CHAPTER SEVEN

The Game Improves Through *Iteration*

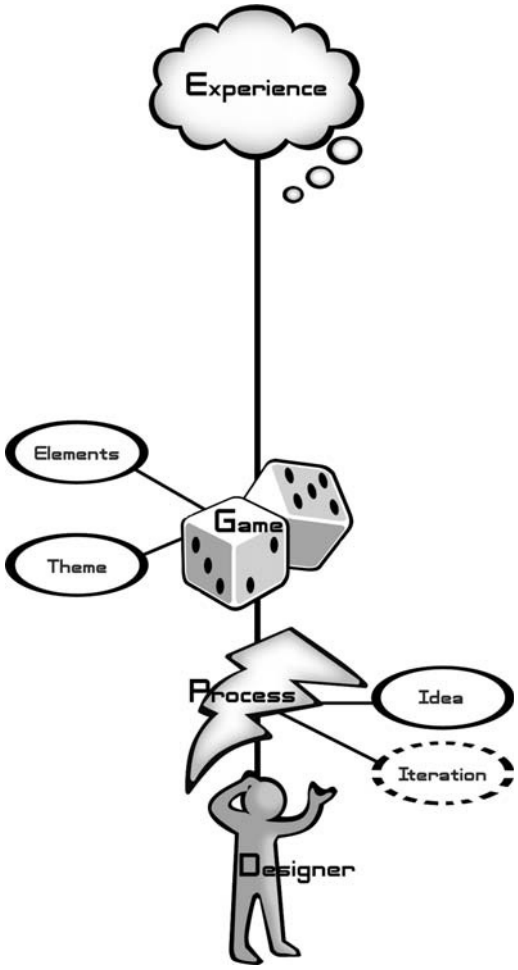


FIGURE **7.1**

Choosing an Idea

After a painfully rapturous brainstorming session, you have a huge list of ideas in front of you. This is where many designers trip up. They have so many ideas they like, they aren't sure what to pick. Or, they have a lot of mediocre ideas, but nothing spectacular, so again they aren't sure what to pick. So they float around for too long, in a vague haze of indecision, hoping that the "right idea" will suddenly become clear, if they just wait a little longer.

But something magic happens when you pick an idea and decide you are going to make it happen. As Steinbeck says in *Of Mice and Men*, "A plan is a real thing." Once you make the internal decision, "Yes, I'm going to do this," flaws you missed before suddenly become evident, as do benefits. It is kind of like flipping a coin to make a decision — when the coin comes down, you suddenly know what you really want. There is something inside us that makes us think about things differently before we've decided to do them than after we've committed. So, take advantage of this quirk of human nature — make snap decisions about your design, commit to sticking with them, and immediately start thinking about the consequences of the choice you have just made.

But what if, with the enlightenment that suddenly comes with commitment, you realize you've made the wrong choice? The answer is easy: be ready to reverse your decision when you realize it is wrong. Many people find this difficult — once they have made a design decision, they are uncomfortable letting it go. You can't afford this kind of sentimentality. Ideas are not like fine china, ideas are like paper cups — they are cheap to manufacture, and when one has holes in it, go get another one.

Some people are quite disconcerted by this combination of snap decisions combined with sudden reversals. But it is the most efficient way to make full use of your decision-making power, and game design is all about making decisions — you need to make the best decisions possible, as fast as possible, and this slightly eccentric behavior is the way to do it. It's always better to commit to an idea sooner, rather than later — you will get to a good decision much faster than if you bide your time considering potential alternatives. Just don't fall in love with your decision, and be ready to reverse it the moment it isn't working for you.

So, how do you pick? In one sense, the answer is "Best guess, Mr. Sulu." More analytically, there are many factors you will need to consider as you start developing a seed idea. It can pay to keep in mind what your idea must grow into before you even choose a seed.

The Eight Filters

Your finished design will eventually have to make it through eight tests, or filters. Only when it passes all of them is your design "good enough." Whenever it fails one of these tests, you will have to change the design, and then run it through all eight tests, or "filters," again, because a change that makes it past one filter might

make it fail another one. In a sense, the design process mainly consists of stating your problem, getting an initial idea, and finding a way to get it past all eight filters.

The eight filters are

Filter #1: Artistic Impulse: This is the most personal of the filters. You, as the designer, basically ask yourself whether the game "feels right" to you, and if it does, it passes the test. If it doesn't, something needs to change. Your gut feelings are important. They won't always be right, but the other filters will balance that out.
Key Question*: "Does this game feel right?"*

Filter #2: Demographics: Your game is likely to have an intended audience. This might be an age bracket, or a gender, or some other distinct audience (e.g., "golf enthusiasts"). You have to consider whether your design is right for the demographic you are targeting. Demographics will be discussed in more detail in Chapter 8.
Key Question*: "Will the intended audience like this game enough?"*

Filter #3: Experience Design: To apply this filter, take into account everything you know about creating a good experience, including aesthetics, interest curves, resonant theme, game balancing, and many more. Many of the lenses in this book are about experience design — to pass this filter, your game must stand up to the scrutiny of many lenses.
Key Question*: "Is this a well-designed game?"*

