The Problem with Braid

Aleksa Vujičić

July 4, 2022

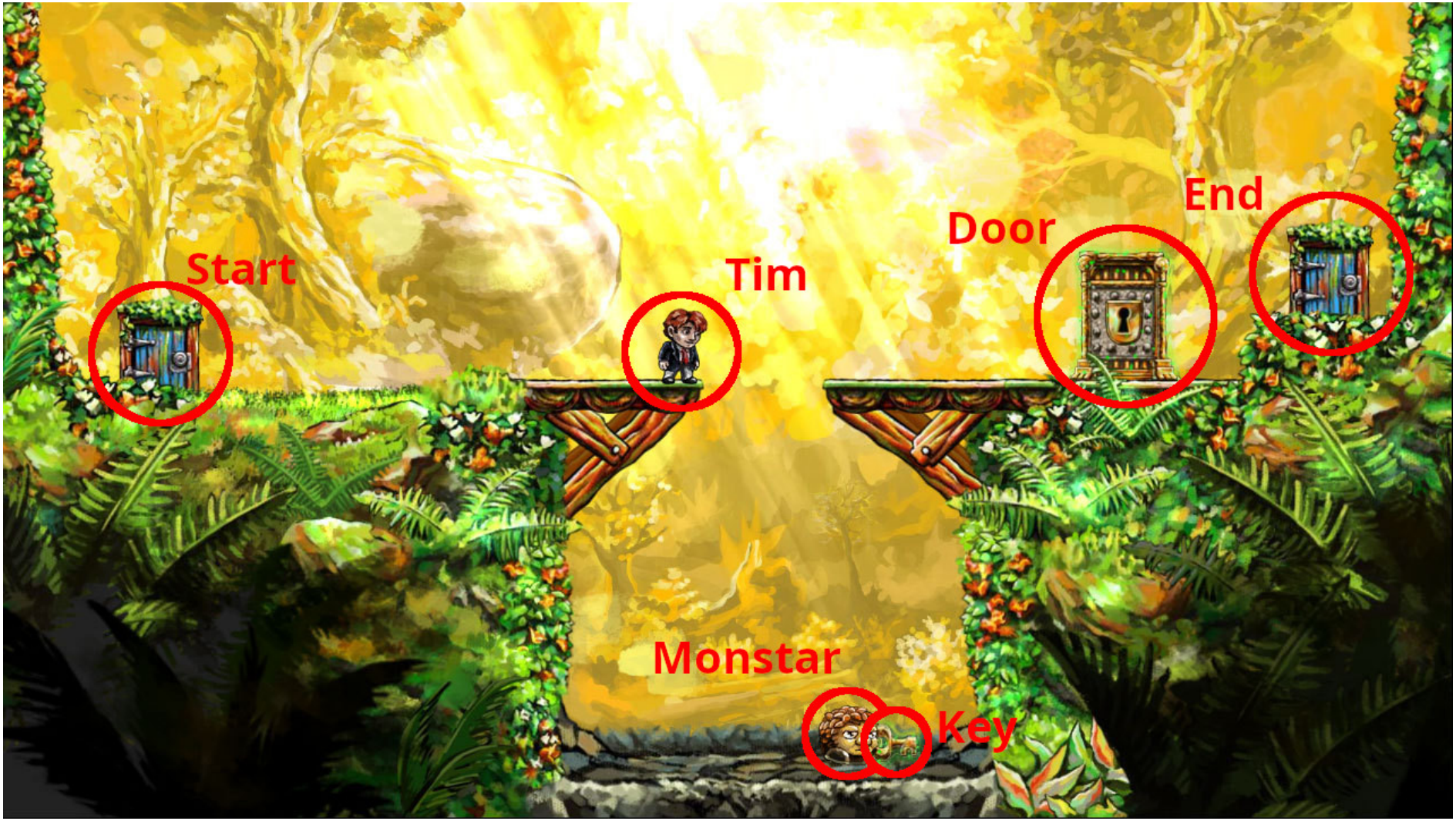
Overview

- Based on a paper ‘Braid is Undecidable’ by Linus Hamilton.
- First we will have a brief look at Braid.
- Then we will define the main question of this talk.
- We then make some weird and crazy Braid levels.
- And then look at some introductory computability theory.

