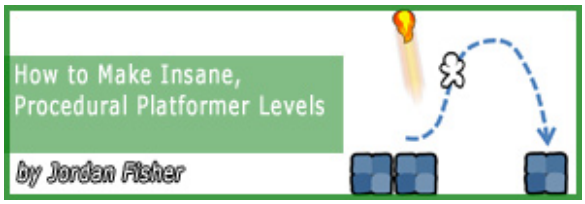
GAMASUTRA
The Art & Business of Making Games

PRINT

How to Make Insane, Procedural Platformer Levels

By Jordan Fisher



[*In this article, Cloudberry Kingdom developer Jordan Fisher explains precisely how he created the algorithmic level design system for the procedurally generated platformer -- and how you can design your own AI. You can back Cloudberry Kingdom on Kickstarter right now.*]

So you want to make a procedural platformer. You want it to spit out levels on demand, and you want the levels to be awesome, challenging, and fun. You want the algorithm to be flexible, so that you can design a new obstacle or change the game physics and instantly have new levels created with your new content. You want it to spit out easy levels for new players, hard levels for core players, and brain melting insanity for leet *StarCraft* gods with APMs over 9000.

Oh, and all this insanity had better be possible to actually beat, or the players will organize a coup and destroy your reputation via Reddit, 4chan, and probably their personal blogs that they started just to pummel you.

Basically, you want a silicon imprint of Miyamoto and team that you can capture in a box of software to distribute to the masses.

That, or a well-trained level design AI. As lead programmer at Pwnee Studios, my job has been developing such an AI for our first platformer title.

Procedural content has done wonders in genres other than platformers. The expansive plains and dungeons in *Diablo*, the beautiful landscapes in *Minecraft*, the creature animations in *Spore*. Dungeon crawlers and sandboxes in particular have a long history of awesome procedural generation. There are side-scrollers with random levels too, such as the amazing *Spelunky* and *Terraria*.

In this piece, I talk more specifically about random levels for a faster paced style platformer (*Mario*, *Sonic*, *Super Meat Boy*). This is a relatively unexplored area for procedural algorithms, and is in many ways much more challenging. We want pixel-perfect jumps, death-defying brushes with lasers, and tunable difficulty for any skill level, all while guaranteeing the levels generated have solutions.

There are three things a good procedural algorithm needs to nail:

1. **Feasibility**. Can you beat it?

2. **Interesting Design**. Do you *want* to beat it?

3. **Appropriate Skill Level**. Is it a good challenge?

Satisfying any one of these is actually pretty easy. Satisfying them all simultaneously forms a very tight constraint problem. Feasibility, in particular, is a constraint that has greatly hampered efforts to make good procedural platformers; however, the other two requirements are just as difficult to perfect.

The first constraint, feasibility, is the most brittle requirement, so we will start there.

Design Requirement #1: It Must Be Feasible

There are simple techniques to guarantee a dungeon in *Diablo* has a path through it, analogous to how one assures that a generated maze has a solution. In a maze, the player has absolute control over their position, not considering the constraints imposed by walls. In a platformer, a player has a much looser control of her position and must factor in the game's physics: momentum, gravity, friction, and so on. This greatly exacerbates the difficulty of the problem.

