CHAPTER 8

Game Design: Levels and Features

Even the most captivating game ideas will eventually become dull if the challenges never change. Good level design is about providing players with new and interesting challenges as they progress through the game. Usually this requires a steady flow of fresh features that introduce new equipment, opponents, enemies, or obstacles into the gameplay. Creating the right kind of features is as important to good level design as the way those features are used in the levels themselves. In this chapter, we'll take a look at the issues surrounding level design and try to give you some pointers for designing your own games.

Selecting Features

So you've created a game prototype with a solid core mechanic and carefully tweaked the gameplay with the help of your friends. Everyone thinks it could be really good and you just need another 25 levels to turn it into a finished game. Sitting down with your designer's notebook, you begin to make a list of all the extra features you want to include—but where do you start, and how do you tell a good feature from a bad one?

The good news is that coming up with plenty of ideas is not usually a problem at this stage. If you start running dry, try showing your game to some of your game-playing friends. Providing your basic mechanics work okay, they'll usually start pouring forth their own suggestions for how you could expand upon it. The bad news is that even a game with amazing potential can end up being dreadful if you don't get the levels and features right. Learning to distinguish a feature idea with potential from a "nonstarter" is an essential skill for a designer to develop, so in this section we'll pass on a few tips.

Features can be divided into two groups: those that change the abilities available for the player to use, and those that change the level obstacles that the player has to overcome. Here's a list of some different types of features that you might consider including:

- New features for the player(s) to use:
    - Abilities (e.g., attack moves, swimming, flying)
    - Equipment (e.g., weapons, armor, vehicles)
    - Characters (e.g., engineer, wizard, medic)
    - Buildings (e.g., garage, barracks, armory)

- New level features for the player(s) to overcome:
  - Opponents (e.g., with new abilities, buildings, or equipment)
  - Obstacles (e.g., traps, puzzles, terrain)
  - Environments (e.g., battlefields, racing tracks, climate)

A game doesn't need to include all the types of features listed here, and the first step is to decide which types you want to include in your game. We're going to use the Koalabr8 game from the previous chapter as an example. This game only includes obstacle-based features, but there's no real reason why it couldn't include new player abilities, equipment, or characters as well. However, we decided to stick to obstacles and save these other options for a sequel. Making these kinds of choices from the start can help to focus your game design and give it a clearer identity. If you try to include too many different types of features, you run the risk of creating a mishmash of ideas with no identifiable strengths.

So before you begin thinking of feature ideas, give some serious thought to the kind of progression you want to see running through your game. Is it about developing the equipment and abilities of a single character or building up a specialist team? Will levels force the player to apply their skills in different kinds of environments, or just pit them against increasingly challenging obstacles? Once you've made some of these kinds of decisions, then you're ready to sit down with some friends and start brainstorming ideas.

Pie in the Sky

> *So the koala picks up this jetpack, right, and everything goes 3D and you have to start flying it through the maze at like 1,000 miles an hour . . .*

Whenever you ask people to brainstorm for you, you can guarantee that many ideas will fall into this category. Faced with such a creative opportunity, it's easy to let enthusiasm get the better of us—especially if we're not the one who has to do all the work! However, as the game's designer and programmer, it should be easier for you to spot ideas that are impossible or just too much work. Be gentle with your collaborators and try to bring them down to earth without

too much of a bump. If you reject all their ideas before they finish their sentence, they'll quickly stop contributing! Explain how the same idea might be turned into something more feasible, and don't dismiss any ideas until you've finished the brainstorming process. It's worth keeping everyone brainstorming, as it's often the case that people who have the craziest ideas eventually come up with a brilliant idea that no one else would have thought of.

Do You Have That in Blue?

