Lean Game Development

Apply Lean Frameworks to the Process of Game Development

Julia Naomi Rosenfield Boeira

Apress®

Lean Game Development

Apply Lean Frameworks to the Process of Game Development

Julia Naomi Rosenfield Boeira

Apress®

Lean Game Development: Apply Lean Frameworks to the Process of Game Development

Julia Naomi Rosenfield Boeira
Porto Alegre, Rio Grande do Sul, Brazil

ISBN-13 (pbk): 978-1-4842-3215-6 ISBN-13 (electronic): 978-1-4842-3216-3
https://doi.org/10.1007/978-1-4842-3216-3

Library of Congress Control Number: 2017960765

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen
Copy Editor: Kim Wimpsett
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484232156. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Julia Naomi Rosenfield Boeira is a software developer at ThoughtWorks and leads VR and AR innovation at its Porto Alegre office in Brazil. She has worked on a series of games and has experience with the Unity, Unreal, Cry, and Panda3D engines, as well as a bunch of other gaming frameworks. She has published articles about AI development and lean game development and is a frequent committer to GitHub.

Acknowledgments

First, I would like to thank my family for always being by my side supporting me. Thanks to Diego and Kinjo, and thanks to the little happy cheerful ones: Fluffy, Ffonxio, Dodonxio, Maluconxio, and Highlander. Furthermore, I must thank Ada Lovelace and Alan Turing for being such great minds at the beginning of computer science.

I also must thank my dad for buying my first C++ book so I would shut up and for all the text revisions that he has made in my life. Thanks to my mother for all the incentives in education. Also, thanks to Microsoft, Nintendo, Sony, and Sega for giving me hundreds of thousands of hours of video game entertainment.

I have to thank the Federal University of Rio Grande do Sul (UFRGS), especially professors Vania and Luís Alberto, for always giving me space to explore my creativity, even though it sometimes went to unimaginable places. Thanks to the professors at PUCRS's specialization course in digital game development for helping me to develop my knowledge of games.

Thanks to everyone at ThoughtWorks, especially to Rodolfo and Marcelo, who carried out the game development efforts with me. Thanks to all the people from ThoughtWorks who helped me; the list is long, but I will mention the six most important: Aneliz, Enzo (who I shared many speeches on CI, CD, and TDD, and many hours of fun), Renata, Otávio, Ana Daros, and Braga. A special thanks to Paulo Caroli for saying to me that the topic of lean game development should be much more than a 25-minute speech and for providing the material on the Internet so people could check it out.

Of course, to Louise, Nancy, Jim, and Apress for believing in lean game development. A special mention goes to Vivian, Bianca, and Casa do Código for the Portuguese version and for helping me with the book.

Foreword

Have you ever noticed how happy kids are when they are playing? Hopscotch, hide-and-seek, pat-a-cake, tag, *Minecraft*, *Halo*, etc. And as some people get older, they cannot stop playing (and having fun) their whole lives! Those are the types of people who work creating games. Don't get jealous—this is the industry of game development.

But you already know this. You have this magnificent book on your hands because you admire software and games. You want to continue playing, having fun, collaborating, creating, and making games evolve.

Well, Julia is just like you. She also likes and works with game development, but she is aiming for more. She wants to improve the lives of those who build these games. To do that, she wants to hone the effectiveness and efficiency in the creative processes of game development. Julia has explored a mix of practices and techniques (lean, agile, design thinking, lean UX, and lean startup, among others) and has gotten some excellent results.

People who have gotten the opportunity to work with her have said, "Wow, this way of working is amazing!" Many others—those who heard about Julia kicking ass but haven't had the opportunity to work directly with her—have asked her for references about this new way of building games.

Julia answered their requests with this book. In it, she has written, simply and effectively, about how she leads game development. In short, she explains how to start with a collaborative workshop to decide on the game's direction, deliver a minimum viable game (MVG) frequently, develop while being driven by tests, and integrate the code continuously.

In this excellent book, Julia shares all the phases of how she created *Super Jujuba Sisters*, a 2D platform game, from its conception to its MVG incremental deliveries.

Enjoy and have fun learning. Good reading!

Paulo Caroli

Introduction

The game industry has been resistant to agile methods, but a great deal of this resistance comes from frustrated attempts to implement some of the tools proposed by agile methods. I believe that agile methods and tools should never overlap the propositions from the Agile Manifesto (http://agilemanifesto.org/).

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Rarely have I seen a place where individuals and interactions are superior to processes and tools. In fact, every time I hear game developers talking about their frustrated attempts with agile methods, I hear them talking about how they used the processes and the tools in an inflexible way, which is unfortunate.

Furthermore, I believe test-driven development was born from the second principle ("Working software over comprehensive documentation") since, in my opinion, extensive tests can be translated as documentation, in addition to helping the working software. I think that in the game world, client collaboration is an essential stage of development since the software is made especially for entertainment. Another important aspect is that the market is constantly changing, as does life, so it's important to adapt to the current needs.

Riot Games is a great example of what game development should be because the game company has already moved past agile in many aspects.

It is in a lean stage of development. Check out more information about Riot Games on its engineering blog (http://engineering.riotgames.com).

Here are a few more interesting links about agile game development:

- *Video Games: Agile Development, Say No to Rage*: https://www.youtube.com/watch?v=Y3BBm3k4YMI&feature=youtu.be;
- *Common Mistakes in Agile Scrum Game Dev, Arch Creatives*: https://www.youtube.com/watch?v=epWidgZ259M&feature=youtu.be
- *WGDS13: Emil Harmsen: Agile Game Development, WGDSummit*: https://www.youtube.com/watch?v=OSjytjjFYsE&feature=youtu.be

This book is primarily for game development teams in various types of companies. These are people who, in a certain way, have faced the challenges presented in this book.

Perhaps the most important of all, this book is focused on indie and small game development companies. Obviously, large enterprises can also use the presented methodologies; however, larger companies have many more adaptation issues and usually need external help to identify their weak and strong points and where things are working fine or not.

I also understand that others interested in game development will have a good starting point to begin game development using the strategies presented in this book. However, the principles can be applied to any work environment, whether software development, artistic, or government.

I hope this book brings you many hours of entertainment and lots of knowledge. There's no need to know how to program to read the book, and every stage involving programming is focused on tests to help game programmers. Furthermore, artists and designers can use the book to develop their development techniques when working with game teams.

Last but not least, business analysts and project managers (the people with the most interest in guaranteeing that the present methodologies work and the people responsible for pushing people out of their comfort zones) are the main target audience of this book.

CHAPTER 1

Introduction

This book's goal is to present a new way of developing games to teams with little, or no, experience in agile or lean methodologies.

Note The word *lean* derives from the Toyota Production System, which has a systematic method for waste minimization without sacrificing productivity.

If you have some experience with game development, now is the time to put it aside. It's time to forget everything you know and, with an open mind, learn something new. The goal of this book is to provide you with a game production model that prevents waste, reduces bugs, and offers continuous reviews; the book even offers a sequence of steps to eliminate unnecessary tasks. When I developed this methodology, I was thinking of small game companies, but it can be used in enterprises as well.

Why Lean Game Development, Not Agile Game Development?

Lean is something beyond agile. In fact, many game companies have been unsuccessful while trying to use agile methodologies. In addition, sometimes companies confuse agile with scrums, thinking of scrums as the only available agile tool.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_1

The same happens with extreme programming (XP), and this confusion can have disastrous results. In fact, it's common to see companies adopting scrums but not adopting the basic principles of agile, which overloads the development teams.

Lean game development can meet all the needs of the game industry, but there are certain game-related aspects to take into account. For instance, game production is never 100 percent efficient since you can't predict every possible problem, and it is far more difficult to find the "minimum" in a minimum viable product (MVP) in game development than in other industries. If you set fixed deadlines, the best you can expect is to get very close to them because unexpected problems will continue to happen, even after the deadlines. It's necessary to behave organically regarding changes, building in the ability to adapt to the environment.

Lean game development offers a methodological alternative to game development that can help you to eliminate waste, get results as fast as possible, strengthen and empower teamwork, and allow you to have a better view of the whole work. How do you improve this visualization of the work? Kanban (which literally means a visualization card) is a classic tool that lean development teams use.

That said, it's important to emphasize that in no way are lean, scrum, XP, or kanban exclusive. They can be used together, allowing the best features of each one to be used.

How Do Lean and Agile Relate to the Game World?

Lean game development is, above all, strongly inspired by agile and can take advantage of agile's tools to develop software. Therefore, let's look at the Agile Manifesto and interpret it to represent the vision of games. For such, I suggest the following point of view for games:

- *Individuals and interactions* over processes and tools
- *Games running* over comprehensive documentation
- *Audience collaboration* over sales
- *Spontaneous development* over following a strict plan

Games and Software Relate Much More Deeply

To successfully understand lean game development, you should first understand that digital games are also software and that software can be seen as a cooperative game of innovation and communication. Games are not only for children; games are used to describe everything from romantic experiences to advanced military strategies, but they can also be used as another form of software development.

GAMES FOR MILITARY STRATEGIES

The Blitzkrieg board game was used for a long time to help train army officers. The game is set in World War II, in which two armies confront each other: Great Blue and Big Red. There are five countries, but the alliances are not built strictly and can vary depending upon how the game is played.

The game has three modes: simple, advanced, and tournament. One of the most interesting aspects is that advanced mode offers many combat units, such as infantry, artillery, armored, assault, shots, bombing, etc.

Unfortunately, the game is usually hard to find, maybe because it's old. Figure 1-1 shows the (gigantic) board of the game with the different colored pieces.

Figure 1-1. *Avalon Hill's Blitzkrieg board. Source: boardgamegeek.com*

When someone proposes to play a game, hundreds of alternatives come to mind: tic-tac-toe, checkers, chess, poker, 21, hide-and-seek, ping-pong, and so on. But usually games fall into certain categories (or several) that help players to realize how they are played and what the goals are.

- *Zero-sum*: These are games in which each user plays on an opposite side, and if one side wins, the other loses. Examples include checkers and tic-tac-toe.
- *Non-zero-sum*: These are games with multiple winners and multiple losers. Examples include poker and hide-and-seek.
- *Positional*: These are games where the overall state of the game can be determined by looking at the board. Examples include chess and tic-tac-toe.
- *Competitive*: All the previously mentioned games are competitive, in which there's a clear notion of winning and losing.
- *Cooperative*: In these games, people play together to win, or until they find it necessary.
- *Finite*: These are games that have an end.
- *Infinite*: These are games where the primarily intention is to keep playing. In other words, the goal is to remain in the game.