Braid - What is it?

- Braid is a video game by Jonathan Blow, released in 2008 .
- You control a guy named Tim.
- You have to get Tim to the end of the level.
- Lots of cool time shifty stuff.
- Seriously go play it.

Gameplay Elements



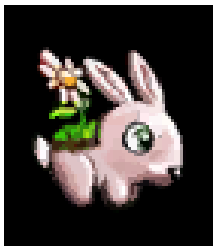
Braid lets play

- Braid gameplay video[1]

[1] https://youtu.be/Y4lwJ7eHjJI

A few more things

- We've seen monstars, cannons, levers, platforms, ladders, and spikes.
- Briefly saw a one-way wall. Behaves as expected, though can be oriented in any direction.
- There is also a bunny enemy.



- They're cute.
- Monstars bounce off bunnies and kill them.

Ok, but where's the math

- Not all Braid levels are beatable - could place exit door behind a wall.
- Can we determine which levels are beatable?
- Impossible to do algorithmically!
- But why?
- Long story short is that we can build a computer in Braid.
- Interestingly - this does not use any of the time stuff.

Turing Machines

- Formally, Turing machines are a tape with zeros and ones, with several instructions (such as read/write, moving 'head', and changing state).
- A combination of these instructions constitutes a program.
- Informally speaking, they can implement and simulate any computer.
- We say a machine is 'Turing complete' if it can simulate a Turing machine.
- Most modern day programming languages are Turing complete.
- Many other examples.

Counters

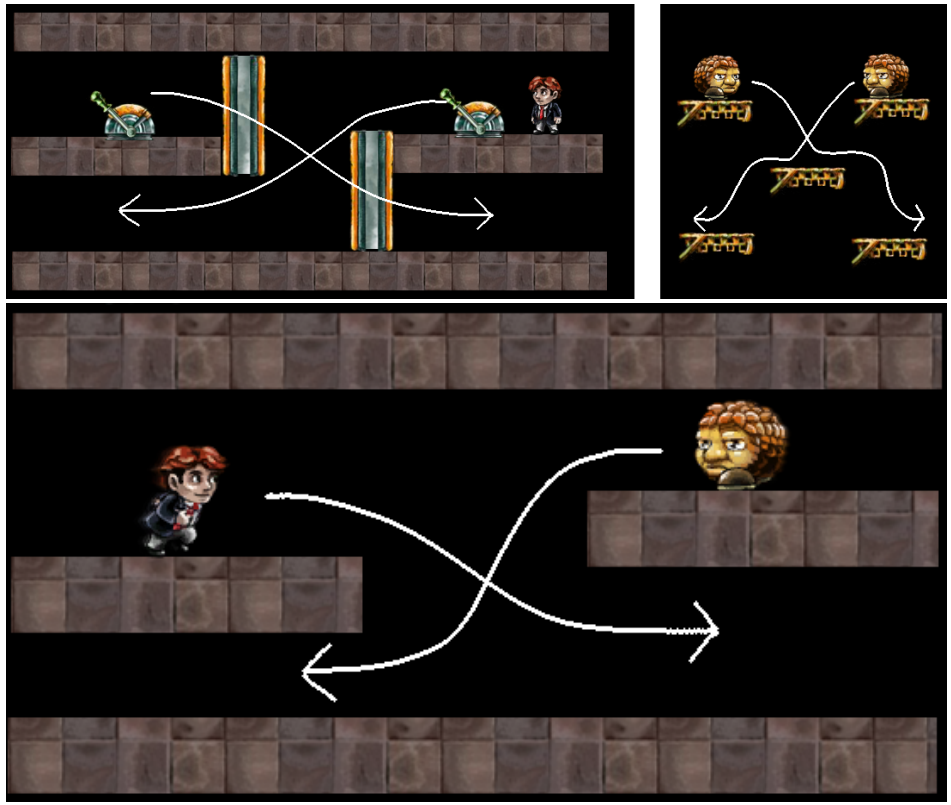
- A counter is a device that can store an arbitrary natural number.
- $(0 \in \mathbb{N})$
- Comes with three instructions:
  1. **Add** - Add one to the counter.
  2. **Subtract and Branch** - Subtract one if non-zero. If zero, then go to a specified line of code instead.
  3. **Halt** - Stop the machine.
- We can create a machine that contains multiple counters.
- Specifically, if we create a machine with three counters, this is (non-trivially) Turing complete.
- We will construct this counter machine in Braid.