> *Yeah—on some levels we could have like, plastic explosives! Awesome! They'd be like the TNT, but sort of plastic looking . . .*

Equivalent features are another kind of idea that designers need to be wary of. These are features that may look different, but actually perform a very similar function to an existing feature. There's nothing wrong with including equivalent features, as long as you don't try to pretend that they're anything more than that. For example, imagine that halfway through the Koalabr8 levels we introduced plastic explosives alongside TNT. As with any new feature, the player would approach them cautiously and try to figure out how they work. One way or another, they would eventually discover that they're exactly the same as TNT and feel slightly stupid and cheated by the whole experience.

Instead, imagine that we decided to theme the game so that half the levels were set in the doctor's workshop and half the levels were set in his laboratory. Every obstacle in the workshop could then have an equivalent high-tech feature in the laboratory setting: the TNT becomes plastic explosives, the rotating saw becomes a laser, the padlocks become electronic doors, and so forth. This approach would make the equivalence more obvious to the player and reduce the chances of their expectations being falsely raised. This way, they can appreciate the visual variation as something they have gained rather than something they have lost because they were expecting more from it.

Starting an Arms Race

> *Of course, once the koalas get into their nuclear tanks, then the TNT and circular saws can't hurt them any more . . .*

If you plan to improve the player's equipment or abilities, then you need to consider the escalating effect this can have on the rest of the game. Everybody loves the idea of über weapons and, if you're aware of the pitfalls, there's no reason why they can't be a great addition to your game. However, problems can arise because über weapons have a tendency to make all previous features redundant. Having a nuclear tank that's invulnerable to TNT and saws would be a lot of fun for one level, but where do we go from there? We need something to challenge the player on the next level, so we invent equivalent features that work against tanks (perhaps land mines and robot drills). Of course, land mines and robot drills will be no use against the koala's alien flying saucers on the following level . . .

Hopefully, you can see that it's possible to end up in an arms race between the player and the game in which the features get increasingly powerful but the gameplay doesn't actually progress at all! The player faces exactly the same challenges with a tank and a land mine as they do with a koala and TNT.

You may think that taking the tank away from the player will solve everything, but this is like giving a child a new toy only to snatch it away again moments later. A better way is to create features that are less powerful in the first place by including a trade-off in their design. We'll discuss the idea of trade-offs more in the next design chapter, but we could achieve this by making the tank just *resistant* to TNT (perhaps it could survive three hits) as well as only moving at half the speed of the koala. Now the tank becomes a tool that can be helpful in certain situations (clearing paths), rather than something that the player wants to use all the time. More important, it's now a feature that varies the challenges of the game and provides a new angle on the gameplay—rather than just repackaging an old one.

One-Trick Ponies

> *On this one level, the doctor's pet psycho-whale swallows all the koalas and they have to escape from his intestines before they get digested!*

As a game developer, you should start measuring every potential feature in terms of the quantity and quality of gameplay it creates. It's easy to end up with many more ideas than it's possible to create, so you must carefully choose the features that provide the greatest return. In particular it's often a good idea to avoid "one-trick ponies"—features that would undoubtedly be cool on one level but couldn't be used on any others. You may actually decide that the psycho-whale level is such a quality idea that you don't mind only using it once (as a final level, for example). However, be aware that you might be able to create a number of features in the same time that can be used on several different levels. In general, such ideas can be a bonus to a game that is already brimming with features, but are probably best avoided until then.

Emerging with More Than You Expected

No—you're not supposed to use it like that . . . oh—cool!

Game designers like to talk about *emergence* as it's one of the most exciting (and potentially dangerous) aspects of game design. Emergence is what happens when the rules of a game interact with each other so that the player can do something the designer hadn't predicted. Emergence can either be an uplifting surprise or terrible shock, depending on the situation. It's difficult to know whether a game contains emergence, unless you know what its designers intended, so we'll use examples from a PlayStation game called Hogs of War, about which we have insider knowledge.

Hogs was a comical war game where players took turns taking potshots at each other's squad until one side was wiped out (see Figure 8-1). There were plenty of crazy weapons to choose from, and surviving pigs could be promoted to specialist roles within their teams. The final level of the single-player game pits the player against the unstoppable "Team Lard" in the ultimate battle for control of the "swill mines." At the start of this level, the battlefield appears to be defended by a single enemy pig hauled up in a pillbox, but major reinforcements soon arrive on the scene. We designed this level to provide the player's biggest challenge yet, requiring them to use all their skill and knowledge to win the battle. However, in practice we also got two different emergent solutions that we never expected.