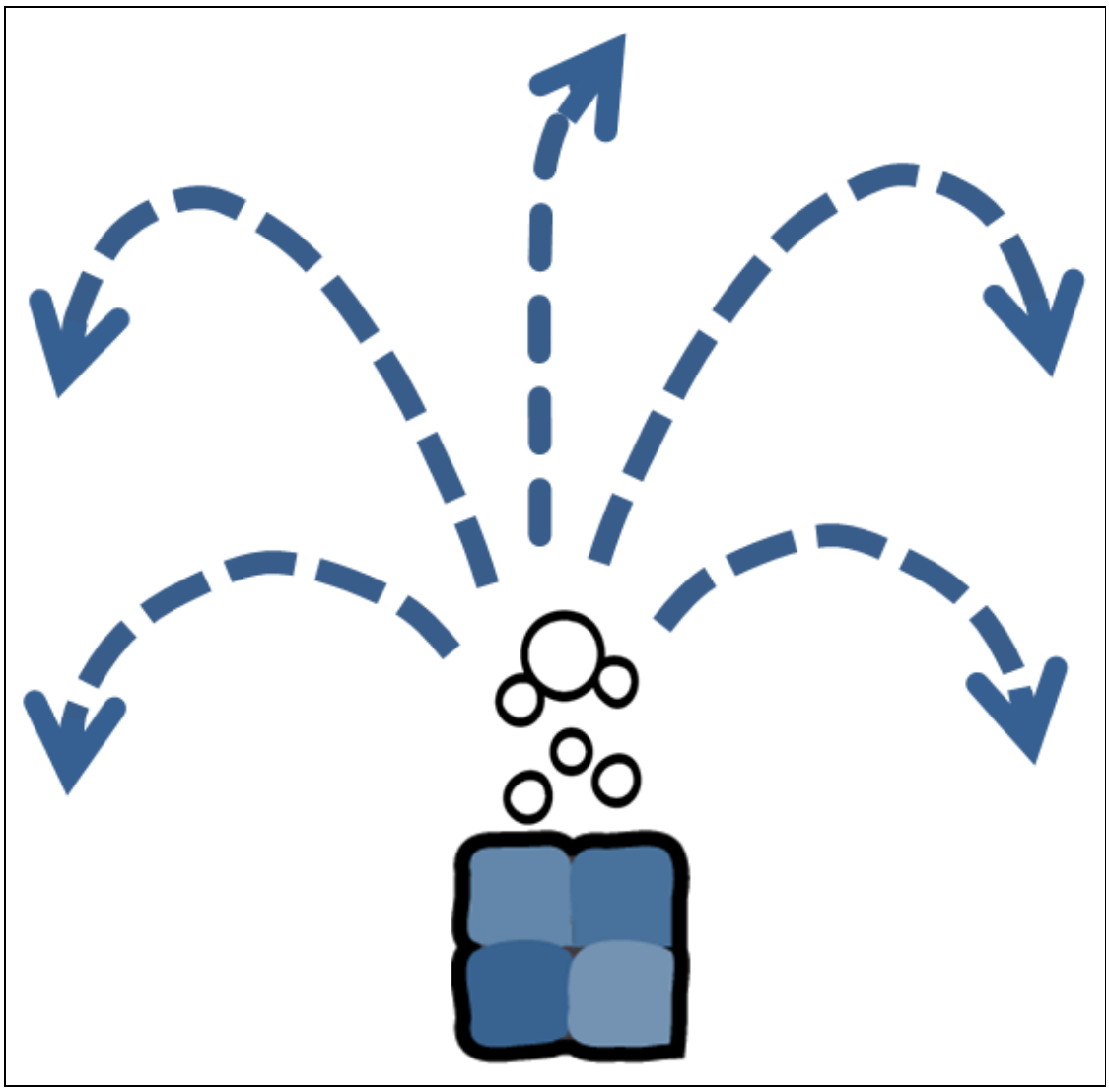
If you generate a maze with no solution, then you can knock down a few walls until a solution appears. If you generate a platformer level with no solution, it's not at all clear how to fix things.

We need to make sure our levels are possible to beat. To satisfy this need for provable feasibility we rely on a very good computer player that we can hand off levels to. The player AI directly proves the levels are possible by beating them. This is easier to say than implement, but luckily good platformer player AI is a well-researched topic, with some notable implementations, such as this one. Implementing a good AI is non-trivial, but fairly straightforward.

Now that we have our awesome ninja AI, we can test our levels before throwing them at our players. Even better, if a player gets stuck on a level, we can let them watch the AI. The player can learn and improve, or at least suspend their incredulity.