Game Plan

- We will have Tim follow a ‘main path’.
- Along this path he will be forced to do some stuff (explained in a bit).
- This stuff will simulate a program running on a counter machine.
- Different programs will have different paths, but with the same components.

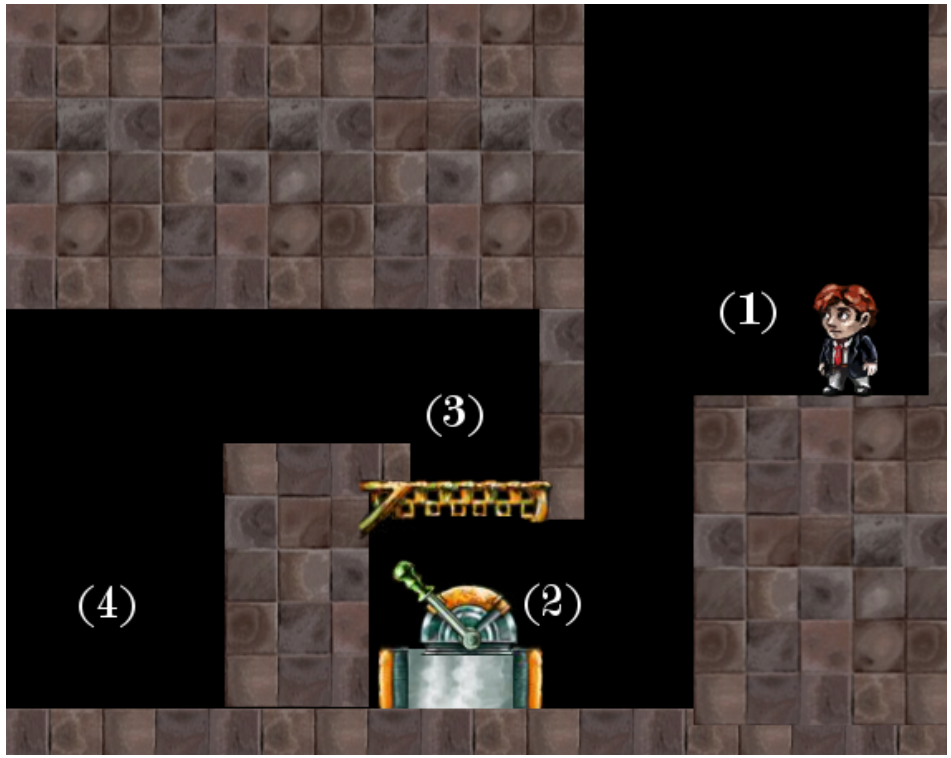
Crossover Gadget

- Three different crossings depending on who needs to cross.
- Used to prevent Tim going off the main path, and to prevent monstars going where they shouldn't.
- May require adding long bits before so that Tim and monstars cannot cross at the same time.
- Means we are not limited to a 'planar graph'.