What Kind of Game Is Software Development?

Many people see software development as a positional game, with a cycle of small victories and a clear goal. But a software game is much more than positions on a board and much more than a game that your team has to win.

Software development is a cooperative game, in which all pieces must help each other in order to reach each one of the goals. Think of a survival game, in which each member of the team has a specific and unique skill that is useful to the group's survival. The software development process is similar to the concept of a cooperative game. There should be no leader; instead, a group of people unite to make the best decisions and divide tasks the best way possible to survive (win).

Where Did We Go Wrong?

Unfortunately, over time, people got the idea that stricter and heavier methodologies, with more control and more artifacts, would be "safer" for a project's development. However, no one likes to play games with hundreds of rules that need to be remembered every minute in order for the player to perform correctly.

The best games are those that can be played in a relaxed way so that if a rule is broken, there won't be big consequences for the result. Furthermore, games that allow the player to be creative and imaginative tend to provide much more cooperation. You just have to observe kids playing board games to realize this.

Taking this into consideration, why not apply this to software development? Let's look at an example of the negative effect that rigidity and heaviness could have on the classic board game Monopoly. Imagine that, besides the players, you have a person solely responsible for administering the bank, one for administrating the real estate, another one for administrating the real estate bank, one for administrating your life (chance cards), one police officer for administrating prisons and the flow of characters, another one to roll the dice, and so on.

This type of model is the software development model used in most companies: it's highly hierarchical, it has several types of control over individuals, it has strict rules, it's difficult to play, and it's aimed

at exploring others. How can this model be superior to a relaxed, fun, creative, and cooperative model? A game's manifesto, the framework, and the methodology should not be applied in a rigid and immutable way. After all, a game must be fun.

Probably, this presumption that heavier methodologies are safer comes from the assumption that project managers can't look at the code and evaluate the degree of development, the status, and the project situation. Adding weight and rigidity to the model won't make the fear and insecurity regarding the project better, however. In fact, the consequence will be making your team delay their work and miss the deadline.

To achieve satisfactory results, always keep in mind that developing software is a team game, in which the project manager is not superior to anyone else. Remember, there are ways to document while the code is being written, and there are ways of visualizing the software development without increasing the pressure on the team.

The following are some prejudices of the game industry regarding the agile methodology:

- Test automation in games is much more complex than in other software industries.
- The visual cannot be tested automatically.
- Hiring children to test the game is cheap.
- The current business model in the sector is based on prompt games**.**
- "I don't like scrum," because scrum was the answer to agile methodologies.
- The sequences of games are not iterations.
- Art cannot be iterated, and games are art.
- Games are developed so the users play longer, not save time.

- Games are not testable.
- From production's point of view, continuous delivery is not attractive to games.

Thus, it's important to understand the basics of lean and remind yourself that software development is a cooperative and fun game, in which all pieces are important. It's a game that always produces more knowledge. Ideally, it must be managed organically and with low hierarchy to prevent waste (of human potential, of time, of excessive documentation, of conflicts, of stress, etc.).

This book will cover several aspects of lean development, such as the basic aspects, the inception, and MVPs, and will apply them to games. You will also learn how to use test-driven development, how to use continuous integration in games, and how to generate hypotheses. Lastly, you will look at how design and build are different and learn more about tests, measurement and analysis, and the generation of ideas.

Summary

In this chapter, I talked about the relationship of lean and agile in the game development world. Also, I discussed the deeper relationship that games and software have, including how software development can be seen as a game.

CHAPTER 2

First Steps with Lean

This chapter explains lean in a deeper sense and how it relates to game development. Also, it presents a visualization of the lean game development cycle. Finally, it introduces some places where lean game development can take advantage of agile methodologies.

Seven Key Principles of Lean

When starting to talk about lean in more detail, it's important to cover the seven key principles of lean.

- *Eliminate waste*: This includes avoiding the following: producing disorderly and unnecessary inventory and requirements, giving excessive importance to defects, processing unnecessary information, and creating long waiting times. To reach those goals, avoid unnecessary code and features. Another important consideration is to not start more activities than can be completed.

  From a business point of view, it's necessary to elaborate on the requirements so they are easily comprehended and to avoid changing them constantly. Especially, avoid bureaucracy. A usual problem is that inefficient communication can lead to a lack of comprehension regarding the job to be

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_2

done. From a developer's point of view, it's important to be careful to not let the job be incomplete or end up with defects and quality issues in the finished code. But maybe the most important thing is to prevent unnecessary changes in the job tasks.

- *Build with quality*: Quality problems lead to waste; in addition, it's a waste to test something more than once. To avoid quality problems, you can use pair programming and test-driven development. Both are fundamental tools, and both will be described in the coming chapters of this book.
- *Generate knowledge*: Generate knowledge while you're working so that the whole team can follow the software development process and have the technical ability to deal with problems. A usual way to generate knowledge is through pair programming. Wikis, dev huddles, and docs are other tools to share knowledge.
- *Postpone commitment*: Complex solutions should not be treated rashly, and irreversible decisions should not be made hastily.
- *Deliver fast*: It's common for people to spend a lot of time thinking of requirements that may or may not come up. It also happens often that they get mentally blocked or start thinking of solutions with excessive engineering. You can avoid this problem by gathering the right people, keeping things simple, and using teamwork.

- *Respect people*: The workplace should be pleasant, so never forget to be courteous with people. No matter what your position is within the company, you must always seek for equity between roles.
- *Optimize the whole*: This principle seems simple and intuitive, but it's usually not taken into account. It's important to identify fails, propose solutions to them, and look for feedback. A whole is not made solely by its parts but by people interacting.

While using the methodologies described next, it's always important to keep lean principles in mind.

Lean Inception

An interesting stage of software development in the lean methodology is the lean inception. Briefly, the *inception* is a workshop, done typically in a week, with many activities of alignment and goal setting. The product evolution ends up being represented by a sequence of minimum viable products (MVPs) and their features.

The main goal is to define the scope of what is being created so that the team has a clear view of the path to be followed, usually through the *MVP canvas*. The MVP canvas will be explained in more detail in later chapters, but you can find a brief definition in the "MVP Canvas" sidebar.

MVP CANVAS

The MVP canvas gathers elements from design thinking, lean startup, and business directives. It's a template to validate new ideas and question the existing ones. It's divided into seven branches.

MVP vision: What product vision must this MVP deliver?

Metrics for hypothesis validation: How do you measure the results of this MVP? And from the business point of view?

Outcome statement: What knowledge are you seeking with this MVP?

Features: What do you intend to build with this MVP? Which actions can be taken in order to simplify the MVP?

Personas and platforms: For whom is this MVP?

Journeys: Which user journeys will be improved in this MVP?

Cost and schedule: What is going to be the cost and the schedule for this MVP?

Read more at www.caroli.org/.

How Does This Apply to Games?

The main tasks of the lean inception are to come up with the game features, its basic game design, and the tools to be used. In short, there's a whole series of possible applications of the inception. It's also important to get the whole team engaged to increase motivation and build empowerment within the group.

Lean PMO

The *project management office* (PMO) is a group of people (or an individual) responsible for keeping an integrated vision of the strategic plan throughout the whole product development, including the deadlines and the costs. These people are responsible for gathering the company's portfolio to guide, plan, and organize the activities of the projects in the best way possible.

A lean PMO manages the game development and organizes and keeps track of requests and MVPs. The PMO does this by considering the body of work, without losing itself in agile technical details, like with what can happen with extreme programming (XP), scrum, and kanban.

The PMO's main role is to guarantee the continuous delivery of the game. It needs to be aware of the whole and not the details and periodically monitor the product development.

How Does a Lean PMO Apply to Games?

To talk about continuous delivery when the game is not even on the market yet seems presumptuous, but continuous delivery doesn't have to be necessarily for a player. The idea behind this is to manage the development steps in such a way that the product keeps evolving.

Lean DevOps

The function of *DevOps* is to connect the practices of DevOps to the lean MVP perspective (which will be explained in the next chapter). DevOps refers to the practices that the team uses while creating the product.

DevOps doesn't have to be executed by one single person; it can be used by a group of people, by different people in different moments, and in different practical activities. It includes working with features and the user stories.

How Does It Apply to Games?

You can, for instance, designate people who are responsible for organizing and applying the techniques, methodologies, and tools that the team has, as well as guiding the game's deployment.

Kanban

Kanban is based on Toyota's just-in-time model. The method consists of visualizing the workflow and from there acting on the process in order to not overload the members of the team. The process is exposed to the team members using the visual management approach, from the early stages to the final delivery.

Physical or virtual boards can be used, such as Trello. Some kanbans are divided into columns and rows, but not all of them need to be. There are some very creative implementations; just search for and use the one that matches your team's needs.

Kanban is composed of several "cards," with each one representing an action that must be taken. The action's degree of completion is marked on a panel, usually called the *heijuka board*. This system has some advantages; it's easy to identify waste and can lead to faster cycles that increase productivity.

Color coding can indicate the status of the card and enable people to identify delays, allowing them to be solved first. Different cards in the same zone usually mean a flow fail and must be resolved.

From my point of view, this helps to eliminate long daily meetings, the need of a scrum master, and other wastes. Many times laborious tasks can be divided in three smaller ones, also increasing efficiency.

The work-in-progress (WIP) concept is used to limit activities that will be "pulled" in the kanban method. From kanban's point of view, the next activity is solely pulled when the WIP has work capacity. The WIP

restrictions identify bottlenecks and possible problem areas in the process, helping to make team decisions (you can read more at www.caroli.org/).

How Can You Take Advantage of Scrum?

As you probably know, *scrum* is an agile framework for the completion and management of complex projects. It is highly recommended for projects that have requirements that change rapidly.

Its progression happens through iterations called *sprints*, which usually last from one to four weeks. The model suggests that each sprint starts with a planning meeting and ends with a review on the work done. So, it consists of short and cadenced cycles, with alignment meetings to monitor the work evolution and team efficiency.

Other stages, such as the retrospective and grooming stages, must be considered keywords. Typically, it corresponds to team ceremonies that aim to review the steps and plan the next ones.

In this framework, there are often meetings called *dailies* that allow the team to check on the progress of the tasks. In these meetings, all the team members stand up and answer three questions: "What did I do yesterday?" and "What am I going to do today?" and "What is blocking me?"
The answers to those questions are going to feed the team's reflection, re-orientation, and decision-making.