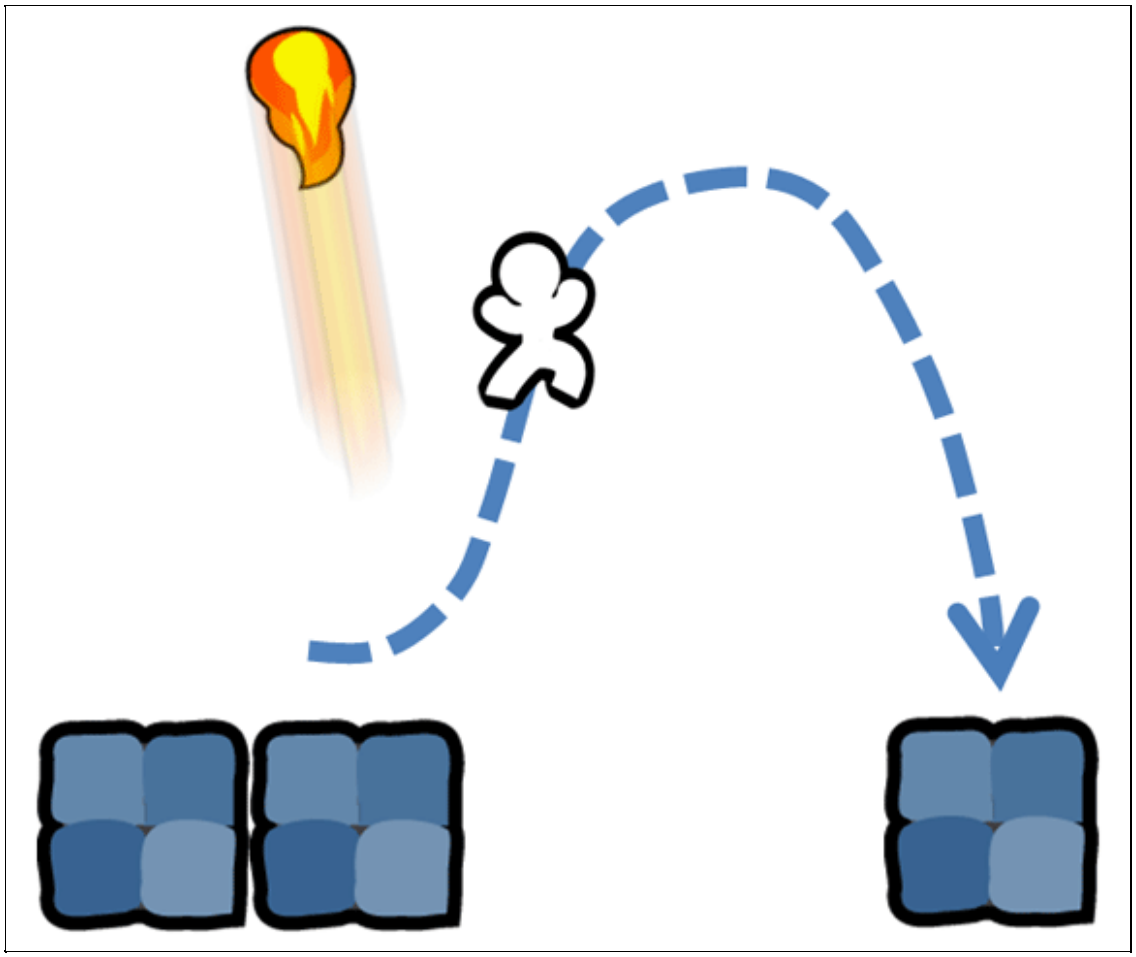
We still have a problem, though. How do we actually make a feasible level in the first place? Like NP-complete problems, it seems like it's easy to verify a solution, but very hard to find the solution to begin with.[1] What we need is for the AI designing the levels to itself have some notion of what is and isn't possible.

The simplest way to do this is to give the AI knowledge about the player physics. Starting with a player standing on a block, we can pre-compute all possible destinations a player may arrive at by jumping in different directions.



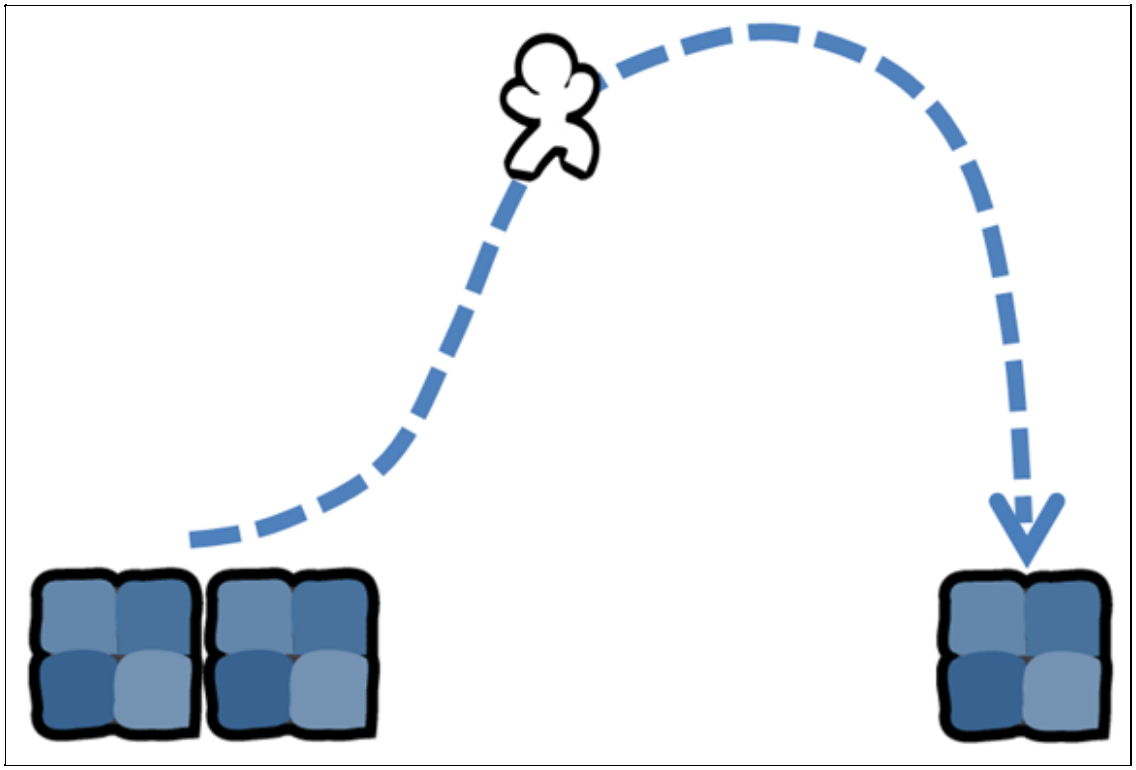
Enumerating possible destinations.