Filter #4: Innovation: If you are designing a new game, by definition there needs to be something new about it, something players haven't seen before. Whether your game is novel enough is a subjective question, but a very important one.
Key Question*: "Is this game novel enough?"*

Filter #5: Business and Marketing: The games business is a business, and designers who want their games to sell must consider the realities of this and integrate them into their game's design. This involves many questions. Are the theme and story going to be appealing to consumers? Is the game so easily explainable that one can understand what it is about just by looking at the box? What are the expectations consumers are going to have about this game based on the genre? How do the features of this game compare to other similar games in the marketplace? Will the cost of producing this game be so high as to make it unprofitable? Will retailers be willing to sell this game? The answers to these and many other questions are going to have an impact on your design. Ironically, the innovative idea that drove the initial design may prove to be completely untenable when viewed through this filter. This will be discussed in detail in Chapter 29.
Key Question*: "Will this game sell?"*

Filter #6: Engineering: Until you have built it, a game idea is just an idea, and ideas are not necessarily bound by the constraints of what is possible or practical. To pass this filter, you have to answer the question "How are we going to build this?" The answer may be that the limits of technology do not permit the idea as originally envisioned to be constructed. Novice designers often grow frustrated with the limits that engineering imposes on their designs. However, the engineering filter can just as often grow a game in new

directions, because in the process of applying this filter, you may realize that engineering makes possible features for your game that did not initially occur to you. The ideas that appear during the application of this filter can be particularly valuable, since you can be certain that they are practical. More issues of engineering and technology will be discussed in Chapter 26.
Key Question*: "Is it technically possible to build this game?"*

Filter #7: Social/Community: Sometimes, it is not enough for a game to be fun. Some of the design goals may require a strong social component, or the formation of a thriving community around your game. The design of your game will have a strong impact on these things. This will be discussed in detail in Chapters 21 and 22.
Key Question*: "Does this game meet our social and community goals?"*

Filter #8: Playtesting: Once the game has been developed to the point that it is playable, you must apply the playtesting filter, which is arguably the most important of all the filters. It is one thing to imagine what playing a game will be like, and quite another to actually play it, and yet another to see it played by your target audience. You will want to get your game to a playable stage as soon as possible, because when you actually see your game in action, important changes that must be made will become obvious. In addition to modifying the game itself, the application of this filter often changes and tunes the other filters as you start to learn more about your game mechanics and the psychology of your intended audience. Playtesting will be discussed in detail in Chapter 25.
Key Question*: "Do the playtesters enjoy the game enough?"*

Sometimes, in the course of design, you may find a need to change one of the filters — perhaps originally you targeted one demographic (say, males ages 18–35), but while designing, you stumbled into something that better fits another demographic (say, females over 50). Changing the filters is fine, when your design constraints will allow it. The important thing is that somehow, by changing the filters or by changing your design, you find a way to get through all eight.

You will be using these filters continuously throughout the rest of the design and development process of your game. When picking an initial idea, it makes sense to evaluate which of your ideas is going to have the best shot of being molded and shaped to the point it can survive this gauntlet. The perspective of the eight filters is a very useful way to evaluate your game, so let's make it Lens #13.

Lens #13: The Lens of the Eight Filters

To use this lens, you must consider the many constraints your design must satisfy. You can only call your design finished when it can pass through all eight filters without requiring a change.

Ask yourself the eight key questions:

- Does this game feel right?
- Will the intended audience like this game enough?
- Is this a well-designed game?
- Is this game novel enough?
- Will this game sell?
- Is it technically possible to build this game?
- Does this game meet our social and community goals?
- Do the playtesters enjoy this game enough?

In some situations, there may be still more filters; for example, an educational game will also have to answer questions like "Does this game teach what it is supposed to?" If your design requires more filters, don't neglect them.

The Rule of the Loop

It is somewhat daunting to consider that all of Chapter 6 and the first part of this one have merely been an elaboration of "1. Think of an idea." On the other hand, ideas are at the root of design, and their production is so mysterious as to be almost magical, so perhaps it shouldn't surprise us that there is so much to say about this single step.

At this point in the process, you have thought of many ideas, and chosen one, and now it is time to move on to the next step: "2. Try it out." And many designers and developers do just that — leap in and try out their game. And if your game is simple — such as a card game, board game, or very simple computer game — and you have plenty of time to test it and change it, over and over, until it is great, you probably should do just that.

But what if you can't just build a working prototype of your game in an hour or two? What if your game vision requires months of artwork and programming before you will even be able to try it out? If this is the case (as it is for many modern videogame designs), you need to proceed cautiously at this point. The process of game design and development is necessarily iterative, or looping. It is impossible to accurately plan how many loops it is really going to take before your game passes all eight filters and is "good enough." This is what makes game development so incredibly risky — you are gambling that you will be able to get your game to pass all eight filters on a fixed budget, when you really don't know if it will.

The naïve strategy, that many still use today, is to start slapping the game together and hope for the best. Sometimes this works. But when it doesn't, you are in a horrible mess. You either have to ship a game that you know isn't good enough, or suffer the expense of continuing development until it is. And often, this extra time and expense is enough to make the project completely unprofitable.