The first of these was an example of "good emergence," where the player used their ingenuity and lateral thinking to win the game fair and square. Rather than face the enemy in a straight firefight, they turned to the landscape to help them gain an advantage. The player's weakest pigs were quickly wiped out on this level, but they could be sacrificed to knock enemy pigs into pools of poisonous grime that surrounded the mines. The player could then use pigs trained in espionage to hide from the enemy until they all died from the effects of poison. Okay, so it wasn't the most valiant solution, but it was a clever one and it emerged from the interaction between the poison and hide features in the game.

The other emergent solution was an example of "bad emergence" as it exploits an interaction between oversights in the game's programming in order to complete the level. It was possible to use a jetpack to land a commando pig on the roof of the enemy pillbox in the very first turn of the game. The game's programming wouldn't normally allow its pigs to leave the safety of a pillbox, but another pig standing on its head (so to speak) confused it into coming out and attacking the commando. Out in the open, the player could use their best attacks to quickly kill it—before the reinforcements arrived. Since there were now no enemy pigs left on the battlefield, the game's programming concluded that the player had won the battle and finished the game. Using this method, the final level could be completed in two turns and without the player's team ever having to fight the reinforcements!

Of course, the problem with emergence is that you can't deliberately design it into a game; otherwise it wouldn't be emergent! Nonetheless, you can create features that encourage interplay with other game features and hope that good things come out of it. At the same time, you need to make doubly sure that this interplay doesn't reveal oversights in your programming that create the "bad" rather than the "good" kind of emergence. Fortunately, there's another kind of emergence between the programmer and level designer that's much easier to control, and we'll discuss it more in the section "Emerging Springs," later in this chapter.

Figure 8-1. *A Commando pig encourages an enemy Grunt to take a plunge in Hogs of War (reproduced by kind permission of ZOO Digital Publishing Ltd.).*

Designing Levels

Okay, so you've whittled your feature ideas down to a manageable shortlist of those with the most potential. Now before you immediately rush off and start programming, you first need to consider how you're going to use these features to structure your levels. It might be that all the best-sounding features are too hard for introductory levels, so you need to swap a few ideas for simpler ones. Levels and features will always continue to evolve as you work on them, but it certainly helps to make an overall plan from the start. In this section, we'll discuss some of the skills that a designer can use to help them think about this process and come up with a well-structured level plan.

The Game Maker's Apprentice

The title of this book conjures up images of a young apprentice learning a mysterious craft from a slightly crazy old professor. However, if you were thinking of yourself as the apprentice in this scenario, then think again. "The Game Maker's Apprentice" is actually a reference to the *game player*, which makes you (as the Game Maker) the crazy professor! Being a good game designer has a lot in common with being a good teacher—and just realizing this can improve your approach to game design. Inexperienced designers often see players as their opponents, and try to trick or outwit them by creating impossible challenges. As Phil Wilson discovered from his own first game designs (in this book's foreword), this does not help to make games fun! A successful game designer needs to treat players like their apprentices, balancing the game's theatrical threat of failure with a true desire to train the player to master the game. Once you accept this as your goal, you'll get a lot more satisfaction from players completing the challenges you set for them, as it proves that you're a successful game designer rather than a poor opponent!

In the last design chapter, we learned that games shouldn't be too hard or too easy to complete. This is not as straightforward as it seems because players have different levels of ability, but including difficulty modes and optional subgoals can help (as we saw in the Evil Clutches example). However, level designers also need to consider how players' abilities develop as their experience with the game grows. A task that was quite challenging for the player on level 1 may have become quite easy by the time they reach level 20. Knowing something about the way that people learn really can help a designer to create levels that pitch the difficulty correctly as players progress through the game.