The AI then takes this information and uses it as a constraint. Each block it places must be within a certain range of some other block, dependent on the relative heights of the blocks. For simple player physics this is a good model, but if the physics also has momentum, friction, and variable jumping heights, then the pre-computation suddenly becomes a lot bigger. We need to know where the player can end up depending on every start configuration: running at half speed, running at full speed, doing a full jump, a half jump, etc.



Short jump

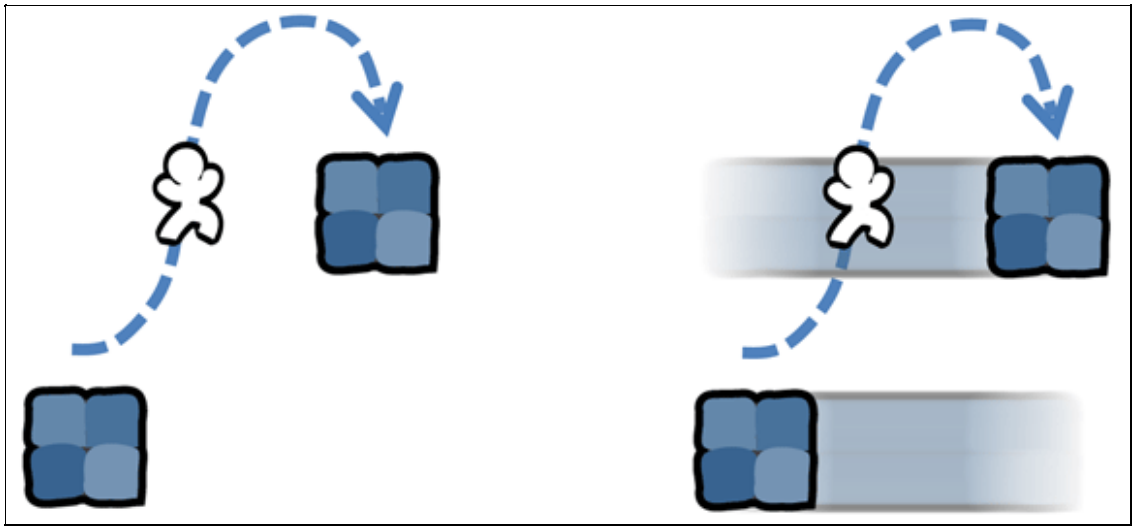
Imagine a simple situation where a player is about to run and jump from one block to another. In the first case, a passing fireball forces the player to stop before proceeding to jump. The player now jumps, starting from a standstill, with no initial momentum, retarding the full extent of the jump. The player could first backtrack to get a running start, but perhaps there is an advancing wall of doom impinging on the player.



Long jump

Now imagine a second, simpler case without the fireball. The player can run at full speed and can clear a longer jump. Good for the player, bad for the AI designer. Now it's not enough for the AI to know the relative positions between pairs of blocks, the AI must also know what the player context of each block is. The AI needs to know what state and situation a player will be in when the player is on Block A, so that it can calculate how far away it can place Block B.

Unfortunately, things are even more complicated than this. It turns out it's not enough to know just the player's state and how far the player can jump in different states. Imagine another simple situation, where a player is jumping from a lower block to a higher block. In the first case, the blocks are stationary, and the player can successfully clear the jump. In the second case, the blocks are moving in such a way that even though the final position of the block when the player intends to land is within jumping range, the block itself intersects the player's path earlier on in the jumping arc.



Left: valid path. Right: invalid path.