In truth, this is a problem for all software projects. Software projects are so complex that it is very difficult to predict how long they will take to build, and how long it will take to find and fix all of the bugs that will surely appear during development. On top of all that, games have the added burden of needing to be fun — game developers have a couple of extra filters that non-game software developers don't need to worry about.

The real problem here is the Rule of the Loop.

The Rule of the Loop: The more times you test and improve your design, the better your game will be.

The Rule of the Loop is not a lens, because it is not a perspective — it is an absolute truth. There are no exceptions to the Rule of the Loop. You will try, at times in your career, to rationalize it away, to convince yourself that "this time, the design is so good, we don't have to test and improve," or "we really have no choice — we'll have to hope for the best," and you will suffer for it each time. The horrible thing about computer games is that the amount of time and money it takes to test and adjust the system is so much greater than for traditional games. It means computer game developers have no choice but to loop fewer times, which is a terribly risky thing to do.

If you are indeed embarking on the design of a game that is likely to involve long "test and improve" loops, you need to answer these two questions:

- Loop Question 1: How can I make every loop count?
- Loop Question 2: How can I loop as fast as possible?

The software engineering people have thought about this problem a lot over the last forty years, and they have come up with some useful techniques.

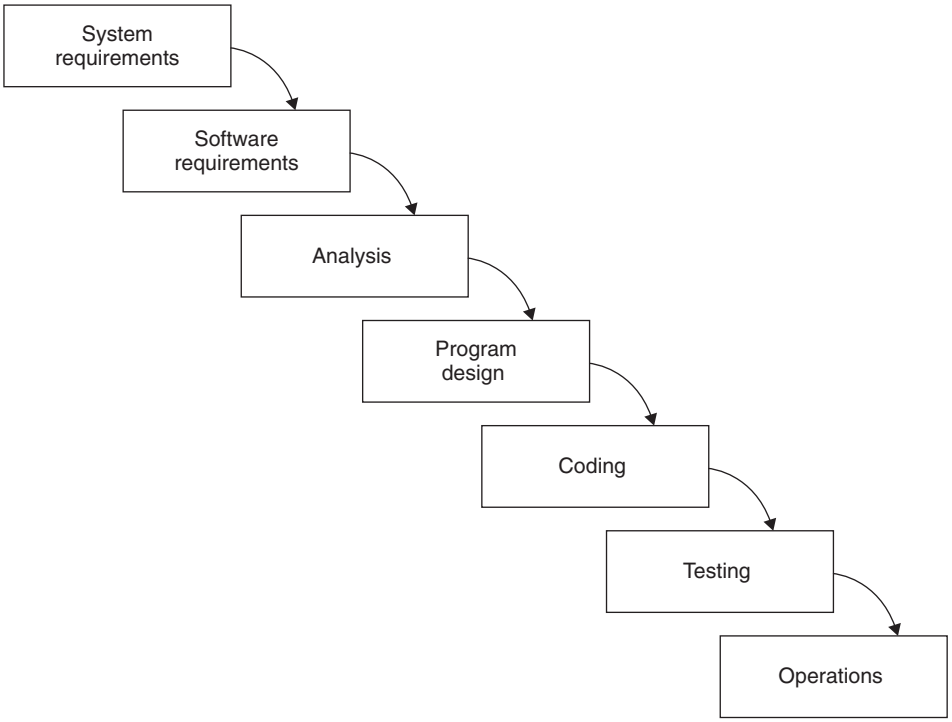
A Short History of Software Engineering

Danger — Waterfall — Keep Back

In the 1960s, when software development was still relatively new, there was very little in the way of formal process. Programmers just made their best guesses about how long things would take, and they would start coding. Often the guesses

were wrong, and many software projects went disastrously over budget. In the 1970s, in an attempt to bring some order to this unpredictable process, many developers (usually at the behest of non-technical management) tried to adopt the "waterfall model" of software development, which was an orderly seven-step process for software development. It was generally presented looking something like this:

FIGURE **7.2**



And it certainly looks appealing! Seven orderly steps, and when each is complete, nothing remains but to move on to the next one — the very name "waterfall" implies that no iteration is needed, since waterfalls generally do not flow uphill.

The waterfall model had one good quality: it encouraged developers to spend more time in planning and design before just jumping into the code. Except for that, it is complete nonsense, because it violates the Rule of the Loop. Managers found it incredibly appealing, but programmers knew it to be absurd — software is simply too complex for such a linear process to ever work. Even Winston Royce, who wrote

the paper which was the foundation for all of this, disagreed with the waterfall model as it is commonly understood. Interestingly, his original paper emphasizes the importance of iteration and the ability to go back to previous steps as needed. He never even used the word "waterfall"! But what was taught at universities and corporations everywhere was this linear approach. The whole thing seems to have been wishful thinking, mostly promulgated by people who did not actually have to build real systems themselves.

Barry Boehm Loves You

Then, in 1986, Barry Boehm (pronounced "beam") presented a different model, which was based more closely on how real software development actually happens. It is usually presented as a somewhat intimidating diagram, where development starts in the middle, and spirals out clockwise, passing through four quadrants again and again (Figure 7.3).