Learning Curves

In general, the more time we spend on a task, the better we become at doing it. Once you've mastered the basics of a new skill, it's easy to improve by spending more time practicing it. However, the more you improve, the harder it becomes to make further progress, until eventually you seem to stop improving altogether. Figure 8-2 shows a model of how this process works for something like learning to juggle. We've taken the number of successful throws and catches that a learner makes in a minute as a measure of their juggling skill. You can see from the graph (or if you've ever tried it) that it takes quite a lot of effort to first grasp the technique for juggling, and it may be a long time before a beginner can make even a dozen successful catches in a single minute. Nonetheless, sooner or later they will get the hang of it and their catches per minute will start to increase rapidly for a while. However, once they reach the stage where they rarely drop balls, it becomes a lot harder to make further improvements. Their skill will still increase as they refine their precision and technique, but these improvements will get smaller and it will take longer to increase their catches per minute by the same amount.

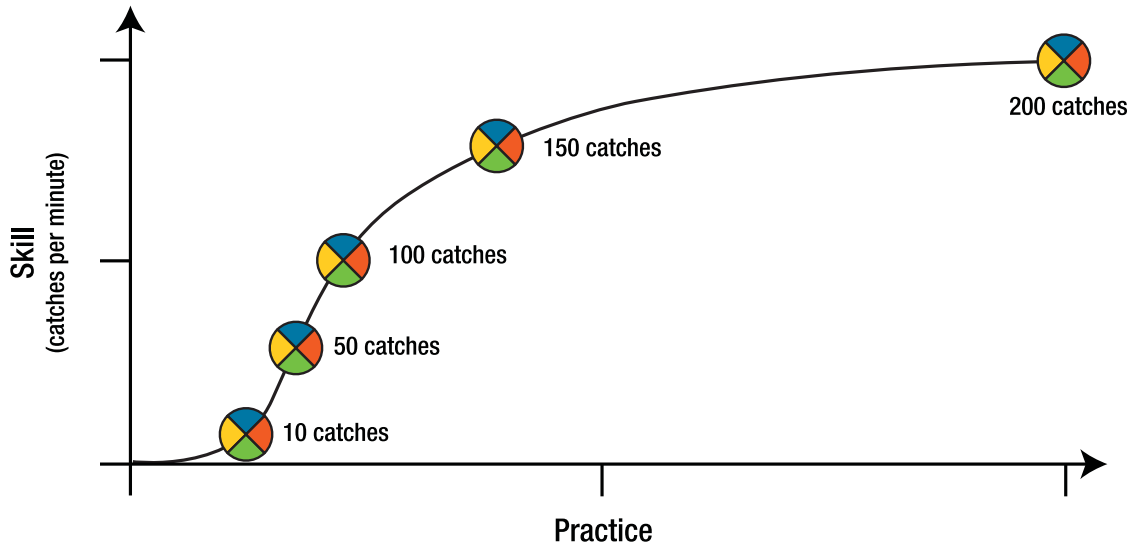

Figure 8-2. *A learning curve shows how a learner's performance changes with practice.*

Learning curves apply to computer games, too, and taking this pattern into account can improve a game's level designs. To understand how this information can help a designer, it's useful to think of learning curves as having the three different stages shown in Figure 8-3. These stages separate the emotional states that a player often goes through when they face the kind of challenge that a new computer game presents:

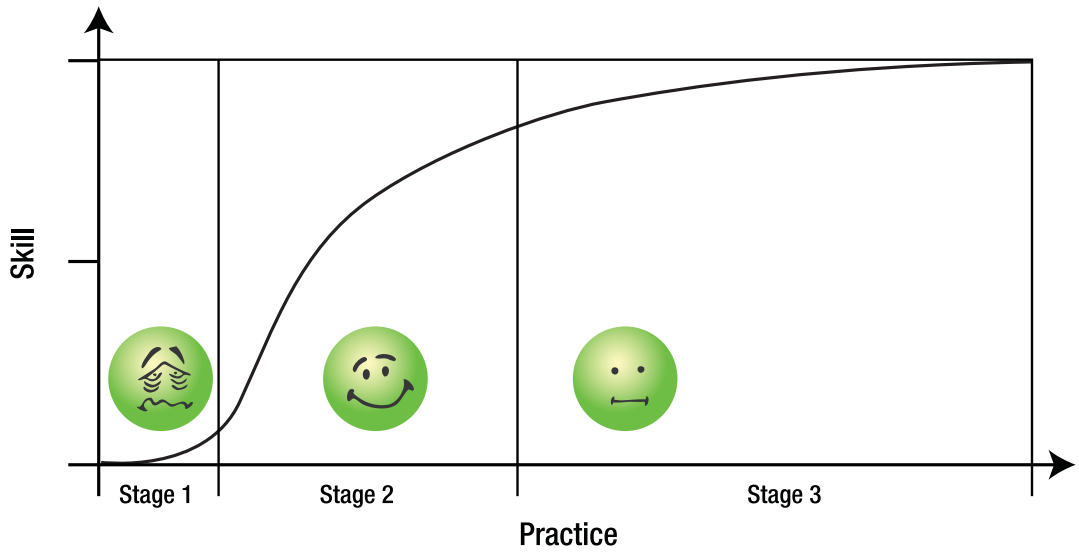

Figure 8-3. *The three stages of a learning curve tell us about the player's likely emotional state.*

Stage One: At the start of a completely new challenge, players have to absorb and process a lot of information at the same time. This means their progress is slow and there is a big risk of frustration as a result. People are often afraid of failure, too, so if the challenge appears too great, they may decide to give up rather than risk feeling humiliated. Therefore, a game design should give the player plenty of support during this period, perhaps by guiding players using a training mode or making them invulnerable until they have grasped the basic elements of the new challenge.

Stage Two: At this stage the player has found their feet and their skill is increasing at a rapid rate. As the player becomes aware of their success, any feelings of frustration will fade away and they'll become happily engrossed in the challenge. This is the ideal state for the player to be in, and it is the designer's job to try and make it last for as long as possible.

Stage Three: This stage represents the player's mastery of the challenge. This doesn't mean they couldn't improve any further, but a flat slope means playing for a long time to make even a little more progress—for example, trying to knock another few seconds off your best lap time after you've already come first in every race. If the goals stop challenging the player or it just seems like too much effort to improve, then it is only a matter of time before players will get bored. A designer must try and make sure that this doesn't happen too early in the game. If players feel they have mastered all the available challenges before they reach the end of a game, there's not much chance that they will bother completing it.

Difficulty Curves

The learning curve provides a good representation of how a player might come to grips with a game like Evil Clutches, where the game's challenges never change. However, games like this have a limited lifespan because players soon get bored when they reach stage three. The aim of good level design is to slowly introduce new challenges and features into the game at a rate that keeps the player in stage two of the learning curve. Introducing new features and challenges also plays a key part in determining the difficulty of a game. Difficulty curves are related to learning curves but represent how the game's difficulty changes over time rather than the player's skill level. If you think of the player's journey through the game as a bit like climbing a mountain slope, then it should look something like the one shown in Figure 8-4.