Suddenly we need to keep track of the player context, the range of possible jumps, as well as how all possible player trajectories interact with every block we place and even *intend* to place. It may turn out that an obstacle we place at the end of a level affects the player's path at the beginning of the level. This is known as a dense problem. Dense in the sense that where we should place each object in our universe is intimately dependent on the location of every other object, forming a dense tangle of messy dependencies.

---

[1] We could just randomly sample levels until we find one that works, but presumably the space of provable levels is much smaller than the space of all levels, which would make this method impossible in practice. I wonder what a truly random sample from the space of provable levels looks like, though.

Luckily, our design AI has an ally: the player AI. Suppose we have a partially complete level that is known to be feasible up to its current endpoint (that is, a player can navigate from the beginning to the impromptu end). The design AI considers placing a new block, to extend the level further.

It queries the player AI as to whether the new block interferes with the feasibility of the existing level, as well as whether it's possible for the player to even get to this new block. If the player AI gives the okay, the design AI places the block and continues. Otherwise, the design AI throws the block in the recycle bin and tries something else. We repeat this process until we have arrived at our desired destination.

More specifically we represent the space of all possible levels as a tree, where the first node is the empty level, and child nodes are equivalent to parent nodes with the addition of one more block. The design AI then follows a depth-first search of the tree.

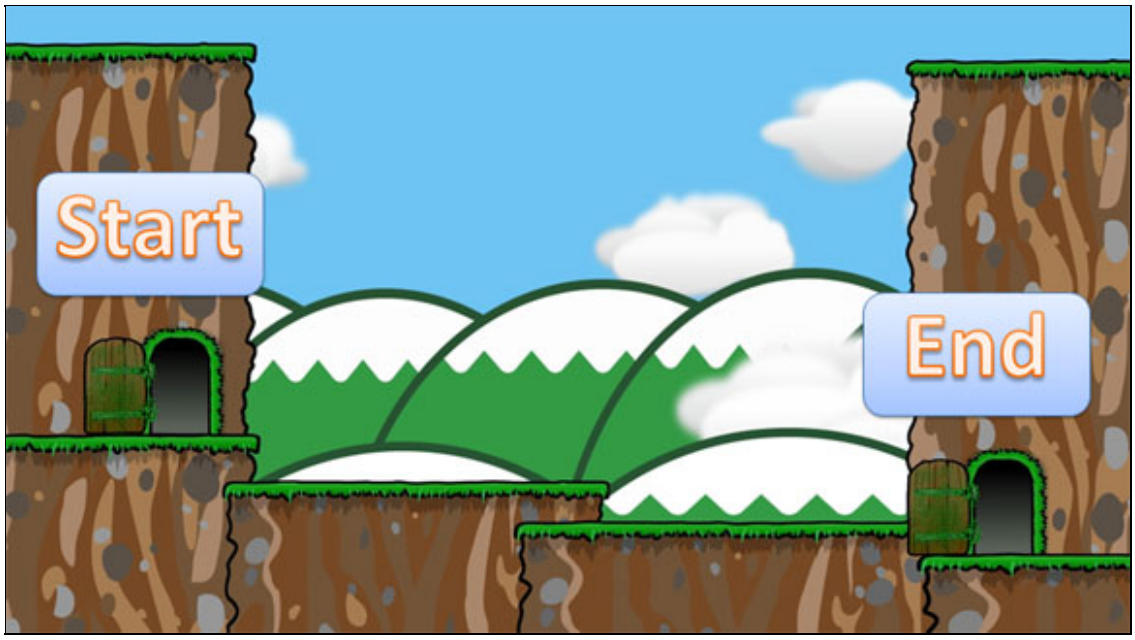
Eventually (theoretically) the search arrives at a leaf node of the tree, which corresponds to a completed level.[2]

The result is a collection of blocks, which we know comprise a feasible level. With this skeleton in hand we can spruce things up by adding obstacles throughout the level, making sure not to add any obstacles that interfere with the player AI's successful path through the level.

Now that we have an algorithm to make beatable levels, it's time to tweak it to hit our other design points.

Design Requirement #2: It Must Be Interesting

It's actually pretty easy to satisfy feasibility if we don't care about anything else. With slightly less than two minutes of coding, I whipped up a brand new procedural level generator. Behold its output:



A most excellent level.

Each level is completely new, never before seen -- and completely uninteresting!