His model has a lot of complex detail, but we don't need to go into all of that. There are basically three great ideas wrapped up in here: risk assessment, prototypes, and looping. In brief, the spiral model suggests that you:

1. Come up with a basic design.
2. Figure out the greatest risks in your design.
3. Build prototypes that mitigate those risks.
4. Test the prototypes.
5. Come up with a more detailed design based on what you have learned.
6. Return to step 2.

And basically, you repeat this loop until the system is done. This beats the waterfall model hands down, because it is all about the Rule of the Loop. Also, it answers the questions we stated earlier:

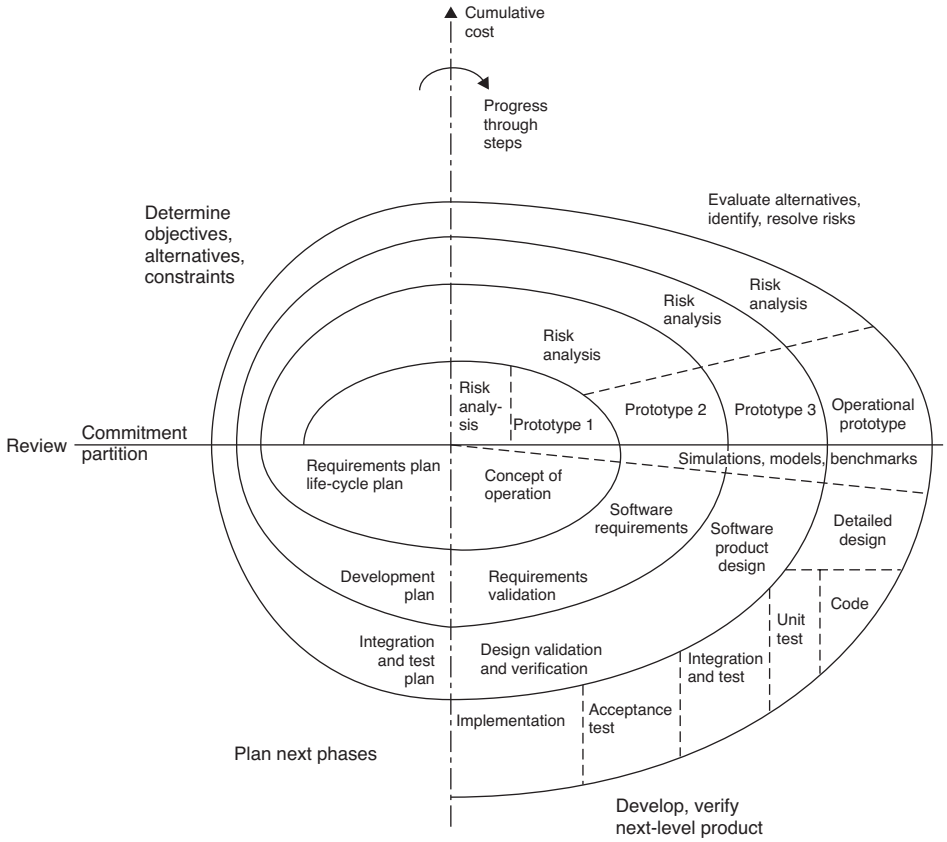
- **Loop Question 1**: How can I make every loop count?

  Spiral Model Answer: Assess your risks and mitigate them.
- **Loop Question 2**: How can I loop as fast as possible?

  Spiral Model Answer: Build many rough prototypes.

There have been many descendants of the spiral model, which you may want to investigate. Although these have their differences, they all feature risk assessment and prototyping at their core.

FIGURE
7.3



The spiral model of software development

Risk Assessment and Prototyping

Example: Prisoners of Bubbleville

Let's say you and your team have decided you want to make a videogame all about parachuting into a city. You have a brief design description that you based on the elemental tetrad:

Prisoners of Bubbleville — Design Brief

Story: You are "Smiley," a parachuting cat. The good people of Bubbleville are trapped in their houses by an evil wizard. You must find a way to defeat the wizard, by repeatedly parachuting into the city and sliding down chimneys to visit the citizens and get clues about how to stop the wizard.

Mechanics: As you parachute toward the city, you are trying to grab magic bubbles that rise up from the city and use their energy to shoot rays at evil vultures that

try to pop the bubbles and rip your parachute. Simultaneously, you must navigate down to one of several target buildings in the city.

Aesthetics: A cartoony look and feel.

Technology: Multiple platform 3D console game using a third-party engine.

One approach you could take would be to just start building the game. Start writing code, designing detailed levels, animating the characters, while you wait for it all to come together, to see what it will really be like. But this could be incredibly dangerous. Assuming this is an eighteen-month project, it might take as long as six months before you even have anything you can playtest. What if you learned, at that point, that your game idea wasn't fun? Or your game engine wasn't up to the job? You would be in real trouble. You would be one-third of the way through the project and would only have completed a single loop!