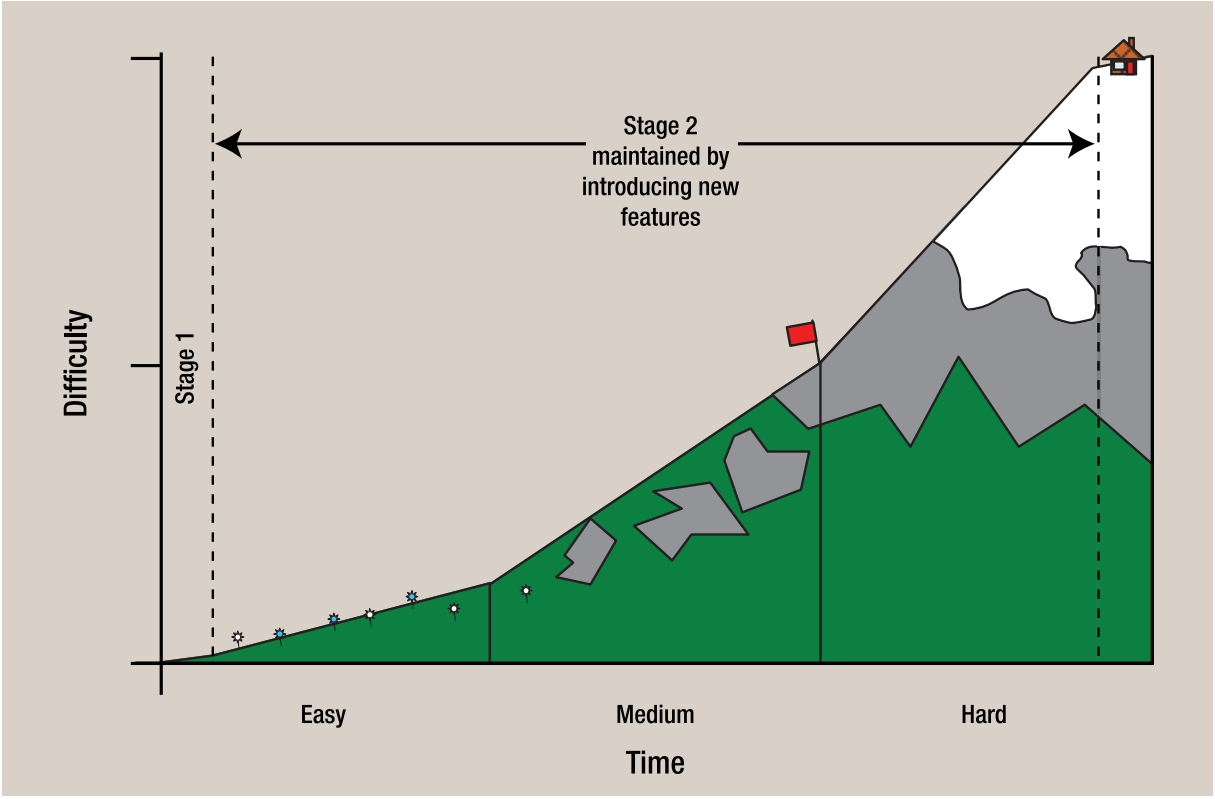

Figure 8-4. *The difficulty curve of a computer game is represented here as a mountain.*

The mountain starts with a shallow slope, in order to support players during the hardest stage of their own learning curves, but gradually increases as the game progresses. As the player's skill grows and they approach the top of their learning curve (see Figure 8-3), the game introduces new, harder challenges to bring them back down the curve and keep them in stage two. In this way, the player's positive emotional state is maintained while the difficulty of the game gradually rises as a result of the extra challenges. By the time the player reaches the steepest section of the slope, they should be well prepared for the game's greatest challenges. At this stage, knowing that they have a lot more distance behind them than in front also helps to drive players on to complete the game.

The slope of a game's difficulty curve has a big effect on the playability of the game. If you make it too steep at the start, then players give up, but if it gets too shallow at any point, players will get bored. Unfortunately, the only way to guarantee that you've got the difficulty curve right is to test and tweak your game with lots of different players. Once you've designed a few games, you'll begin to develop your own ideas and strategies for doing this, but here's one suggested approach that may help you find your way:

1. Decide how many levels you want in your game.
2. Divide this total into three equal groups for EASY, MEDIUM, and HARD levels.
3. Design each level and decide which group it belongs to:
    - All players should be able to complete EASY levels. Design these for players who have never played a game of the same genre before.
    - Most players should be able to complete MEDIUM levels. Design these for casual game-players of this genre.
    - Good players should be able to complete HARD levels. Design these for yourself and your friends who play these kinds of games.
4. If you end up with too many levels in one group, then redesign some of them to make them harder or easier for another group.
5. Play all of the levels again yourself and arrange them in order of difficulty.
6. Test your levels on different players:
    - Friends or family members who dislike games of this genre should be able to complete the EASY levels without help (although bribery may be required!)
    - As the game's designer, you should be able to complete all of the MEDIUM levels without ever having to restart a level.
    - Your friends who like games of this genre should eventually be able to complete all of the HARD levels without any help from you.
7. Tweak or reorder your levels according to the outcome of your tests.