Okay, so we want a little more out of our algorithm than that. I wish I could say there was a magic bullet for making interesting levels: a mathematically elegant way to prescribe flow, spacing, rhythm, and player expectation within a level; a simple algorithm that constrains our search amongst the uncountable possible levels to only those levels worth playing. Such a powerful insight may exist, but I have not found it. The good news is that the problem is amenable to a brute force attack.

The general principle I've followed is to make the design algorithm as generic as possible. When faced with a technical choice amongst implementation details, I implement both with a parameter to control which implementation is active. The same principle applies to the parameters themselves. Should the parameter be a simple Boolean, fixed for any given level, or should it fluctuate throughout the construction of a level? I don't want to make a decision between these two choices, so instead let's make another parameter that controls how the first parameter should behave.

The solution sounds abstract, so let me give some concrete examples. The design AI performs a depth first search of our level tree. The level tree is massive. Each node has millions of children. How do we select which branch to traverse down? We need a heuristic, and there are an uncountable number of possible heuristics (most of which suck).

Some of the heuristics are simple preferences, pruning our tree by imposing additional constraints. For example, a preference for blue moving blocks over green bouncy blocks, which will produce decidedly different levels compared to the opposite heuristic. Some of the preferences are more nuanced. Maybe I want a moving block, but only if the phase of its orbit has a nice relationship to the other moving blocks in the level.

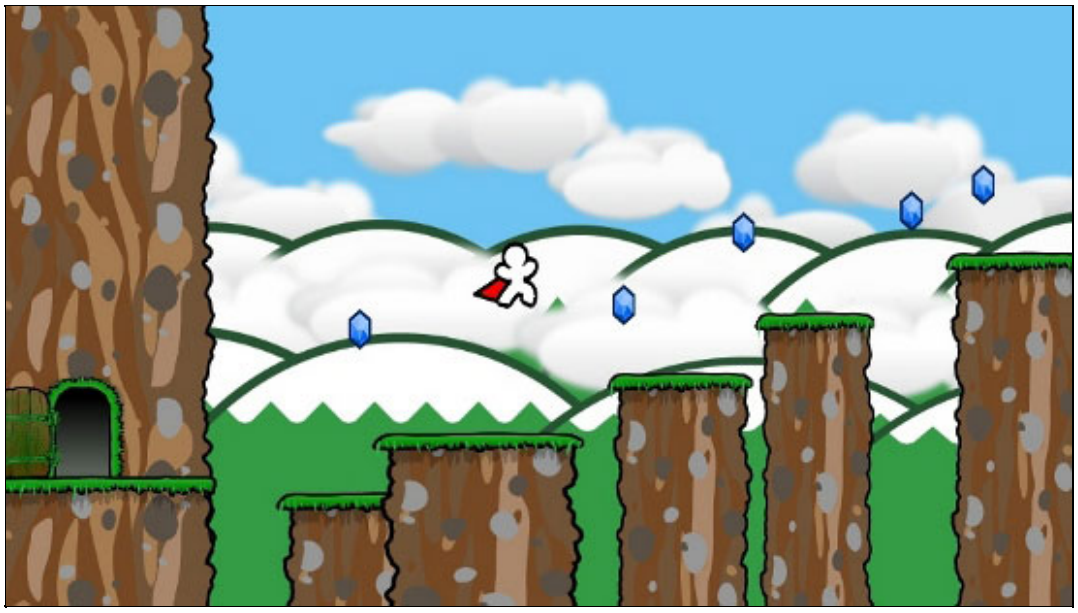
We can also impose restrictions on our player AI as we give it branches of the tree to test. Maybe we restrict the player AI such that it is never running forward at less than half speed. This leads to levels with a greater sense of flow. We can also restrict the range of motion of the AI, forcing it to avoid certain areas, leading to levels with paths that are more convoluted.

How wide should blocks be? Should there be unnecessary blocks? How many? Do we want a ceiling? Can you hit your head on it? Should it impose on the player's jumps? If we want multiple types of blocks in a level, how do we choose between them at any given location, etc. etc.? For every question, there is a parameter (and often meta-parameter) to provide an arbitrary answer.