Instead, the right thing is to sit down with your team, and do a risk analysis. This means making a list of all the things that might jeopardize the project. A sample list for this game might be:

Prisoners of Bubbleville — Risk List

Risk #1: The bubble collecting/vulture shooting mechanic might not be as fun as we think.

Risk #2: The game engine might not be able to handle drawing an entire city and all those bubbles and vultures at once.

Risk #3: Our current thinking is that we need thirty different houses to make a full game — creating all the different interiors and animated characters might take more time than we have.

Risk #4: We aren't sure people will like our characters and story.

Risk #5: There is a chance the publisher might insist we theme this game to a new summer movie about stunt parachuting.

In reality, you will probably have many more risks, but for the sake of our example, we'll just consider these. So, what do you do about these risks? You could just cross your fingers and hope these things don't happen, or you could do the smart thing: risk mitigation. The idea is to reduce or eliminate the risks as soon as possible, often by building small prototypes. Let's look at how each of these risks could be mitigated:

Prisoners of Bubbleville — Risk Mitigation

Risk #1: The bubble collecting/vulture shooting mechanic might not be as fun as we think.

Game mechanics can often be abstracted and played in a simpler form. Have a programmer make a very abstract version of this gameplay mechanic, perhaps in 2D, with simple geometric shapes instead of animated characters. You can probably have a working game in a week or two, and start answering questions about whether it is fun right away. If it isn't, you can make quick modifications to the simple prototype, until it is fun, and then begin work on the elaborate 3D version. You'll be doing more loops sooner, wisely taking advantage of the Rule of the Loop. You might object to this approach, thinking that throwing out the 2D prototyping code, which the players will never see, is wasteful. In the long run, though, you will

have saved time, because you will be coding the right game sooner, and not endlessly coding and recoding the wrong game.

Risk #2: The game engine might not be able to handle drawing an entire city and all those bubbles and vultures at once.

If you wait for all the final artwork to answer this question, you could put yourself in a horrible situation: If the game engine can't handle it, you now have to ask the artists to redo their work so it is less strain on the game engine, or ask the programmers to spend extra time trying to find tricks to render everything more efficiently (or most likely, both of these things). To mitigate the risk, build a quick prototype, right away, that does nothing but show the approximate number of equivalent items on screen, to see if the engine can handle it. This prototype has no gameplay; it is purely to test technical limits. If it can handle it, great! If it can't, you can figure out a solution now, before any art has been generated. Again, this prototype will be a throwaway.

Risk #3: Our current thinking is that we need thirty different houses to make a full game — creating all the different interiors and animated characters might take more time than we have.

If you get halfway into development before you realize that you don't have the resources to build all the artwork, you are doomed. Have an artist create one house and one animated character immediately to see how long it takes, and if it takes longer than you can afford, change your design immediately — maybe you could have fewer houses, or maybe you could reuse some the interiors and characters.

Risk #4: We aren't sure people will like our characters and story.

If you really are concerned about this, you cannot wait until the characters and story are in the game to find out. What kind of prototype do we build here? An art prototype — it might not even be on a computer — just a bulletin board. Have your artists draw some concept art, or produce test renders of your characters and settings. Create some storyboards that show how the story progresses. Once you have these, start showing them to people (hopefully people in your target demographic) and gauge their reactions. Figure out what they like, don't like, and why. Maybe they like the look of the main character, but hate his attitude. Maybe the villain is exciting, but the story is boring. You can figure most of this out completely independent of the game. Each time you do this, and make a change, you've completed another loop and gotten one step closer to making a good game.

Risk #5: There is a chance we might have to theme this game to a new summer movie about stunt parachuting.

This risk might sound absurd, but this kind of thing happens all the time. When it happens in the middle of a project, it can be horrible. And you can't ignore this kind of thing — you must seriously consider every risk that might threaten your project. Will a prototype help in this case? Probably not. To mitigate this risk, you can lean on management to get a decision as fast as possible, or you could decide to make a game that could more easily be re-themed to the movie. You might even come up with a plan for making two different games — the key idea is that you consider the risk immediately and take action now to make sure it doesn't endanger your game.

Risk assessment and mitigation is such a useful perspective to take, it becomes Lens #14.

Lens #14: The Lens of Risk Mitigation

To use this lens, stop thinking positively, and start seriously considering the things that could go horribly wrong with your game.

Ask yourself these questions:

- What could keep this game from being great?
- How can we stop that from happening?

Risk management is hard. It means you have to face up to the problems you would most like to avoid, and solve them immediately. But if you discipline yourself to do it, you'll loop more times, and more usefully, and get a better game as a result. It is tempting to ignore potential problems and just work on the parts of your game you feel most confident about. You must resist this temptation and focus on the parts of your game that are in danger.

Eight Tips for Productive Prototyping

It is widely understood that rapid prototyping is crucial for quality game development. Here are some tips that will help you build the best, most useful prototypes for your game.

Prototyping Tip #1: Answer a Question

Every prototype should be designed to answer a question and sometimes more than one. You should be able to state the questions clearly. If you can't, your prototype is in real danger of becoming a time-wasting boondoggle, instead of the time-saving experiment it is supposed to be. Some sample questions a prototype might answer:

- How many animated characters can our technology support in a scene?
- Is our core gameplay fun? Does it stay fun for a long time?
- Do our characters and settings fit together well aesthetically?
- How large does a level of this game need to be?

