following a play trace through the level and taking periodic snapshots. The level components near the player in each pair of snapshots would become a case, and the player positions from each snapshot would be its anchors. The cases could even be automatically annotated with properties according to in-game events such as the player dying. Ideally, this would allow the use of ORE to reconstruct variations in a particular style, simply by extracting chunks from a level designed in that style and using those for generation. Of course, the chunk extraction algorithm might have to be quite elaborate if ORE requires a very specific mix of chunks to work well, or a mixed strategy using a small, bland base chunk library along with extracted chunks might be successful.

C. On-Line Extension

In theory, ORE could be modified to run during play, and the chunk selection algorithm could be influenced by player behavior and in-game events to achieve some measure of dynamic difficulty adjustment. Of course, the various post-processing algorithms would have to be adapted run on-line as well, so this would not be a trivial task. Also, the generation algorithm would have to be constrained to a particular area within the level. On the other hand, this alteration is likely to provide at least a modest speed increase, which should be enough to run the algorithm without noticeably slowing down the game (at least with a chunk library about as big as the present one).

D. Playability

Levels that aren't playable can be incredibly frustrating, even if they are relatively rare. Thus, adding a playability guarantee to ORE is an important research goal. To do this, a playability checking step could be added to the algorithm, after chunk selection. When the check failed, the algorithm could either select a new chunk, or it could try to add another chunk to make the result playable again. A detailed player model would be the basis for playability checking.

E. Generalizability

Besides work to improve ORE itself, the application of ORE to new domains offers interesting possibilities. It would also be interesting to investigate alternate compatibility constraints for new domains, and to characterize the authoring burden associated with developing a new chunk library. Although ORE seems relatively domain-independent, a project that uses it in some other domain would be able to verify or disparage this claim.

VI. Conclusion

Games like *Infinite Mario* and *Spelunky* use algorithms that are based on piecing together chunks to form a level, but only do so within strict constraints. Less-constrained iterative techniques like case-based reasoning are well known, but haven't been directly applied to level generation. occupancy-regulated extension is inspired by both of these approaches: it uses level chunks as its core unit of operation, but has the same context-based approach as case-based reasoning. The key insight that allows it to be an effective algorithm is the use of potential positions as anchor points. By focusing on occupancy, diverse level chunks can be combined into complex levels with emergent details while avoiding degenerate chaos. The focus on occupancy creates local constraints that force chunks to fit together in meaningful ways, and, combined with compatibility checking, prevents chunks from interfering with each other too much. Once this balance is achieved, the many desirable properties of the chunk-based approach can be realized: the system is amenable to use as a mixed-initiative tool, and supports customization of the basic algorithm using chunk weights, alternate chunk libraries, and component-aware post-processing. Ultimately, the quality of the algorithm will be tested in the CIG 2010 level generation competition, but the initial prospects for occupancy-regulated extension as a procedural content generation algorithm are good.

References

[1] M. Persson, "Infinite Mario Bros," (Game) http://www.mojang.com/notch/mario/, last accessed: June 10, 2010.

[2] D. Yu, "Spelunky," 2009, (Game) http://www.spelunkyworld.com/, last accessed: June 10, 2010.

[3] K. Compton and M. Mateas, "Procedural level design for platform games," in *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*, 2006.

[4] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2d platformers," in *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*. New York, NY, USA: ACM, 2009, pp. 175–182.

[5] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proceedings of the 2010 International Conference on the Foundations of Digital Games*, 2010.

[6] N. Sorenson and P. Pasquier, "The evolution of fun: Automatic level design through challenge modeling," in *Proceedings of the First International Conference on Computational Creativity (ICCCX)*. ACM, 2010.

[7] C. Pedersen, J. Togelius, and G. Yannakakis, "Optimization of platform game levels for player experience," in *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2009.

[8] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: Dynamic difficulty adjustment through level generation," in *Proceedings of the 2010 International Conference on the Foundations of Digital Games*, 2010.

[9] A. Laskov, "Level generation system for platform games based on a reinforcement learning approach," University of Edinburgh, Tech. Rep. EDI-INF-IM090699, 2009.

[10] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[11] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 International Conference on the Foundations of Digital Games*, 2010.