Probably the most important tip here is that a game should only start to challenge its designer about two thirds of the way through the game (where the red flag is on the mountain). We've said it before, but the easiest trap for a designer to fall into is to make your games too hard because you find your own game so easy. And remember—even your final level shouldn't be so hard that you can't complete it yourself!

■ **Note** It's easy to get difficulty and learning curves confused, and people often do. If we want to suggest that something is hard, we will sometimes say it has a "steep learning curve," but as we have seen, a steep learning curve is more likely to suggest that something is easy!

Saving the Day

Giving the player the opportunity to save their progress can also make a big difference to the difficulty of a game. Players these days don't expect to have to go right back to the start of a game every time they fail to complete it, and may quickly lose interest if they do. Save points are considered a natural part of playing longer games, but a designer needs to be wary of the effect that saving can have on the way that a game is played. Providing the player with the ability to save at any point is a gift that many players will not be able to resist abusing—even when it begins to ruin their own playing experience! Saving the game every 30 seconds breaks up the flow of a game and can make its challenges much easier than they were designed to be. Even worse, a designer may be tempted to take it into account and create challenges that require the player to save all the time in order to succeed. This is not a good path to go down and is best avoided by creating automatic save points (such as at the end of each level) or rationing their use, so that the player has to find some kind of token before they can save their progress.

Applying It All

We've covered a lot of theory in this chapter, but we haven't been applying it to our example game as we went along. This had been a deliberately different approach to the first game design chapter because we wanted to emphasize that level design involves planning. Designing as you go along can be a good way to come up with your game's core game mechanics, but if you approach level and feature design in the same way, you're unlikely to end up with a well-structured game. However, now that we have the theory under our belt we can use this to finish off Koalabr8 and turn it into a complete playable game. You'll find a new version of the game containing all the changes we make in this section in `Games/Chapter08/new_koala.gm6` on the CD.

Features

The features we already have for Koalabr8 avoid all the undesirable issues we mentioned at the start of this chapter. There are no equivalent features, one-trick ponies, or arms race issues in any of these, so we already have eight good features to use in our levels. Furthermore, one of the features (boulders) also has the potential for emergent behavior, because it's designed to encourage interactions between features (by allowing the player to destroy them!). This already gives us the following features to structure our level designs around:

- Explosive TNT
- Moving circular saws
- Red exits—Allow any number of koalas to exit the level
- Blue exits—Allow a single koala to exit the level
- Blue locks/switches—Permanently open locked passageways
- Yellow locks/switches—Open locked passageways for a limited time

- Red locks/switches—Open locked passageways only whilst pressed
- Boulders—Can be pushed by koalas and destroy other hazards

We're going to create 18 game levels and two training levels in total, so we already have enough features for a new one nearly every two levels. However, we think there's room for just one more feature, so we'll try to create another that encourages emergence by creating more potential for interactions between features.

Emerging Springs

Earlier in the chapter we mentioned that there were "good" and "bad" kinds of emergence. Unfortunately, the structured nature of puzzle games means that emergence often ends up being the bad kind. If the player discovers how to use a feature in a way the designer didn't expect, it's often more likely to prevent the player from solving the puzzle than helping them to complete it! There's already a good example of this if the player uses a boulder to crush a switch. This interaction has actually been used to create an interesting level, but you can probably see how it could potentially cause problems in the wrong situation.