Using scrum encourages the empowerment of a multifunctional and self-organized team. The team's efficiency depends on its ability to work together. The team should not have a leader, and everyone should help make decisions.

SCRUM KEYWORDS

Here are some important terms to know:
Scrum master: This is a person familiar with the framework, whose goal is to facilitate the team in the scrum process. It can be anyone from the development team.

Product owner: This is a representative of the company, of the clients, or of the users. This person guides the team on the product's vision. The person must lead the development effort through explanations and priority setting.

Continuous Integration

Continuous integration is a software development practice in which team members frequently integrate their code. Each integration is verified through a compilation and automated tests to detect integration errors quickly.

Integration supposes a high level of automated tests. You'll learn more about continuous integration later in the book, but the usual tools for games are Unity Cloud Build and Jenkins.

Going from Build Measure to Lean Game

Next, I will briefly present some steps of the lean software development process. You can adapt the steps in Figure 2-1 for game development, but the central idea is to generate ideas, build, code, measure, get data, learn with data, and generate new ideas, all in a loop.

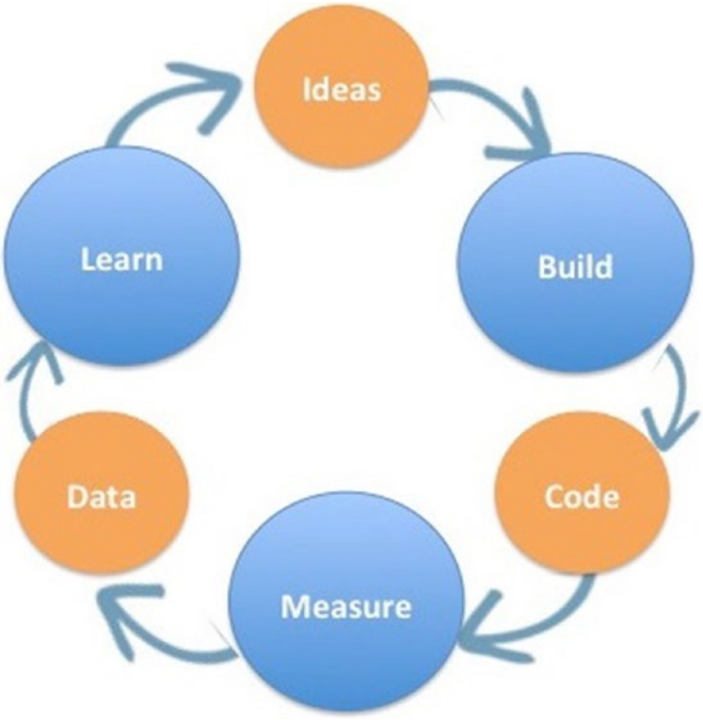

Figure 2-1. *Build-measure-learn diagram. Source:* `www.caroli.org/what-to-build`.

Figure 2-2 shows the outcome of Figure 2-1 when applied to the game development process.

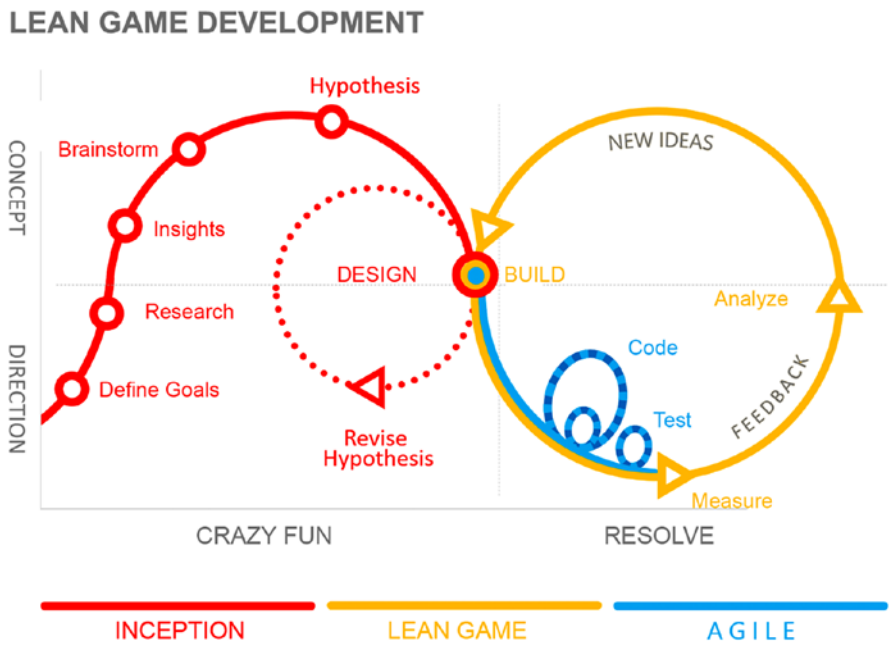

Figure 2-2. *Lean game development diagram. Inspired by:* `www.lithespeed.com/lean-ux-dont-part-1-3-2/`.

The following summarizes the steps shown in Figure 2-2:

1. *Inception*: The concept is set at the lean inception stage. It comprises a series of substeps, as follows:

    a. *Define goal*: The team gathers to build the ideals in which the inception is going to be based on.

    b. *Research*: The team gets information on the possibilities allowed by the goals.

    c. *Give insights*: These are moments of inspiration, when team members aim to "think outside the box."

   d. *Brainstorm*: This is when you expose your creativity. Don't be afraid of your ideas. The more absurd, the better.

   e. *Make hypotheses*: This moment allows you to turn all ideas into measurable goals.

2. *Design*: In my opinion, this is one of the most interesting moments because it's when the game becomes alive.

   a. *Revise hypotheses*: With the current game design, you can do the first measurement, which consists of determining whether the hypotheses created are proper to the context.

3. *Build*: Here you define what is possible to do and how to do it.

   a. *Code*: Here you develop the art, animation, code, models, and so on.

   b. *Test*: Here you get the first feedback of what is being done, especially if it's automated.

4. *Measure*: With the build ready, you must measure the impact of the game. There are many ways of doing this, but feedback from the community is a great way to start.

5. *Analyze*: Here you get an understanding of everything that was measured.

6. *Iterate*: With lean, it's always important to iterate to generate new ideas, concepts, features, and knowledge. To iterate means to start again.

Looking Deeper at the Inception

The inception is a useful activity to kick-start the project and define the stories of users. At this stage, the following are the important questions:

- What are the goals of this project?
- Who is going to use this system?
- What are the roles each team member will perform in the game development?

Once you set some goals, the team can start to develop the journeys from the personas (you'll learn about the concept of personas in the next chapter), using this sequence: *As X, I want Y and, therefore, Z, so what does X want from the system in order to get Z?* From these questions and answers, you create stories on which the software must be based.

How Does It Apply to Games?

As in other examples, the model presented is not perfect for games because there are some ideals missing. So, how can you adapt and improve this model? I suggest asking the following questions:

- What is the game's narrative?
- What is the central characteristic of the game?
- What do you want to achieve with this game?
- What kind of user are you aiming for?
- How can the user play the new game?

As you answer these questions, you can determine the players' journeys, as shown here:

- As X, I want to be able to do Y; therefore, I want Z.
- So, what does X want to do in the game to get Z?

Test-Driven Development

Test-driven development (TDD) prevents problems by writing tests before developing code. If developers know how the game is tested, they are more likely to think of all the test scenarios before coding, and this can improve a lot of the code design. The following are the most common steps of TDD:

1. Write the test.
2. Run the test and see it fail.
3. Write the code.
4. Run the test and see that the test works.
5. Refactor.
6. Iterate.

Chapter X focuses on TDD, especially regarding games.

Lean and Games

With all this basic knowledge you've seen so far, it's possible to understand superficially what lean is and how you can apply these topics to games. This is a good time to reflect on what Figure 2-2 can teach you and what steps you can take in lean game development.

Summary

In this chapter, I reviewed the seven key principles of lean development. I briefly introduced lean practices and how they apply to games. Finally, I showed how the lean game development cycle can make the development process richer with a few agile concepts.

CHAPTER 3

An Inception in Practice

I explained what an inception is in the previous chapter; in this chapter, you will see why it's so important. Many times, before starting a project, team members don't know each other or their environment. So, one of its functions is to increase the chemistry between team members.

There are several techniques called *ice-breakers* that help the team with this. After covering some ice-breakers, I will cover topics such as goal setting, strategies and game scope, mapping, and features prioritization. These techniques are the basis of this chapter.

Inception Defined

The idea of inceptions comes Paulo Caroli's blog description, provided in the "How to Model an Inception" sidebar.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_3

HOW TO MODEL AN INCEPTION

Usually, you can follow this simple model:

- The inception lasts from one to five days, depending upon the game complexity and the necessity.
- You count on the presence of all development team members, publishers, business analysts, and other parties interested in the game's success.
- A "war room" is booked for the inception for as long as it's necessary.
- Having lots of colored sticky notes will allow you to do lots of brainstorming.

You can find more information about inceptions at the following locations:

- www.caroli.org/back-to-the-basics-what-made-this-agile-inception-especial/
- www.caroli.org/user-stories-and-business-hypothesis/

Anatomy of an Inception

Here's an example to walk you through the inception process. My team wanted to develop a computer game, which is our passion, in a way that we would gain advantages from the techniques we work with daily. We decided on a 2D game in which the character Jujuba must survive a series of challenges to rescue her sister Xuxuba from the evil Marcibal, a member of the Xandra gang.

We needed to create a model of game development that would allow other game companies to develop more efficiently and with more quality. So, we needed to start from somewhere, i.e., the inception, so we gathered people who were interested in it and started to think.

Our inception was attended by some people from outside the development team since the team was small. It lasted nearly a day because of limited time availability, but its stages were finished throughout the week by the development team.

The following sections will review the inception ideation stage, which consists of defining goals, research, insights, brainstorming, and hypotheses generation, as depicted in Chapter 2's lean game development diagram.

Defining Goals

This stage involves creating a set of goals that the team defines as priorities of the game.

Researching

We asked all the team members to research games available on the market and the general public's interest in games. We also asked each team member to share their gaming interests and experience. This helps to create a benchmark of what a game should be like.

Generating Insights

This stage is for reaching an understanding of the game's needs and respecting its goals in order to generate clear ideas on how to reach those desired goals. It's an epiphany of narrative and art in the context of games.