Resist the temptation to overbuild your prototype, and focus only on making it answer the key question.

Prototyping Tip #2: Forget Quality

Game developers of every stripe have one thing in common: they are proud of their craft. Naturally, then, many find the idea of doing a "quick and dirty" prototype completely abhorrent. Artists will spend too much time on early concept sketches — programmers will spend too much time on good software engineering for a piece of throwaway code. When working on a prototype all that matters is whether it answers the question. The faster it can do that, the better — even if it just barely works and looks rough around the edges. In fact, polishing your prototype may even make things worse. Playtesters (and colleagues) are more likely to point out problems with something that looks rough than with something that looks polished. Since your goal is to find problems immediately so you can solve them early, a polished prototype can actually defeat your purpose by hiding real problems, thus lulling you into a false sense of security.

There is no getting around the Rule of the Loop. The faster you build the prototype that answers your question, the better, despite how ugly it may look.

Prototyping Tip #3: Don't Get Attached

In *The Mythical Man Month*, Fred Brooks made the famous statement "Plan to throw one away — you will anyway." By this he means that whether you like it or not, the first version of your system is not going to be a finished product, but really a prototype that you will need to discard before you build the system the "right" way. But in truth, you may throw away many prototypes. Less experienced developers often have a hard time doing this — it makes them feel like they have failed. You need to enter the prototyping work with the mindset that it is all temporary — all that matters is answering the question. Look at each prototype as a learning opportunity — as practice for when you build the "real" system. Of course, you won't throw out everything — you'll keep little pieces here and there that really work and you'll combine them to make something greater. This can be painful. As designer Nicole Epps once put it, "You must learn how to cut up your babies."

Prototyping Tip #4: Prioritize Your Prototypes

When you make your list of risks, you might realize that you need several prototypes to mitigate all the risks that you face. The right thing to do is to prioritize them, so that you face the biggest risks first. You should also consider dependence — if the results of one prototype have the potential to make the other prototypes meaningless, the "upstream" prototype is definitely your highest priority.

Prototyping Tip #5: Parallelize Prototypes Productively

One great way to get more loops in is to do more than one at a time. While the system engineers work on prototypes to answer technology questions, the artists can work on art prototypes, and the game scripters can work on gameplay prototypes. Having lots of small, independent prototypes can help you answer more questions faster.

Prototyping Tip #6: It Doesn't Have to be Digital

Your goal is to loop as usefully and as frequently as possible. So, if you can manage it, why not just get the software out of the way? If you are clever, you can prototype your fancy videogame idea as a simple board game, or what we sometimes call a **paper prototype**. Why do this? Because you can make board games *fast,* and often capture the same gameplay. This lets you spot problems sooner — much of the process of prototyping is about looking for problems, and figuring out how to fix them, so paper prototyping can be a real time saver. If your game is turn-based to start with, this becomes easy. The turn-based combat system for Toontown Online was prototyped through a simple board game, which let us carefully balance the many types of attacks and combos. We would keep track of hit points on paper or on a whiteboard, and play again and again, adding and subtracting rules until the game seemed balanced enough to try coding up.

Even real-time games can be played as paper prototypes. Sometimes they can be converted to a turn-based mode that still manages to capture the gameplay. Other times, you can just play them in real-time, or nearly. The best way to do it is to have other people help you. We'll consider two examples.

Tetris: A Paper Prototype

Let's say you wanted to make a paper prototype of *Tetris*. You could cut out little cardboard pieces, and put them in a pile. Get someone else to draw them at random, and start sliding them down the "board" (a sketch you've drawn on a piece of paper), while you grab them, and try to rotate them into place. To complete a line, you have to just use your imagination, or pause the game while you cut the pieces with an X-acto knife. This would not be the perfect Tetris experience, but it might be close enough for you to start to see if you had the right kinds of shapes, and also enough to give you some sense of how fast the pieces should drop. And you could get the whole thing going in about 15 minutes.

Doom: A Paper Prototype

Would it be possible to make a paper prototype of a first person shooter? Sure! You need different people to play the different AI characters as well as different players. Draw out the map on a big piece of graph paper, and get little game pieces to represent the different players and monsters. You need one person to control each

of the players and one for each of the monsters. You could then either make some turn-based rules about how to move and shoot, or get yourself a metronome! It is easy to find free metronome software online. Configure your metronome to tick once every five seconds, and make a rule that you can move one square of graph paper with every tick. When there is a line of sight, you can take a shot at another player or monster, but only one shot per tick. This will give the feeling of playing the whole thing in slow motion, but that can be a good thing, because it gives you time to think about what is working and not working while you are playing the game. You can get a great sense of how big your map should be, the shapes of hallways and rooms that make for an interesting game, the properties your weapons should have, and many other things — and you can do it all lightning fast!