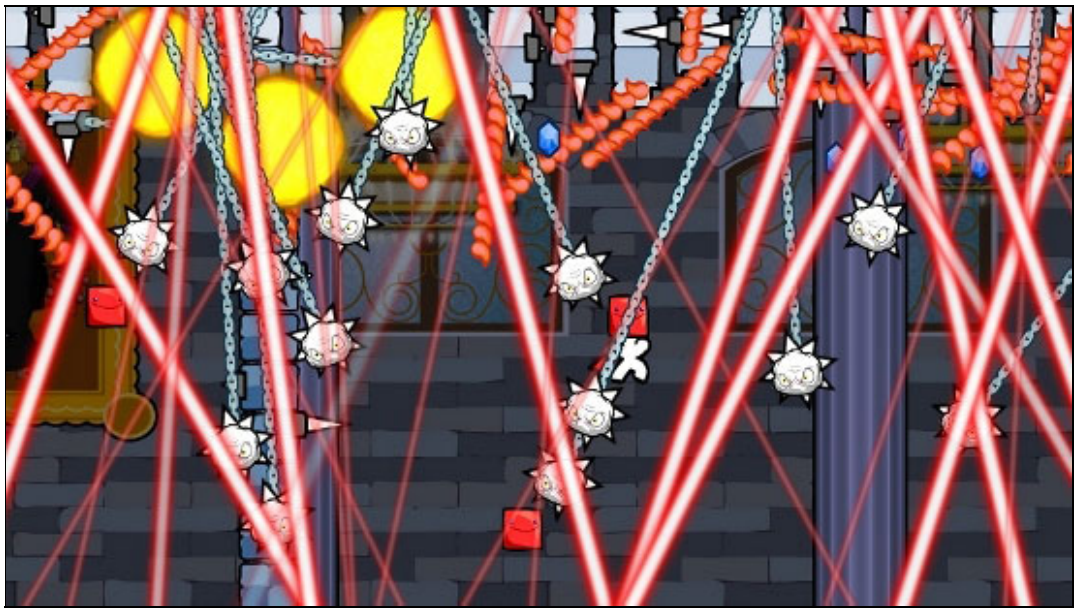
There are hundreds of parameters buried in the depths of the algorithm. Each parameter interacts with every other parameter through the level creation process, creating a seething, chaotic pool of coupled variables. I no longer have any idea exactly how changes to them will manifest in the final product. The system is too complex. Sometimes, mysteriously, levels will start looking very uninteresting, maybe ugly, or worse: not fun. The resulting debug sessions aren't really about debugging so much as black magic.[3]

Design Requirement #3: It Must Be Challenging

Ultimately, no matter how interesting a level looks or whether or not it's possible to beat, a level is useless if it doesn't provide a challenge to the player. A challenging level must hit the sweet spot between easy:



and masochistic:



Remember the hundreds of parameters we created to make our level interesting? Well, it turns out we need thousands of parameters to control for difficulty. As before, many of our parameters are answers to questions.

How far apart are blocks? How close to the edge of a block do you need to be before jumping? Do you have to use the full height of the jump, or can you use a half jump? Are there fireballs? How many? How fast are they? How close should a player ever have to get to one? Are they all moving in the same direction, or are there multiple axis-aligned directions (hard), or even multiple arbitrary directions (harder)?

Do fireball emitters all fire at the same time? What if we have 10 different obstacle types? How should they mix and match? If there is a lot of jumping in a level, and a player is more focused on completing jumps correctly, should we decrease the amount of obstacle evasion required? Should there be safe spots? Should players need to stop and evaluate, or should they be forced to continue running at full speed?

Making all these parameters is the easy part. The hard part is setting them correctly. At one point, I toyed around with a genetic algorithm for evolving the parameter values, but I stopped myself when I realized I was putting a dangerously opaque optimization method on top of a stupidly delicate constraint satisfaction engine. Instead, I captured an army of beta testers to do my bidding (in exchange for pizza and beer). Three years and thousands of man-hours later and the parameter space has finally been carved up into well-defined regions, from easy up to humanly impossible, with a continuous scale in between. Now there is a single parameter to rule them all: *difficulty rating.*

A player can now sit down and dial up a level to any difficulty, or she can just sit back and let the game pick levels for her. There are lots of ways the game can pick the correct challenge for a player. It can adaptively adjust the difficulty

as it observes the player's death rate, or it can provide a simple ramped difficulty of ever more difficult levels. You can even adaptively change the level difficulty on the fly, as a player plays it, although in practice we've found it decreases the player's sense of accomplishment.

---

[2] Sometimes a node has no descendants that contain paths to our desired end point. The AI then deletes the last block it created and proceeds down a different branch of the tree. It's possible that the AI may need to backtrack multiple steps.

[3] I can't stress enough how important good tools are, especially good visualization tools. For any level I can look back and see exactly how the design AI created it. I can step through the depth first search of the tree, analyze the heuristics used, and visualize the current state of the level at every step. Without this, I would have been shooting in the dark.

Why Should You Make a Random Platformer?