Insights are different from brainstorming because insights represent a comprehension of ways to achieve goals, and many times they serve as a foundation for the brainstorming session, which aims to generate ideas based on insights and on goals.

We presented the more relevant and fun ideas to the game development team. They are provided in the following sections.

The Game's Narrative

Here is the game's narrative:

- It's a *Mario*-like game.
- The character Jujuba faces enemies from a gang.
- Marcibal and the Xandra gang kidnap women to perform satanic rituals.
- It's a post-apocalyptic world.
- There are hidden and explosive things.
- Who doesn't like cats and bombs?
- There are stun guns.
- Kamehamehaaaaaa!

The Game's Main Characteristics

These are the main characteristics of the game:

- It's a 2D platform.
- Characters can walk, jump, and fall.
- Characters can die and be reborn (that is, they have several lives).
- Characters can kill enemies.

The Game's Intended Outcome

These are the game's intended outcomes:

- To prove that lean game development can be used for game development
- To prove that small projects can take a lot of advantages from the agile and lean methodologies
- To provide a lot of fun and learning

The Game's Intended Audience

This is the game's intended audience:

- People who want to have fun with an immersive game
- People who like to focus on fast games
- Users who like to focus on games with increasing difficulty levels

How Are Users Able to Play?

This is how users can play:

- Moves are restricted to the keyboard, namely, the arrow keys, the spacebar, and the A, S, D, W keys.
- Will there be a video game joystick?
- There are no external resources.

Prioritizing Ideas

After gathering all these ideas, it's necessary to prioritize them as follows:

- Each person builds their favorite sequence (in case of the game narrative, we chose to order the three favorite). When everyone is done, the classification is exposed.
- Using the sequence generated by the preference classification (preferential vote system), you create a board containing the sequences in each one of the items listed previously.
- From these sequences, you create action items that again will be voted on and ordered, but this time they're all in the same group.
- The items are used in a discussion to define which ones have priority and which ones are easier to solve first.
- With these action items, you can generate the minimum viable products (MVPs) described in the next chapter, the personas, and the user journeys.

Developing Personas

Personas are a way for the team to visualize the profiles of its main users, while developing the game and during the inception (Figure 3-1). They gather attributes such as personal interests, age, job, hobbies, and so on.

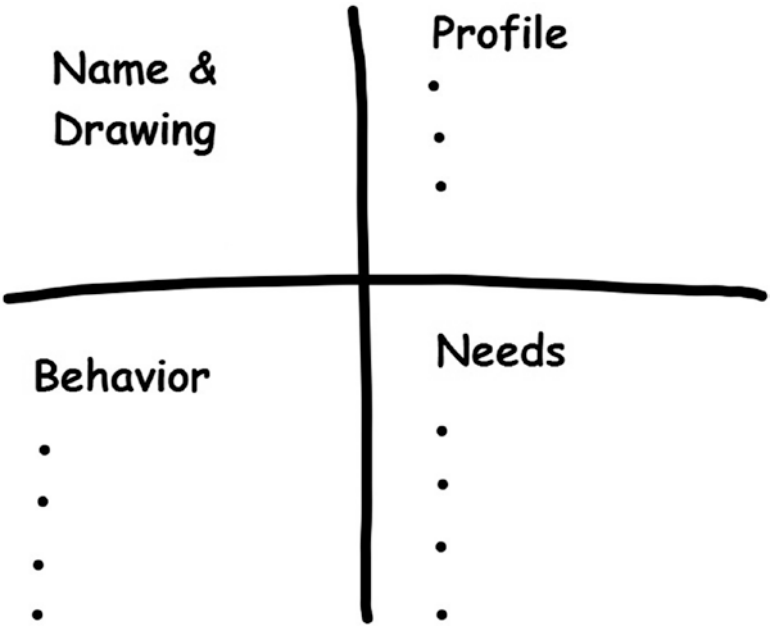

Figure 3-1. *Natalia Arsand's personas template. Source:* *www.caroli.org/atividade-identificando-personas*

Follow these steps when creating personas:

1. Ask the inception members to divide into pairs or small groups, and then deliver the template shown in Figure 3-1 of personas to each group.
2. Ask each group to create a persona, using the template as a reference.
3. Ask the participants to present their personas to the whole team.
4. Ask the team to switch groups and repeat steps 1 to 3.

At the end of the activity, a group of personas will have been created, and the different types of game users will have been described. The stakeholders who know the project's goals must participate actively in the

activity, helping the team to create the personas and suggesting changes in their descriptions, as needed. Furthermore, many of the ideas that came up at the inception can be well used here, showing the need of previous research on the game. (For more information, see www.romanpichler.com/blog/10-tips-agile-personas/ and www.caroli.org/new-talk-user-story-a-holistic-view/.)

Journeys of Personas

The following persona wish list was copied exactly as it was developed during the inception for our *Mario*-like game:

- Like Carlita, I want to be able to make the "princess" save the day; therefore, I want the *main character to be a girl*.
- Like Rodrigo, I want to be able to play for hours. Therefore, I want a *different game experience in each level*.
- Like Thais, I want the enemies to be real and personified; therefore, I want them to be based on *real and credible situations*.
- Like Guilherme, I want the *game to remind me of the old days* of video games; therefore, I want an experience like *Mario* 2.0.

Stories

Stories are the stage in which you extract the concepts of art, the features, the mechanics, and the user experience of personas, as follows:

- Carlita wants Jujuba to be a *strong character* and, like in *Frozen*, wants the plot to be about sisters.

- Like Rodrigo, I want *every level to present new challenges*, such as new enemies and new skills.
- Thais wants the characters to produce a *sense of familiarity and/or humanity*.
- In Guilherme's opinion, it's important to, at least, *jump, walk, fall*.

Brainstorming

The following is a paraphrased series of rules designed to make a more productive brainstorming session. These were taken from Gabriel Albo of ThoughtWorks and were inspired by the ideas proposed in *7 Tips on Better Brainstorming* (OPENIDEO, 2011).

- Have one conversation at a time.
- Try to create as many ideas as possible.
- Build ideas over others' ideas.
- Encourage crazy ideas.
- Be visual.
- Maintain the focus.
- Don't criticize or judge.

Creating Hypotheses

Lean hypotheses are the main way a team can guide and evolve the development of the game. With generated hypotheses, an MVP can be elaborated on and use the hypotheses as validators of the development.

Chapters 4 and 5 cover MVPs and hypotheses in more detail.

Summary

In this chapter, you saw how an inception works and what kind of information you can get from it such as narratives, aesthetics, and game features, as well as your goals, your users, and basic mechanics. You also saw how you can do a prioritization of ideas generated in an inception and how to develop personas that will guide you through development.

With this information in hand, you got the features that the game must present and an idea of their sequence. From this moment on, the concept of MVPs emerges, as well as how to prioritize them and order them.

In the next chapter, you will get into the details of how to build the MVPs, how users journey, and how personas and hypotheses are extracted and used so that they generate the concepts of value of delivery (MVPs) and validation of delivery (hypotheses).

CHAPTER 4

MVPs: Do We Really Need Them?

In this chapter, you will learn what a minimum viable product (MVP) is and how it can be applied to game development. I'll also cover how to deliver your game prototypes for validation and how to apply MVP canvas concepts to prioritize your prototypes. Finally, you will learn how these concepts are developed in a real project.

MVP and MVG Defined

A *minimum viable product* (MVP) is a minimum set of essential features necessary to have a functional product that delivers value to a business (minimum product) and that can also be used by the end consumer (viable product). It works incrementally so that each new MVP/increment delivers the next set of minimum features, allowing the product to add value constantly.

The concept of MVP is not so simple when it comes to games because in most cases companies want to deliver a finished and complete game, and their clients want the same. Therefore, you need a *minimum viable game* (MVG), but it's necessary to remember that a minimum viable game must add some value. In sum, an MVG is a wrapper expression for MVP in the gaming industry. MVGs can be interoperable, but as you will see in

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_4

this chapter, an MVG demands a little for being viable, and the delivery methods are not necessarily for the end user.

When developing, you need to figure out what the uniqueness of your game is. What quality makes it unique? Are all the features necessary? Is it necessary to develop everything from scratch, or are there tools to help the development? Do these tools satisfy all your requirements? Do you have the necessary feedback for your idea? Is your team integrated?

At this point, you need to understand these two distinct concepts:

- *MVP*: Defines the moment in which the game is presentable to the client (even as a demo)
- *The prototype*: An internal step to define the minimal development to achieve a valuable business result

When you develop games, especially those with new features, it's necessary to keep in mind that game development is made of prototypes of features and mechanics. No one starts to develop a game with advanced features, animation for all objects, and perfect art. Usually, a prototype looks like Figure 4-1, in other words, something similar to a 3D *Atari* game.

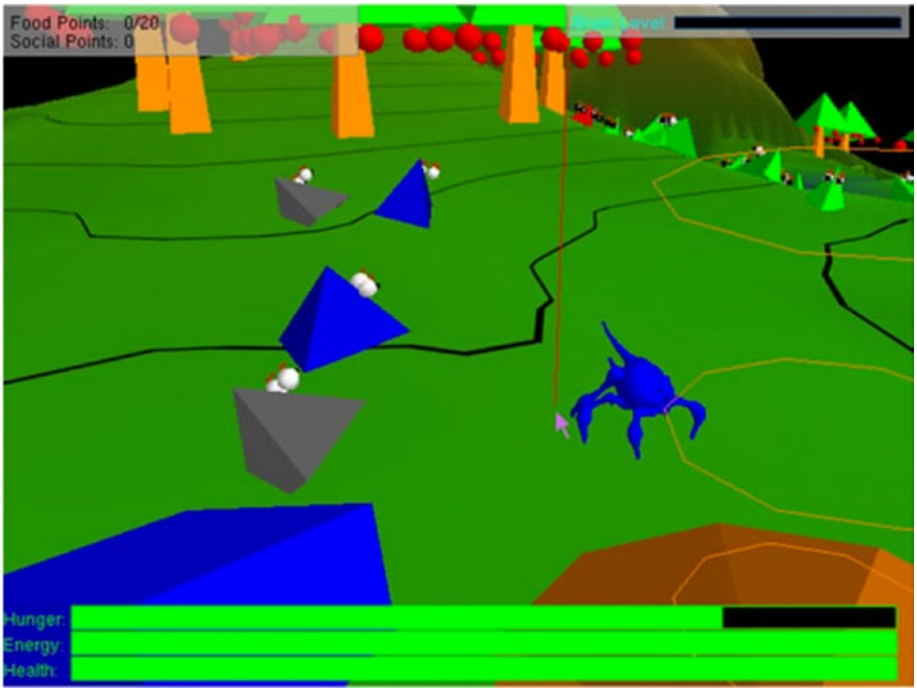

Figure 4-1. *Example of a 3D prototype. Source: New York Film Academy (https://www.nyfa.edu/student-resources/getting-the-most-out-of-your-prototype-game).*

Building Prototypes

The goal of a prototype is to find problems during the development process. You don't want a blocking bug in the game that surprises the client while having fun.