Lever Pull Gadget

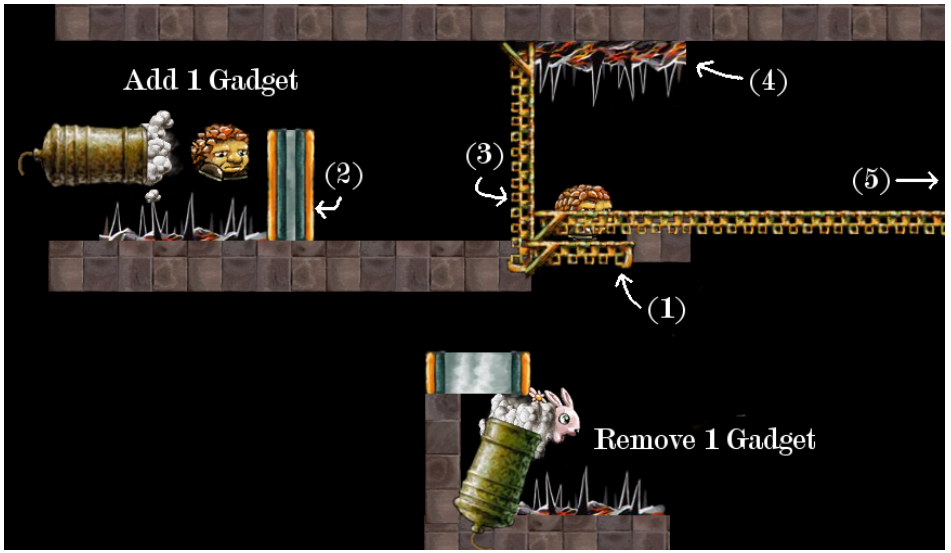
- Tim arrives from above - has to fall down.
- Only way out is to pull lever.
- Lever moves platform up then back down.
- Platform pushes Tim above one-way floor, so he cannot return.
- Tim must proceed onwards.



- Net result is that Tim must pull lever exactly once to proceed.
- Gadget self-resets so can be used multiple times.

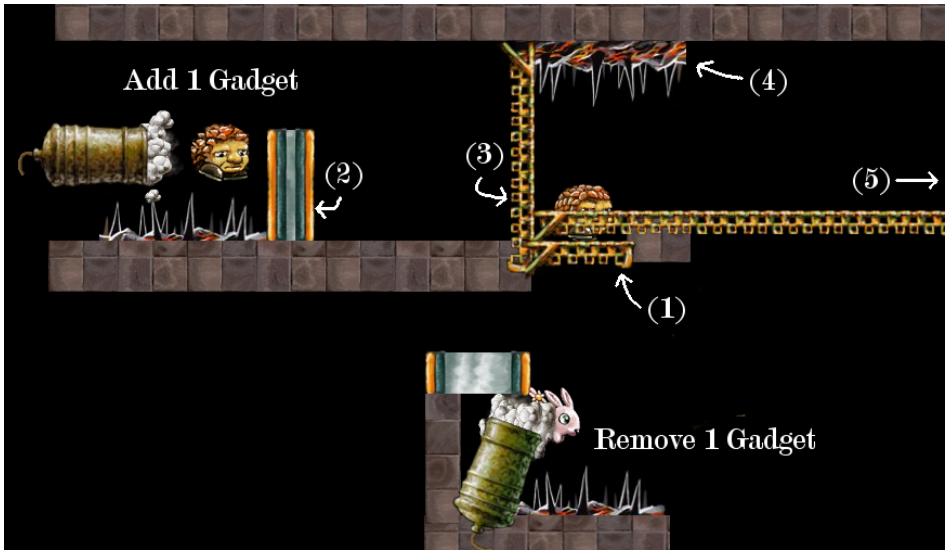
Counter Gadget (Add 1)

- (1) is a holding cell for monstars
- No collision with each other so can hold arbitrarily many.
- Cannon on left fires monstars periodically.
- A lever can pull platform (2) down long enough so exactly one monstar gets to cell.
- Otherwise monstars hit platform and die to spikes.



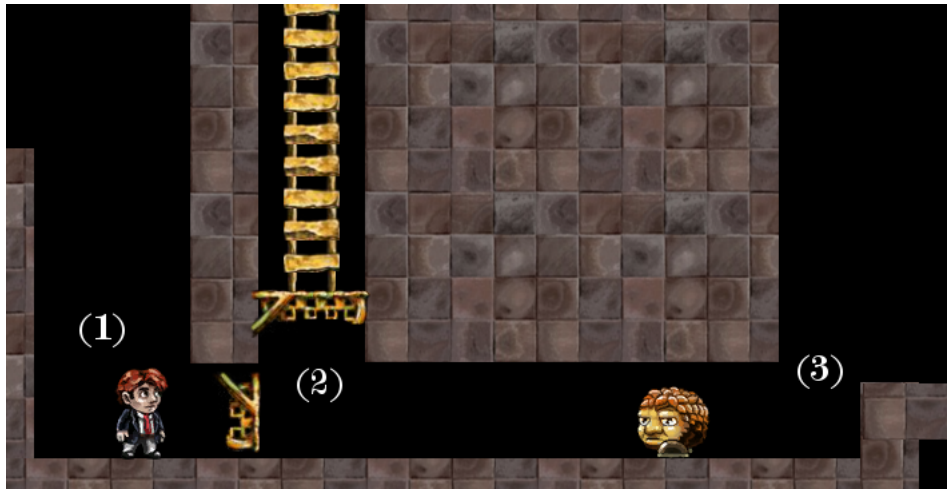
Counter Gadget (Subtract 1)