Fortunately, there is another kind of "good" emergence that can take place between a programmer and level designer. This is where the level designer is able to utilize interactions between features in ways that the programmer hadn't expected. It happens as a result of the same kinds of interactions between features that we discussed before, and can trigger a really constructive evolution of ideas in a game's design. It also has the advantage that any "emergent bugs" that the designer finds can be fixed before the game is finished. With the right kind of relationship between the programmer and level designer, this kind of emergence can have a really positive effect on the gameplay. It's well worth trying to recruit your friends as level designers for this very reason!

We're going to create springs as a new feature that has the potential to create emergence. Springs will send koalas flying in a straight line at twice their normal speed until they hit a solid object. Flying koalas are vulnerable to dangerous obstacles in the same way, but will automatically operate switches and detonators along the way. Traveling at speed immediately creates interplay between other objects that involve speed, like timed switches and circular saws. Making switches operate as koalas fly over them creates more interplay, and we can enhance this a bit by making it possible to turn blue switches back on again after they have been turned off. Therefore, the following are the changes we've made to the features of the game:

- Add springs that turn koalas into flying koalas.
- Make flying koalas travel at double speed.
- Make blue switches alternately switch between on and off.

Training Missions

To help the player through the early stages of their learning curve at the start of the game, we're going to create two training levels. These levels will introduce the player to the main features of the game (TNT, saws, and switches) with a bit of helpful prompting on the screen (see Figure 8-5). However, the main challenge of this game is controlling more than one character

at once, so the training levels will give the player a chance to get used to this, too. They will complete the training level first with just one koala, and then try the same level again with two. In this way, we should be able to make sure that the difficulty curve isn't so steep that it puts players off before they have a chance to get into the game.

Figure 8-5. *The new Koalabr8 training level contains helpful prompts.*

Dividing Levels

Our finished game will have 20 levels, including the two training levels, so that's six in each of the easy, medium, and hard difficulty brackets (18 in total). To help us make sure that the easy levels live up to their name, we'll try to avoid using features that involve timing on these levels (saws and yellow switches). Timing adds extra pressure for the player, so we'll save these until they're a bit more comfortable with the game. Instead, the easy levels will attempt to teach the player how to control more than one koala at once.

The medium levels will try to build on the player's skills by making them control their koalas under pressure. We'll introduce saws and switches and provide situations in which the whole team has to be moved in synchronization in order to avoid obstacles. The hard levels will develop these skills even more and add cryptic elements to the puzzles that involve counterintuitive or inventive interactions between the features.

To give the player a sense of progression, we're also going to make the easy levels slightly smaller than the medium levels and only use the whole screen for the hard levels. We'll also give the player a little extra breathing space when new features are introduced by putting those features in situations where they can be explored without too much danger. So after a lot of designing, testing, and tweaking we eventually came up with the 18 levels that you'll find in the latest version of the game (see Figure 8-6).

Figure 8-6. *A finished Koalabr8 level contains the new spring feature.*

Summary

And that's it! We hope you enjoy playing our new version of Koalabr8 and that it will inspire you to have a go at creating more levels and features for the game. Perhaps you'll even find some emergent solutions to the levels that we hadn't spotted! In this chapter you've learned that there is more to level design that churning out the first ideas that come into your head. We've looked at a number of principles that can help you to design better levels and followed them through with the Koalabr8 example. The main issues for level and feature design were:

- Choose features by
  - Brainstorming with your friends.
  - Being wary of equivalent features.
  - Avoiding starting an arms race.
  - Avoiding one-trick ponies.
  - Encouraging emergence through features that interact.
- Structure your level designs by
  - Including training levels to support the player when they most need it.
  - Carefully controlling the difficulty curve of your game.
  - Tweaking your level designs based on how hard *others* find your game.

However, perhaps the most important thing to remember from this chapter is that the player is *your apprentice* and a good game designer uses their level designs to teach the player how to master their game. That concludes this chapter and the third part of the book. In Part 4, we'll be looking at multiplayer games, and what better way could there be to learn about them than by shooting at your friends with a great big tank?