These disturbances are common in race games, for instance, when there's a jump higher than expected by the developers and the car gets stuck inside the mountain. To help to overcome this obstacle, the MVG has as many prototypes as needed, each one of them with a unique feature, a uniquely implemented mechanic, or a newly implemented piece of art or animation.

Don't try to implement thousands of things at the same time; your emotional well-being will thank you. An MVG is solely a generic concept that describes the minimum set of prototypes that have to be assembled to make your product commercially viable; a prototype is the minimum set of features that have to be developed in order to deliver value. Another difference is that the MVG delivers value to the client, while the prototype delivers value to the business.

The PO Role in MVPs/Prototypes

You know that an MVP/prototype needs a product owner (PO). Who can be the team's PO? Who can be the person in charge of assuring that the game's development is in sync with the desired vision? When you think of POs, you probably make associations with the idea of scrum. But ask yourself whether this association is the only way to go.

When a game is being developed, it's easy for the team to lose itself and spend years developing a new game each quarter. Remember the game *FEZ*, featured in the documentary *Indie Game: The Movie*? It was an amazing game, which me and my son played for hours. However, the game creator, designer, and developer was also the PO, the person responsible for guaranteeing the game's vision. This duplicity of roles meant the game took years to be launched, and consequently, this time had a negative impact on the product marketing.

The PO must be someone who understands how the users and the market are going to behave. In most times, the PO doesn't have to be someone in the company but someone who can make small suggestions to assure the integrity and correct vision of the product; it should be someone capable of setting the most important features and, preferably, who can follow the development of the game at all stages.

Another important role of the PO is to determine when MVGs are reached. For instance, a good practice for MVGs is to launch alphas and betas, like *Minecraft* did. *Minecraft* is one of the most successful indie games of all times, and its impact on the market was huge. This is undeniable (I myself have spent hours playing it with my son).

What, then, are good minimum viable products when it comes to games? Here are my suggestions:

- *Alpha*: An MVP be an internal launch in the company so the developers and designers can test it and give their opinion. Friends and family are good options.
- *Demo*: Who doesn't like to see a demo being played on the Internet? YouTubers are a great chance to get your game known by the audience, without forgetting that every kind of feedback can be positive for your game.
- *Beta*: In this case, a more general audience can be reached. Enjoy this step to get feedback; identify errors, bugs, and missing features; and determine the strong and weak points of your game.
- *Soft launch or early launch (prelaunch)*: These are important stages that allow a good evaluation of how much your product is worth. At this stage, the game is almost fully finished, so it's a good time to get as much feedback and fix the errors.
- *Marketing*: The marketing department is also part of the game team and will use its resources. What about a site that allows those people to play a small feature of the game?
- *Public launch*: In this case, it's desirable that your MVG is ready to be launched to a general audience.

- *Cross-platform*: The best option is to launch a game first on a single platform and then expand to others. So, this is the stage in which you release the game on different platforms. Many publishers have been adopting this model recently.
- *Updates*: This stage of an MVG is not mandatory like the previous items, but it's interesting to know that your game can be improved with updates. Without the Internet, it was impossible to improve games after release, but nowadays updates are frequent.

Getting More from Less

It's important to always be open to receiving feedback, even checking out the comments on the Apple Store. Another good way to measure your success is to use a Twitter account to receive important feedback about the game. In short, be proactive when searching for feedback.

Remember that people are more likely to give you feedback when something is not working right in the game. Try to compare the feedback you get with the initial vision of the game. Try to keep the following in mind regarding MVPs (in our case, MVGs and prototypes):

> *The minimum viable product is that version of a new product that allows the team to collect the maximum learning from clients with minimum effort.*
>
> —Eric Ries (2011), author of *The Lean Startup*

Now, you may be wondering how *minimum* must an MVP be? According to Ries, "Probably much smaller than you think." Does this idea also apply to games? I believe so, especially when you consider the stages described previously. If your game has a feature that can be cut out while

the game continues viable, it probably is not part of your MVP. In case of *Super Mario Bros.* (Figure 4-2), what would be the MVP?

Figure 4-2. *Image of the first version Super Mario Bros. Source:* *www.goliath.com/interest/super-mario-bros/*

In the minimum viable game of *Super Mario Bros.*, what features are required? It's likely that most people would think about mushrooms, Yoshi, Koopas, pipes, fire flowers, and hidden blocks, but certainly the answer is no to all of these items. The minimum features are "walk," "jump," and maybe "fall in the moat" in a single level. If this is fun and exciting, other features can be added to complement the game experience.

You don't believe this could be an MVG? I have a few examples of this in autorun mode: *Canabalt*, *Jetpack Joyride*, Tomb Runner, *Temple Run*, *Subway Surfers*, and *Ski Safari*. Essentially, it's necessary to test the game foundations before adding content because that facilitates feedback on the basics.

How about a role-playing game (RPG) such as *Final Fantasy*? The core of the game is to sum up to the battle system, which is similar to many other games, such as *Pokémon Red/Blue*. People used to play *Pokémon*

and *Final Fantasy* to have fun with *Pokémon* battles and people battles. But if the battle is bad, the rest of the game won't be interesting; thus, a game has to be good without quality graphics and with just names and colors.

Recognizing When a Game Is Not Viable

An important stage in game development is to recognize when your game is not viable. This is perhaps is one of the toughest decisions developers need to make, but it's necessary many times. No one wants to spend two years developing 2,000 features and find out that the game is not fun.

The following are some ideas of how to measure whether MVPs, MVGs, and prototypes are interesting:

- *Exploding kittens*: We all like board games (maybe because they are part of our childhood) because they allow us to interact with other people, especially when they combine kittens and grenades. Board games are products of high production, from the layout to the game design to the manufacturing. Therefore, it's important to know from the start whether people are going to be interested in the product you are developing. Aside from wanting to get your investment back, you will want some profit as well. A great way of determining whether your idea is worth it is to start a Kickstarter campaign, with videos, blog posts, Twitter, forms, and so on. That will allow you to see whether your idea has flaws.
- *Registration forms*: A good way to start your project is to create a page describing your idea and containing a registration form that allows people to participate on an e-mail list, get updates, offer feedback and make

suggestions, and get notifications when the game is launched. Perhaps you could even give these people early access to your game or use the page to measure people's interest in it.

- *Twitter wanderings*: Showing conceptual artwork and ideas of your game design can give ideas to other people but also can allow you to get some great feedback. You can also use Twitter to share other information as well.
- *Cardboard de facto*: This idea doesn't apply to every game, but it works for many of them. A board game inspired by a software game that is going to be developed can be a good start for setting mechanics, coming up with features, and exploring your creativity. Who knows, maybe it even turns into a project that can be sold. But the most important part is to check whether your idea is good and whether it's worth putting into practice.

Thinking Simpler First

A good way to start game development is to think about what is the simplest to do in an MVG and then set that as a goal. In my opinion, a great way to start developing games via MVGs is to go through the following list of games, identifying what would be each one's MVG. The following list is ordered from the simplest to the hardest.

- Race games: *Need for Speed*
- Top-down shooters: *Halo Spartan Assault*
- 2D platform: *Mario*

- Puzzles: *Tetris*
- 3D platform: *Mirror's Edge*
- First-person shooter (FPS): *Call of Duty*
- Japanese RPG (JRPG): *Final Fantasy*
- Fighting games: *Street Fighter*
- Action-adventure: *Tomb Raider*
- RPGs: *Skyrim*
- Real-time strategy (RTS): *Age of Empires*

Think of MVPs (in this case, MVGs and prototypes) according to Figure 4-3. MVPs must occur incrementally so that each MVP has some return.

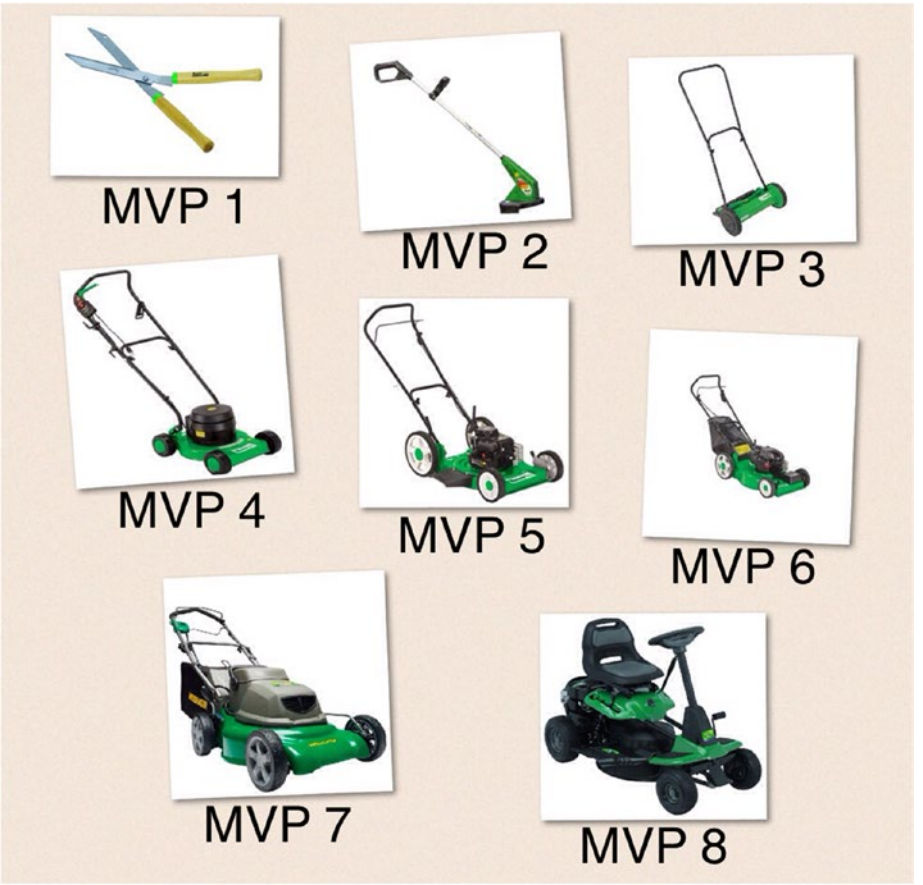

Figure 4-3. *MVP example. Source:* www.caroli.org/produto-viavel-minimo-mvp.