- Similarly cannon below fires bunnies.
- A lever pull sends exactly one bunny to cell.
- If the cell has no monstars, bunny flies into spikes (4) and dies.
- Otherwise, one monstar will bounce off bunny, and walk off screen (5).



Branch Gadget

- When Tim arrives, there may or may not be a monstar in the pit.
- If there is a monstar, must bounce off it to the ladder above. Will die if he tries go right.
- If there is no monstar, he cannot reach the ladder, and must go right.



- Net result is Tim branches based on the presence of monstar.
- This gadget also self-resets.

All together now

- First place three counter gadgets (as far apart as needed).
- Whenever there is an "Add" instruction, add lever gadget connected to a counter gadget.
- For "Subtract and branch", we first place a lever gadget, connected to a counter gadget.
- If there is a monstar removed, force it into a separate 'box'.
- Then add another lever gadget, which will redirect the monstar from the 'box' to a branch gadget.
- Then place this branch gadget.
- Place all these far enough apart to allow time for the monstar to walk to where it needs to be.
- Finally, for "Halt", simply place the exit door.

The Halting Problem

- We can create Braid levels which simulate programs.
- If a particular program ends ('halts'), then the corresponding level is beatable.
- And if it never halts, then that level is unbeatable.
- Can we write a program to determine if any arbitrary program halts?
- This is the *Halting Problem*

A Super-machine

- By definition, programs use finitely many symbols and are finite in length.
- So there are countably many programs - we will interchangeably refer to programs as numbers and vice versa.
- Suppose we had some program which could compute the function f, defined as follows:

$$f(i,n) = \begin{cases} 1 & \text{if program } i \text{ halts on input } n \\ 0 & \text{if program } i \text{ never halts on input } n \end{cases}$$

- I.e. This program is a solution to the Halting Problem.
- No assumption about how it works - 'black box'.

A Super-super-machine

- Define new program p:

```
def p(i):
  if f(i,i) == 0: return 0
  else: loop forever
```

- Since p is a program, we can associate it with a number j_p.
- So what happens when we evaluate $p(j_p)$?
- If $f(j_p, j_p) = 0$, then it $p(j_p)$ returns 0. But $f(j_p, j_p) = 0$ means that program p on input j_p will loop forever.
- Similarly, if $f(j_p, j_p) = 1$, then it loops forever. But $f(j_p, j_p) = 1$ means that program p on input j_p will halt.
- Contradiction!
- Hence no program can compute the function f.

The Problem with Braid

- We can create levels in Braid, which force the player to simulate Turing machines.
- These levels are beatable precisely if the corresponding Turing machine halts.
- However the Halting Problem is undecidable.
- That is, we cannot write a program that will always determine if any Turing machine will halt.
- So we cannot write a program that will determine if any Braid level is beatable or not.

Other Games

- Our levels made no use of Braid's time mechanics.
- Key component are cannons - can store an arbitrary amount of monstars.
- What about other games? Do they also have undecidable problems?
- True for any game that can simulate a Turing machine - Conway's game of life, Minecraft, Magic the Gathering.
- Also Team Fortress 2, Super Smash Bros: Brawl, Mario Kart all have undecidable problems[2].
- Also look at the original Braid paper for other cool complexity theory stuff.

[2] https://drops.dagstuhl.de/opus/volltexte/2018/8805/pdf/LIPIcs-FUN-2018-14.pdf "Cooperating in Video Games? Impossible! Undecidability of Team Multiplayer Games"

End

Thanks for listening!