I must admit, I originally took on this project for no other reason than that I wanted to see insane *Mario* levels. Hectic, crazy, ridiculous *Mario* levels, densely filled with Bullet Bills and fire spinners and so on; but, after I got an initial prototype built, I started to see how much more potential the engine really had.

What does procedural generation give you?

First, purely from the perspective of a designer, the game design pipeline is made much more flexible. Normally the physics and obstacles of a game are designed, followed by the levels. If you go back and change the physics or how an obstacle acts, you are forced to tweak all your levels to accommodate the change.

Essentially you must lock in your physics before proceeding. With a procedural algorithm all that changes. I can change the physics at any point in the development cycle and all the levels will automatically take the new physics into account. In fact, we are even shipping a physics editor with our game so that players can tinker with the physics and have the design AI create levels for it.

More importantly though is the question, what do random levels give the player? The obvious answer is replayability, but in my opinion that is not that important. Here's a list of a few of my favorites:

Challenge. No matter how good a player gets, the algorithm can make a level that will challenge her. Likewise, if a player is more casual, the algorithm can cater to her skill level.

New game types. We can make game types that *depend* on random levels, rather than just utilizing them. For instance, imagine an infinite string of levels strung together in a *Tetris*-like progression of difficulty. Without randomness, this game would devolve into a memorization challenge. With randomness, the focus shifts toward the player's skill, rather than muscle memory.

Customizability. Players can tinker with the algorithm's parameters, creating levels built to their specifications. There is a vast universe to explore.

Uniformity. This is opposite of customizability, but it's an important tool. We can fix the algorithm's parameters but still get an infinite string of levels, all very similar in feel, but different nonetheless. For instance, we can fix the "flow" of a level, so that each level can be beat without pausing or backtracking. Maintaining this consistency can help a player get in the zone, although it has to be balanced against trying to keep the player interested.

Why Shouldn't You Make a Random Platformer?

There's no doubt that random levels are going to yield a different experience than hand-crafted levels. I don't claim that they are superior, but I hope I have convinced you that they need not be inferior. They're different, and they may not be what you want for your game's experience. For us, we are focusing on a very tight, fast-paced arcade experience, and we have been well served by our algorithm. If you are focusing more on puzzle solving, or Easter eggs, or unique gimmicks in each level, then random levels may or may not be the right tool for you.

There is one other good reason not to make a random platformer.

IT'S HARD.[4] We've been in development for three years (which is a silly amount of time for a platformer). I spent four months on pure exploratory research and an unknown number of months maintaining the ever-growing monolithic beast.

But, it's been worth it. I can sit down and design a new hero, specify how fast he runs, how high he jumps, friction, etc. Then I can hand it to the AI and I'm off playing a new adventure, engaging in levels designed exactly for my new hero, which I've never played before. As a designer, it's an amazing feeling to see a new world unfold from your creation. I hope it is an amazing feeling for the players as well.

---

[4] Hard for me, at least. I hope sometime in the future an elite hacker will implement this algorithm to the max, making the most epic platformer in the history of the world.

Appendix: Pseudocode Example

```
// Make an empty level, and add a start and end platform.
Level = EmptyLevel();
Level.AddBlock(StartBlock);
Level.AddBlock(FinalBlock);
// Create a feasible path between the start and end platform.
CompleteLevel(Level);
// Recursive method to take an incomplete level and make it feasible by adding more blocks.
bool CompleteLevel(Level)
{
    // We will keep track of blocks we tried which turned out invalid.
    ListOfBlocksTried = { };
    // Keep trying different blocks until we find one that is valid.
    // If we try too many blocks, perhaps the current level can not be made feasible, so return to the parent node.
    while (ListOfBlocksTried.Length < 10)
    {
        // Choose a block to try and add it to the level.
        NewBlock = Heuristic(Level, ListOfBlocksTried);
        Level.AddBlock(NewBlock);
        // Check to see if we can reach the new block.
        if (PlayerAI.BlockIsReachable(Level, StartPos, NewBlock))
        {
            // If we can reach the exit platform, we are done
            if (PlayerAI.BlockIsReachable(Level, StartPos, FinalBlock))
            return true;
            else
            {
                // Otherwise, recursively proceed down the tree. If the recursive call returns true, then our new blocks is a good choice.
                if (CompleteLevel(Level)
                return true;
            }
        }
        // Either the new block wasn't reachable, or it led to a level that couldn't be made feasible with additional blocks.
        // Either way, remove the new block and add it to our list of used blocks, so we don't use it again.
```

```
            ListOfBlocksTried.Append(NewBlock);
            Level.RemoveBlock(NewBlock);
        }
        return false;
    }
```

Return to the full version of this article