From the MVP Canvas to Lean Game Development

The MVP canvas is a model to validate ideas for a product. It was developed in the book *Lean Startup* and is a visual card that describes the elements of an MVP, such as the vision, hypotheses, metrics, features, personas, journeys, and even finances (www.caroli.org/the-mvp-canvas/). So, think of the MVP canvas like this:

- *Personas and platforms*: Who is this MVG for?
- *Journeys*: Which users will benefit from this MVG?
- *MVG's vision*: What is the vision for this MVG?
- *Features*: What is going to be built in this MVG? What is going to be simplified in this MVG?
- *Cost and schedule*: What will the cost and the schedule for this MVG be?
- *Statements*: What knowledge do you want to obtain with this MVG?
- *Metrics and hypotheses validation*: How are you going to measure the results of MVG?

One of the most important things you can learn with this system is the idea of "build, measure, learn." This represents the idea of developing the game, measuring the result in order to improve, learning about what was developed, and thus getting to build something better.

In this case, the MVP will be measured, built, and "studied" from a simple model of idealization. When building the MVG and testing it, you can use this line of thought:

- "We believe that…." *[the MVG's vision]*
- "We will…." *[the expected outcome]*
- "We know this happened based on…." *[the metrics to validate the game hypotheses]*

The following is how my team developed the previous items:

- "We believe in the well-structured fundamental mechanics of the game."
- "We will manage to provide better entertainment than a game full of features."
- "We know this happened based on the average time per game per user and in feedback that talks about the quality of our mechanics."

MVGs and Prototypes of Super Jujuba Sisters

Let's consider the game described in the previous section, called *Super Jujuba Sisters*, and its specifications. One of the ways to understand what its most important features are is to require that the team put all of them into a canvas and, later, to collaboratively organize them in order of importance. To do this, the team can put them into a graphic canvas, in which the *y*-axis corresponds to the priority in the game and the *x*-axis indicates the technical difficulty in implementing features.

From that, you can see a logical evolution of MVGs. Figure 4-4 shows this concept, and Figure 4-5 shows how you can sequence the features to establish the division of prototypes and MVGs.

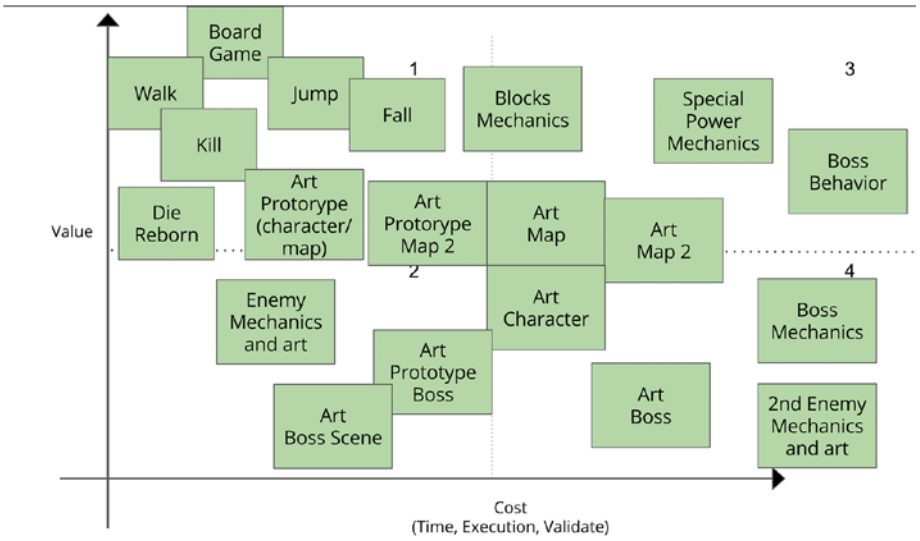

Figure 4-4. *Canvas showing the relation between priority and technical difficulty*

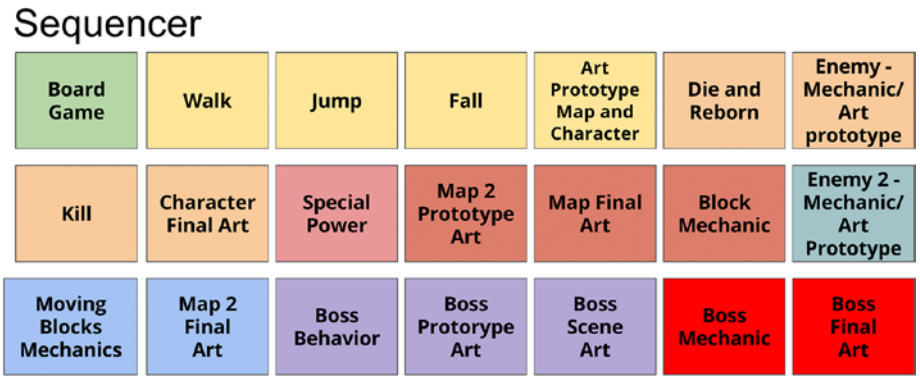

Figure 4-5. *Feature sequencer to generate MVPs and MVGs*

Dividing Your MVGs

Here is the evolution of the MVGs:

- Prototype 1
  - Game mechanics to walk, jump, and fall
  - Prototyped layout for the main character and map
- Prototype 2
  - Mechanics to be reborn, killed, and die
  - Mechanics of enemy 1 of the Xandra gang
  - Prototyped layout for enemy 1
  - Layout for the main character
- Prototype 3
  - Mechanics for special power
  - Layout for special power
  - Layout for enemy 1
- Prototype 4
  - Layout for new map
  - Mechanics of hidden blocks
- Prototype 5
  - Mechanics for enemy 2
  - Prototyped layout for enemy 2
- Prototype 6
  - Layout of scenario 3
  - Mechanics of moving blocks

- Prototype 7
  - Mechanics for Marcibal
  - Prototyped layout for Marcibal
- Prototype 8
  - Marcibal's behaviors
  - Marcibal' scenario layout

Splitting the MVGs or Increments

Therefore, you can split the MVGs as follows:

- *MVG0*: Board game to test mechanics and layouts
- *MVG1*: Prototype 1 + Prototype 2
- *MVG2*: MVG1 + Prototype 3 + Prototype 4
- *MVG3*: MVG2 + Prototype 5 + Prototype 6
- *MVG4*: MVG3 + Prototype 7 + Prototype 8

Summary

In this chapter, you learned what an MVP is and how this concept can be applied to games. You also learned how they are generated and how they can work from the point of view of development and continuous delivery. You learned about MVG metrics and the importance of creating hypotheses to guide you through iterations.

You also learned about the functions and the roles of people engaged in the vision created by MVGs. You saw an example of splitting MVGs for a game, and you learned that you need hypotheses and metrics to measure, analyze, and guide you through development.

CHAPTER 5

Generating Hypotheses

Along with the creation of minimum viable games (MVGs), hypotheses are one of the most important parts of an inception because they guide and measure your product through iterations. The MVGs determine which set of activities must be done, while hypotheses allow you to validate if your MVGs have the desired effect and if the product's vision is being met.

The first generation of hypotheses is intuitive because it's part of the inception, which is the initial stage of development. But what is the role of hypotheses? Their first function is to help develop the game design, help build the narrative, and (most important) help validate the MVG.

However, the main difference between hypotheses and stories is that stories guide the development while hypotheses continuously feed the backlog. Let's use the examples of stories generated at the example inception to generate hypotheses.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_5

Remember the personas presented in Chapter 4? There are two cases in which you want to use them as guides to your hypothesis generation. The journeys of personas and the stories developed are as follows:

- Like Guilherme, I want the game to remind me of old video game times; therefore, I want an experience like *Mario* 2.0.
- Like Carlita, I want to be able to make the "princess" save the day; therefore, I want the main character to be a woman.
- To Guilherme, it's important to jump, walk, and fall.
- Carlita wants Jujuba to be a strong character and, just like in *Frozen* (the Disney animated film), wants things to resolve between the sisters.

Based on these stories and journeys, you can come to the following hypotheses:

- We believe that building the features of walking, jumping, and falling into the game will result in a game with strong basic mechanics. We know that we are successful when we have positive feedback from manual tests and from third parties through the demo we will have.
- We believe that by building a strong nonstereotypical female character, with interesting functionalities, we will be able to reach the female audience, who will get excited by our game. We know that we are successful when lots of girls start to comment in our feed.

The standard model for developing hypotheses during inceptions is as follows (GOTHELF, 2013):

- We believe that by building the feature *[name of the feature]* for the audience *[intended audience]*, we'll achieve *[expected outcomes]*.
- We will know that we were successful when we have *[the expected market signal]*.

When Hypotheses Are Not Created from the Inception

Hypotheses help you to validate things in every cycle. They are fundamental to completing the measurement and analysis steps because they set the validations you have to make. For instance, if you get negative feedback from the female audience regarding the main character, you should stop and analyze where you are and start developing new ideas using that feedback.

Here are some examples of possible measures:

- When you realize the female audience is offended by the character, you have to read the criticism (for example, is it the clothes, inexistent armor, disproportionate body, nonsensical story, lack of story?), check what you did wrong, and think of new solutions to generate hypotheses.
- When you realize the audience has complained that the jump mechanics are visually ugly and give the impression that the character is moving abruptly upward and downward, you have to consider the problem. Is it an issue of improving the animation?

Is it that it doesn't look real? Is it that is not adequate to the game's needs? At this point, it's necessary to check in which stage you went wrong and suggest new modeling.

- If you realize that the audience complained that the first level is pretty cool but the other ones are exactly the same, making the game not that interesting, how can you make the other levels more appealing? Can you add new characters? Is the game mood poorly conceived? Are there proper obstacles in the scenario? It's necessary to see what is missing and come up with new ideas to make the user experience more interesting.

