Chapter 6
Rules and mechanics

Mark J. Nelson, Julian Togelius, Cameron Browne, and Michael Cook

Abstract Rules are at the core of many games. So how about generating them? This chapter discusses various ways to encode and generate game rules, and occasionally game entities that are strongly tied to rules. The first part discusses ways of generating rules for board games, including Ludi, perhaps the most successful example of automatically generated game rules. The second part discusses some more tentative attempts to generate rules for video games, in particular 2D games with graphical logic. Most approaches to generating game rules have used search-based methods such as evolution, but there are also some solver-based approaches.

6.1 Rules of the game

So far in this book, we have seen a large number of methods for generating content for existing games. If you have a game already, that means you can now generate many things for it: maps, levels, terrain, vegetation, weapons, dungeons, racing tracks. But what if you don’t already have a game, and want to generate the game itself? What would you generate, and how? At the heart of many types of games is a system of game rules. This chapter will discuss representations for game rules of different kinds, along with methods to generate them, and evaluation functions and constraints that help us judge complete games rather than just isolated content artefacts.

Our main focus here will be on methods for generating interesting, fun, and/or balanced game rules. However, an important perspective that will permeate the chapter is that game rule encodings and evaluation functions can encode game design expertise and style, and thus help us understand game design. By formalising aspects of the game rules, we define a space of possible rules more precisely than could be done through writing about rules in qualitative terms; and by choosing which aspects of the rules to formalise, we define what aspects of the game are interesting to explore and introduce variation in. In this way, each game generator
can be thought of as an executable micro-theory of game design, though often a simplified, and sometimes even a caricatured one [32].

6.2 Encoding game rules

To generate game rules, we need some way of representing or encoding them in a machine-readable format that some software system can work with.\(^1\) An ambitious starting point for a game encoding might be one that can encode game rules in general: an open-ended way to represent any possible game. The game generator would then work on games in this encoding, looking for variants or entirely new games in this space. But such a fully general encoding provides a quite unhelpful starting point. A completely general representation for games cannot say very much that is specific about games at all. Some kinds of games have turns, but some don’t. Some games are primarily about graphics and movement, while others take place in an abstract mathematical space. The only fully general encoding of a computer game would be simply a general encoding for all software. Something like “C source code” would suffice, but it produces an extremely sparse search space. Although all computer games could in principle be represented in the C programming language, almost all things that can be represented in C’s syntax are not in fact games, and indeed many of them are not even working programs, making a generator’s job quite difficult.\(^2\)

Instead of having a generator search through the extremely sparse space of all computer programs to find interesting games, a more fruitful starting point is to pick an encoding where the space includes a more dense distribution of things that are games and meet some basic criteria of playability. That way, our generator can spend most of its time attempting to design interesting game variants. Furthermore, it’s helpful for game encodings to start with a specific genre. Once we restrict focus to a particular genre, it’s possible to abstract meaningful elements common to games in the genre, which the generator can take as given. For example, an encoding for turn-based board games can assume that the game’s time advances in alternating discrete turns, that there are pieces on spaces arranged in some configuration, and that play is largely based on moving pieces around. This means the game generator does not have to invent the concept of a “turn”, but instead can focus on finding interesting rules for turn-based board games. An encoding for a side-scrolling space shooter would be very different: here the encoding would include continuous time;

\(^1\) There are many other uses for machine-readable game rules, such as for use in game-playing AI competitions [12, 6] and in game-design assistants targeted at human game designers [20, 9]. This chapter focuses on encodings for generating rules, but multi-use encodings are often desirable.

\(^2\) This is not to say generating games encoded as raw programs would be impossible: genetic-programming techniques evolve programs encoded in fairly general representations [29], and applying genetic programming to videogame design could produce interesting results. But the techniques in this chapter focus on higher-level representations, which allow the generators to work on more familiar game-design elements rather than on low-level source code.