Prototyping Tip #7: Pick a "Fast Loop" Game Engine

The traditional method of software development is kind of like baking bread:

1. Write code
2. Compile and link
3. Run your game
4. Navigate through your game to the part you want to test
5. Test it out
6. Go back to step 1

If you don't like the bread (your test results), there is no choice but to start the whole process over again. It takes way too long, especially for a large game. By choosing an engine with the right kind of scripting system, you can make changes to your code while the game is running. This makes things more like working with clay — you can change them continuously:

1. Run your game
2. Navigate through your game to the part you want to test
3. Test it out
4. Write code
5. Go back to step 3

By recoding your system while it is running, you can get in more loops per day, and the quality of your game goes up commensurately. I have used Scheme, Smalltalk, and Python for this in the past (I'm a big fan of Panda3D: www.panda3d.com), but any late-binding language will do the job. If you are afraid that these kinds of languages run too slowly, remember that it is okay to write your games with more

than one kind of code: write the low-level stuff that doesn't need to change much in something fast but static (Assembly, C++, etc.), and write the high-level stuff in something slower but dynamic. This may take some technical work to pull off, but it is worth it because it lets you take advantage of the Rule of the Loop.

Prototyping Tip #8: Build the Toy First

Back in Chapter 3, we distinguished between toys and games. Toys are fun to play with for their own sake. In contrast, games have goals and are a much richer experience based around problem solving. We should never forget, though, that many games are built on top of toys. A ball is a toy, but baseball is a game. A little avatar that runs and jumps is a toy, but Donkey Kong is a game. You should make sure that your toy is fun to play with before you design a game around it. You might find that once you actually build your toy, you are surprised by what makes it fun, and whole new ideas for games might become apparent to you.

Game designer David Jones says that when designing the game *Lemmings*, his team followed exactly this method. They thought it would be fun to make a little world with lots of little creatures walking around doing different things. They weren't sure what the game would be, but the world sounded fun, so they built it. Once they could actually play with the "toy," they started talking seriously about what kinds of games could be built around it. Jones tells a similar story about the development of *Grand Theft Auto*: "Grand Theft Auto was not designed as Grand Theft Auto. It was designed as a medium. It was designed to be a living, breathing city that was fun to play." Once the "medium" was developed, and the team could see that it was a fun toy, they had to decide what game to build with it. They realized the city was like a maze, so they borrowed maze game mechanics from something they knew was good. Jones explains: "GTA came from Pac-Man. The dots are the little people. There's me in my little, yellow car. And the ghosts are policemen."

By building the toy first, and then coming up with the game, you can radically increase the quality of your game, because it will be fun on two levels. Further, if the gameplay you create is based on the parts of the toy that are the most fun, the two levels will be supporting each other in the strongest way possible. Game designers often forget to consider the toy perspective. To help us remember, we'll make it Lens #15.

Lens #15: The Lens of the Toy

To use this lens, stop thinking about whether your game is fun to play, and start thinking about whether it is fun to play *with*.

Ask yourself these questions:

- If my game had no goal, would it be fun at all? If not, how can I change that?
- When people see my game, do they want to start interacting with it, even before they know what to do? If not, how can I change that?

There are two ways to use the Lens of the Toy. One way is to use it on an existing game, to figure out how to add more toy-like qualities to it — that is, how to make it more approachable, and more fun to manipulate. But the second way, the braver way, is to use it to invent and create new toys before you even have any idea what games will be played with them. This is risky if you are on a schedule — but if you are not, it can be a great "divining rod" to help you find wonderful games you might not have discovered otherwise.

Closing the Loop

Once you have built your prototypes, all that remains is to test them, and then based on what you have learned, start the whole process over again. Recall the informal process we discussed earlier:

The Informal Loop:

1. Think of an idea.
2. Try it out.
3. Keep changing it and testing it until it seems good enough.

Which we have now made a bit more formal:

The Formal Loop:

1. State the problem.
2. Brainstorm some possible solutions.
3. Choose a solution.
4. List the risks of using that solution.
5. Build prototypes to mitigate the risks.
6. Test the prototypes. If they are good enough, stop.
7. State the new problems you are trying to solve, and go to step 2.

With each round of prototyping, you will find yourself stating the problems in more detail. To give an example, let's say you are given the task of creating a racing

game — but there has to be something new and interesting about it. Here is a summary of how a few loops of that process might play out.

Loop 1: "New Racing Game"

- Problem Statement: Come up with a new kind of racing game
- Solution: Underwater submarine races (with torpedoes!)
- Risks:
  - Not sure what underwater racetracks should look like
  - This might not feel innovative enough
  - Technology might not be able to handle all the water effects
- Prototypes:
  - Artists working on concept sketches of underwater racetracks
  - Designers prototyping (using paper prototypes and by hacking an existing racecar game) novel new effects (subs that can also rise out of water and fly, tracking missiles, depth charges, racing through a minefield)
  - Programmers testing out simple water effects
- Results:
  - Underwater racetracks look okay if there is a "glowing path" in the water. Underwater tunnels will be cool! So will flying submarines following tracks that go in and out of the water!
  - Early prototypes seem fun, provided the submarines are very fast and maneuverable. It will be necessary to make them be "racing subs." The mix of flying and swimming feels very novel. Subs should go faster when flying, so we will need to find a way to limit the amount of time they can spend in the air. The little playtesting we have done makes it clear this game must support networked multiplay.
  - Some water effects are easier than others. Splashes look good, so do underwater bubbles. Making the whole screen waver takes too much CPU, and is kind of distracting anyway.