These examples show the need to elaborate on the hypothesis to assure you reach your goals. In Chapter 10, I will talk more about feedback.

Summary

In this chapter, you saw how to extract hypotheses from stories, journeys, and personas, as well as how hypotheses are great sources of support for MVGs. You also learned how to generate ideas and how they work in a cycle of many iterations.

Now you need to understand things from a more practical and less theoretical point of view so that you have the resources to guide you in the next steps of lean development, such as test-driven development and continuous integration. These topics are discussed in the next chapter.

CHAPTER 6

Test-Driven Development

Although the literature about test-driven development (TDD) is extensive and qualified, it's worth discussing TDD in this chapter in relation to game development since it's an extremely important subject. You will quickly learn about basic TDD concepts and why there is so much poorly tested code out there. You will understand how TDD can be applied to games in code and art and which practices can improve TDD.

TDD Defined

TDD is a software development style in which you have the following at your disposal:

- An enormous set of tests created by the person wrote the code
- No code in production without a test and without it having been tested
- A method in which the test is written first
- Code written in a way so that its only goal is to pass the test in a simple way

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_6

The process is that the code developer will write the tests, thinking in advance which will be necessary tests and how they should be ordered to implement code based on them. The code will be written after the tests and conceived only to pass the previously written test. One of the advantages of this method is that it implies "thinking before developing." Of course, a test expert can be consulted.

The following are additional important aspects of TDD:

- *Run the code often*: TDD allows you to get a visual of the code's general status.
- *Maintain the whole code tested in the smallest units*: The simpler and more unitary the code, the easier it will be to fix build issues and tests that fail.
- *Treat tests that fail as broken builds*: All tests should pass every time before continuing.
- *Keep testing simple*: Complex tests may not be testing the desired unit or acting as unitary tests (TDD can involve many more tests than the units).
- *Tests should be independent from each other*: Each test should be testing isolated units, features, or components.

Here are some advantages of TDD:

- The software becomes more understandable and easier to implement.
- TDD allows for simpler solutions.
- TDD results in fewer defects.

In fact, TDD can make the code more elegant. Furthermore, it can work pretty well as a form of documentation. I should emphasize that

people often confuse the two TDD concepts of unit testing and test first. To simplify, "TDD > test first > unit testing."

In other words, TDD uses a test-first strategy and uses unit testing for the designing and coding work. However, the use of unit tests in your code does not necessarily make your development test-driven because the presence of tests is just part of TDD.

Figure 6-1 presents a typical TDD cycle.

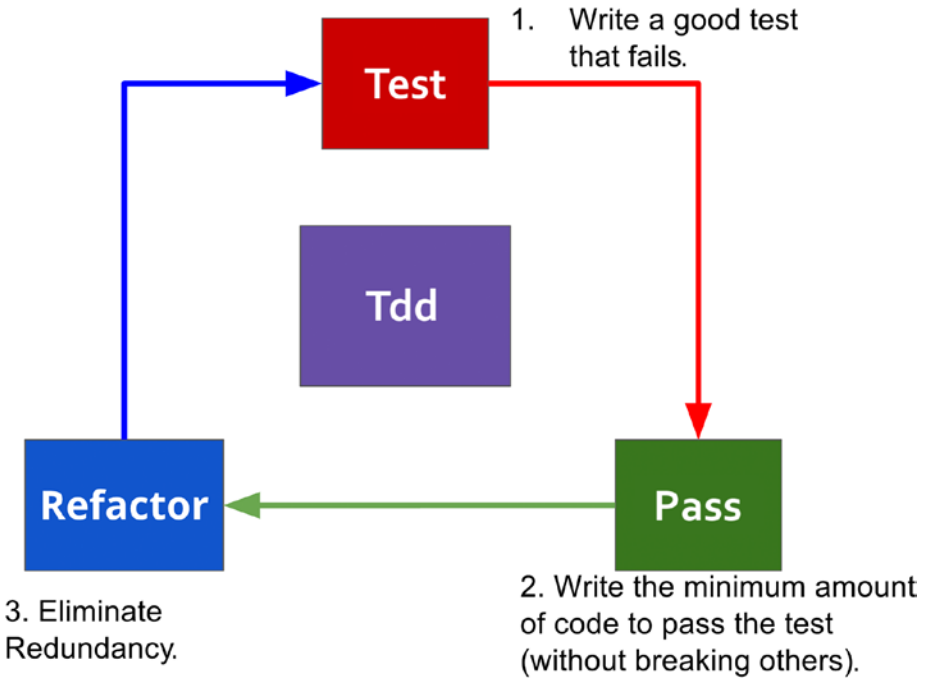

Figure 6-1. *TDD cycle*

As you can see, TDD occurs as follows:

1. *Write the test*: Without a test, you don't know where to begin.
2. *Run the test and see it fail*: What's the use of a test if you know it's going to pass? It *has to fail*.

3. *Write the code*: Now write the minimum code for that test.
4. *Run the test and see it pass*: "Wow! Now it passed!" Your code is correct.
5. *Refactor*: Certainly there are things that can be improved in the code.
6. *Iterate*: Go on to the next test.

Tests Are Good, But Why Is There So Much Poorly Tested Code?

I believe that most companies, in almost every sector, consider tests as something positive. No one wants to wear a bullet-proof vest that was never tested, just like no one wants a software application that was never tested. So, why is there so much poorly tested software out there?

It's because there is a series of problems with the traditional approach of tests, explained here:

- If the tests are not comprehensive enough, errors can be put into production, causing devastating effects.
- Tests are usually made after the code is written. When you have already completed your programming activity, coming back to an earlier version of software can be irksome.
- Usually, tests are written by developers other than those who have written the code. As they cannot comprehend the code in its entirety, comprehension and approach errors can happen while writing the tests.

- If the person who writes the test does it based on documentation or other artifacts in addition to the code, any artifact update can make the test obsolete.
- If tests are not automated, they are not going to be executed often, regularly, or exactly the same way.
- Fixing a problem in a given place can create a problem elsewhere. In this case, if the structure of tests is not comprehensive, it won't detect this new problem.

TDD solves all the following problems:

- The programmer writes the test before the code is written. In this case, the code is based on tests, assuring testability, comprehensiveness, and synchrony between test and code.
- The tests are automated and run frequently.
- When a bug is found, the process allows it to be quickly repaired; this procedure is guaranteed by the tests' comprehensiveness.
- When the code is delivered, the tests are also delivered, making it easy to make future changes and extensions in a simple way.

Applying TDD to Games

Game companies usually don't believe that TDD is possible—and this is a mind-set shared by their developers. Of course, there are great differences between corporate software and game software, but best practices, tests, and everything else in the lean methodology can be applied to games indeed.

As an example of this anti-agile mind-set, here are the comments of Rob Galanakis, from his 2014 article "Agile Game Development Is Hard":

- *"Games are art and art cannot be iterated on like other software."* I know the game industry prefers to compare itself to the movie industry rather than the software industry, and I agree that games are art, but art also can be done in small steps (see Figure 6-2).
- *"Games require too much 'infrastructure' to make anything playable."* Playable for a final user, maybe. But remember that not every delivery has to be made for a final client.
- *"Games want users to spend time, not save time."* Here, the goal would be to make the development team save time; the game itself does not have much impact here.
- *"Games are impossible, or significantly more difficult, to test."* I think that it is already pretty clear that games are and should be tested.
- *"Frequent releases are confusing for games, which are traditionally content-heavy."* A change of culture is always necessary; settling is never good.

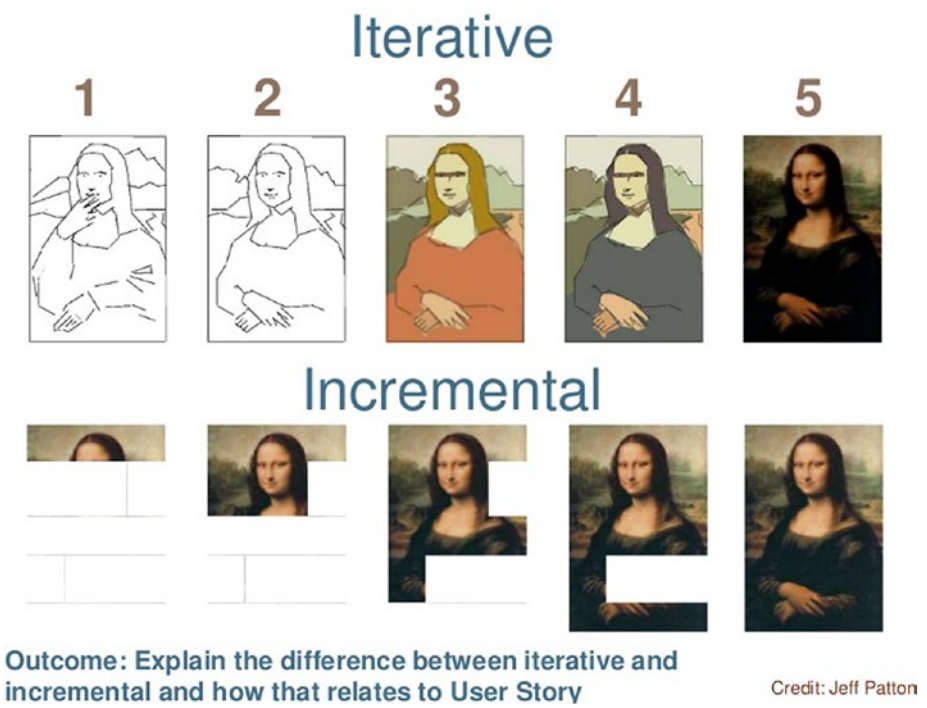

Figure 6-2. *Agile methods applied to art. Source: Jeff Patton (`http://jpattonassociates.com/dont_know_what_i_want`).*

In Appendix B, I talk about some engines and later in this chapter I talk about some ways of applying tests to games. However, I think it's important to present some viable solutions for using TDD now.

- Frameworks such as Pygame (`www.pygame.org/`) and MonoGame (`www.monogame.net/`) are great solutions for using tests, especially because they are always being integrated into new versions of programming languages. Furthermore, every game is made via code, which eliminates the difficulty of testing components.