Loop 2: "Racing Subs" Game

- New Problem Statement: Design a "racing sub" game, where subs can fly.
- Detailed problem statements:
  - Not sure what "racing subs" look like. We need to define the look of both subs and racetracks.

  - Need to find a way to balance the game, so that subs spend the right amount of time in and out of the water.
  - Need to figure how to support networked multiplay.
- Risks:
  - If the racing subs look "too cartoony" they might turn off older players. If they look too realistic, they might just seem silly with this kind of gameplay.
  - Until we know how much time we are spending in and out of the water, it is impossible to design levels, or to do the artwork for the landscapes.
  - The team has never done networked multiplay for a racing game. We aren't completely sure we can do it.
- Prototypes:
  - Artists will sketch different kinds of subs, in a number of different styles: cartoony, realistic, hyper-realistic, subs that are living creatures. The team will vote on them, and we will also informally survey members of our target audience.
  - Programmers and designers will work together on a very crude prototype that lets them experiment with how much time should be spent in and out of the water, and different mechanics for managing that.
  - Programmers will build a rough framework for networked multiplay that should handle all the kinds of messages this kind of game will need.
- Results:
  - Everyone loves the "dino-sub" designs. There is strong agreement between team members and potential audience members that "swimming dinosaurs" are the right look and feel for this game.
  - After several experiments, it becomes clear that for most levels, 60% of time should be spent underwater, 20% in the air, and 20% near the surface, where players who grab the right powerups can fly above the water for a speed advantage.
  - The early networked experiments show that mostly the racing is not a problem for multiplay, but if we can avoid using rapid-fire machine guns, multiplay will be a lot easier.

Loop 3: "Flying Dinos" Game

- Problem Statement: Design a "flying dinos" game where dinosaurs race in and above the water.
- Detailed Problem Statements:
  - We need to figure out if we can schedule all the animation time needed for the dinosaurs.

- We need to develop the "right" number of levels for this game.
- We need to figure out all the powerups that will go into this game.
- We need to determine all the weapons that this game should support (and avoid rapid-fire machine guns because of networking constraints).

Notice how the problem statements gradually evolved and got more specific with each loop. Also notice how ugly problems bubbled to the surface quickly: What if the team hadn't tried out all the different character designs so early? What if three levels of the game had already been designed and modeled before anyone noticed the problem of keeping players in the air for the right amount of time? What if the machine gun system had already been coded up, and the whole gameplay mechanic centered around it, before anyone realized it would break the networking code? These problems got addressed quickly because of so many early loops. It looks like just two complete loops, and the beginning of a third one, but because of the wise use of parallelism, there were really six design loops.

Also notice how the whole team was involved in important design decisions. There is no way that a lone designer could have done this — much of the design was informed by the technology and the aesthetics.

How Much is Enough?

You might wonder how many loops are needed before the game is done. This is a very hard question to answer, and it is what makes game development so difficult to schedule. The Rule of the Loop implies that one more loop will always make your game a little better. So, as the saying goes, the work is never finished — only abandoned. The important thing is to make sure you get enough loops in to produce a game you are proud of before you've used up the entire development budget.

So, when you stand there at the beginning of the first loop, is it possible to make an accurate estimate of when you will have a finished, high-quality game? No. It is simply not possible. Experienced designers, after a time, get better at guessing, but the large number of game titles that ship later than originally promised, or with lower quality than originally promised, is testament to the fact that there is just no way to know. Why is this? Because at the beginning of the first loop, you don't yet know what you are going to build! With each loop, though, you get a more solid idea of what the game will really be, and this allows for more accurate estimates.

Game designer Mark Cerny has described a system for game design and development that he calls "The Method." Not surprisingly, this features a system of iteration and risk mitigation. But The Method makes an interesting distinction between what Cerny calls "pre-production" and "production" (terms borrowed from Hollywood). He argues that you are in pre-production until you have finished two publishable levels of your game, complete with all necessary features. In other words, until you have two completely finished levels, you are still figuring out the fundamental

design of your game. Once you reach this magic point, you are now in production. This means that you know enough about what your game really is that you can safely schedule the rest of development. Cerny states that usually this point is generally reached when 30% of the necessary budget has been spent. So, if it costs you $1 million to get to this point, it will probably cost you another $2.3 million to actually complete the game. This is a great rule of thumb, and realistically, this might be the most accurate way to really plan the release date for a game. The problem with it is that you won't really know what the game will cost or when it will be complete until you have already spent 30% of what it will take to get there. In truth, this problem is unavoidable — The Method just guides you toward reaching a point of predictability as soon as is realistically possible.

The principles of iteration described here might sound special to game design, but they are not. Gradual, evolutionary development is the key to any kind of design.

Now that we have discussed how games should be made, let's consider who we are making them for.