- Unity was one of the first engines to develop an asset for tests. Unfortunately, it has little maintenance. The Unity Test Tools provide a series of tests, mocks, and component-level tests. It's usually hard to use TDD with this asset but, with a little creativity and effort, you can manage to learn how to do so.
- Unreal is an open source C++ engine. With a little dedication from your development team, it's possible to integrate test libraries.
- Crytek recently showed some interest in allowing integration with its engine, CryEngine. See the lecture entitled *AAA Automated Testing* on its web site.
- Using other engines, the solution can be relatively simple. Create a scene that will contain several tests. With empty prefabs, start to create test conditions. From a test script that you developed, attach the scene and test the specific behavior of your prefabs and scripts.

Prefabs are prefabricated components that allow the reuse of them in other parts of the game. Examples are scene objects, enemies, players who can be instantiated, and so on.

Here are some examples of tests on Unity:

- *Unity Test Tools: Test automation in Unity now and in the future, Unite Boston 2015*: https://youtu.be/wJGUc-EeKyw
- *Unity Test Tools: Test automation in Unity now and in the future, Unite Europe 2015*: https://youtu.be/_OYojVTaqxY

Remember, games usually have complex code and many components beyond code. Therefore, it's important to have in mind the simplest test that you can run and how to make it pass in the simplest way possible.

Note I have faced desperate situations while having to test components from a visual point of view, but the solution was pretty simple: I created a scene that ran automatically, and all components within it started as empty. This scene had a single script to manage tests. As each test was written, the components evolved to the point where I managed to create simple prefabs that were ideal for the applications I needed.

Overcoming the Hurdles

TDD for games presents challenges that most parts of software don't have, such as the vast number of platforms (Mac, Windows, Linux, Xbox, Nintendo, PlayStation, iOS, Android). Thus, you need to run unit tests on each platform that you want your game to run on. Usually, you will keep several builds and check whether they break the tests and in which specific cases this happened.

Another challenge is how to test graphics. It's a best practice in software engineering to keep all the graphic-related code in a single module, which contributes to facilitating tests from other modules. Using engines can make TDD unfeasible since the biggest ones are not TDD friendly—although some of them, such as Unity and Cry, have started to address that.

Randomness is something imperative in games. For instance, it's impossible to predict which walking sound should be played; in addition, running tests for random things is complex. One of the ways to solve randomness on tests is to try to get the most out of the tests that are being tested. Perhaps you can pass it as an argument? There's a clear relation between running tests and reducing bugs.

Making TDD Better

TDD was developed to be used with a series of practices that add value to development. Next, you'll learn about some of them.

Refactoring

Refactoring is the process of making changes in existent and working code, without changing its external behavior. This means changing how the code works but not what it does. The goal is to improve the code's internal structure, making it more readable with better performance. For example, after making the code simpler so the test passes, you can refactor it to clean it, especially reducing duplications.

Or, you make sure refactoring does not break the tests.

Here are times to refactor:

- When there's duplicated code
- When you realize the code is not clear or readable
- When you detect potentially dangerous code

Here are some examples of important types of refactoring:

- *Extracting classes*: When a class is too big or its behavior is not clear anymore, it's worth dividing it in two or more so that similar behaviors stay together.
- *Extracting interfaces*: You can use refactoring to facilitate using mocks because they represent a group behavior.
- *Extracting methods*: Refactoring applies when a method is too long or has complex logic. It facilitates understanding methods.

- *Introducing self-explaining variables*: When you need variables that are not clear regarding their function and why they're there, it's good to extract them for a constant that explains its function.
- *Trying to identify patterns*: You can use refactoring to try to identify patterns in code.

Pair Programming

Why is pair programming included in TDD? Basically, pair programming improves your performance of TDD. Here are some of the benefits:

- *Increasing the concentration of developers*: It improves communication among all parties, especially facilitating the identification and resolutions of gaps, in addition to stimulating creative discussion.
- *Reducing defects*: It helps to make the code simpler and the reviews more efficient.
- *Keeping business continuity*: The increase on communication reduces the impact if a project member leaves. However, you might face some challenges regarding pair programming, such as work infrastructure, since assembling teams can be complicated, especially in small companies. Furthermore, if your team doesn't have identical configurations for everyone, such as IDEs and editors, it could be hard to implement pair programming. Also, it's necessary to take into account fatigue because it's common to find people exhausted after hours of pair programming.

- *Reducing issues related to ego conflicts*: Perhaps one of the greatest difficulties of many pairings is the lack of communication and humility when arguing. It's usual that constructive discussions turn into aggressive arguments, which can be fatal to pair programming. It's important to remember that, as everything is done together, it's indispensable to value the pair and not the individuals. Another thing that can be beneficial for pairing is feedback.

Here are some tips for pair programming proposed by Tarso Aires (2015):

- *Don't centralize the driving*: The member of the pair who feels more comfortable in the development environment tends to centralize the driving. It's good to define beforehand an amount of time that each person is in the driver's seat.
- *Manage the focus together*: Having different focuses is extremely common. If this becomes a problem, the more focused partner must bring focus to the less focused one. It can be complicated for the less focused partner to get back her focus alone. Techniques like pomodoro should help—it consists of taking little breaks for relaxing.
- *Avoid working alone*: Sometimes one of the individuals of the pair may have to be absent for a while; in this case, it's convenient to wait for her to get back. Seize the opportunity to do stuff not related to work.

- *Try to mix concentration and relaxation moments*: Focus is essential, but an excess amount of it can be harmful. Remember that we are not robots and we need to rest. Play video games, have a coffee and a conversation, or even take a 15-minute nap.
- *Celebrate the achievements*: At the end of each stage or activity, it's gratifying and empowering to stop and contemplate what was done. Celebrate!
- *Achieve balance*: It's natural that one person has more experience in the activity to be performed. This can bring a difference in the rhythm in which things are being done, harming the pairing. Notice these differences, adapt to your colleague's rhythm, and explain your ideas however many times you find necessary. Communication is essential.
- *Convey the context properly*: It's important to guarantee that both pair members have a common vocabulary. If necessary, draw diagrams and always try to explain as simply and directly as possible.
- *Know how to handle divergences*: Disagreements happen all the time during pairing. In these moments, it's important to stop and listen to everything the other has to say, answering calmly and respectfully. If necessary, call a third person to help in solving the deadlock.
- *Get ready to learn and to teach*: Keep in mind that you can contribute even if you're new in the project, the same way you can learn new things. Present concepts gradually and guide your partner to the solution subtly and clearly. Listen carefully to whatever is being taught.

- *Exchange feedback*: The pair should have a conversation and exchange feedback while memories are fresh. This could be a 15- to 30-minute conversation or exchanging notes in a proper place.

The most common example in pair programming is two people sharing peripherals and the monitor of a CPU and working together. Another common way is to use a tool such as Screenhero, which allows the remote person to edit and change the development environment through screen sharing.

Note If you want to study more about TDD, check out the references in bibliography.

Summary

In this chapter, you learned about the stages of test-driven development and techniques to improve the team's performance, such as pair programming.

Unfortunately, well-tested code does not always assure full quality; for example, if the code is not in its latest version, tests might be broken, or the build of the game might not be functioning. This is when you can introduce continuous integration, covered in the next chapter.

CHAPTER 7

Continuous Integration

In this chapter, you will see the advantages of continuous integration, understand its key concepts, and learn how to apply it to a project. You'll also understand what continuous integration means for a team. Last, you'll understand which tools and frameworks are available for implementing continuous integration during game development.

Why Continuous Integration?

The main goal of *continuous integration* (CI) is to find bugs as soon as possible. The process goes through the following stages:

- *Bug tracker*: You use a tool to identify bugs.
- *Source control*: You commit the code using some code versioning tool. This tool allows you to verify the current state of the code and all the changes it has gone through. You can use it in association with a build agent.
- *Build agent*: This is a tool that generates the code's build. Usually, it is a set of commands in an environment, such as Jenkins, with the goal of making builds available automatically.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_7

- *Unit tests*: At this point, you can verify unit test coverage with mutation tests, which introduce small errors and variations in the code to determine the test comprehensiveness.
- *Automated functionality tests*: Use this kind of test to check whether all the functions and features are working correctly and whether any are defective.
- *Deploy*: Here you release a build for use, updates, and manual tests, among other reasons.

Ideally, integration should happen at least once a day and, if possible, more than that. The more frequent, the better. Each code integration is verified by an automatic build, which helps you detect problems. Furthermore, this practice helps to identify quickly where a mistake is.

"Continuous integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove." —Martin Fowler

---

If integration occurs frequently, looking for bugs becomes a simpler job, and the developers will have more time to develop new features. In addition, it allows the team members to have more clarity regarding the development time since they can spend less time fixing bugs.

These are some of the advantages of continuous integration:

- There won't be any complex and long integrations since the code is integrated every time and each increment is small.
- It increases the visibility of each stage, allowing for better communication. Everyone can follow how their colleagues are developing and in which stage of development they are.

- It facilitates problem identification, allowing you to solve problems immediately. Any problem will be identified as soon as the code runs the automated build. Less time will be spent on debugging and more with features since errors will be quickly identified, without long hours for investigation.
- Trust increases when building a game with a solid foundation because the tests and the continuous integration will assure that the errors are small and quickly identified.
- It reduces tension and the mystery of waiting to see whether the code works. Although builds take time, the time to fix them is dramatically reduced.

Using Continuous Integration

There is extensive literature on the subject of continuous integration, from books about agile methodologies to books about software engineering. My goal is to simplify all that for you with the following steps:

1. Developers put the code in their workspaces.
    - Use `git clone [repository url]` to add the code to your workspace.
    - Use `git status` to check the status of the code.
2. When changes are ready, you commit them in the single repository, usually in a new branch.
    - Use `git checkout -b new-branch` to change the work branch to `new-branch`.

- Use `git add -p` (using `-p` to review all changes or using . and `--all` to add everything) to add the changes to the commit state.
- Use `git commit -m "Here goes the comment"` to commit the changes with comments.
- Use `git push [remote-name][branch]` (with `-f` to overwrite) to send the changes to the repository.

3. You merge the branch with the main repository.
   - Use `git fetch [remote] [branch]` to search all branches of the repository and download all the required commits.
   - Use `git merge origin/master` to synchronize the changes with the master repository.
   - Instead of the previous commands, you can use `git pull [remote]`.
4. The CI server monitors the repository and searches for changes.
5. The server builds the games and runs the unit and integration tests.
6. The server releases an artifact for testing.
7. The server designates a label for the build of the version that it is creating.
8. The server informs the team when the build completes successfully.
9. If the build or the tests fail, the server warns the team.
10. The team fixes the error.
11. You iterate (and continue doing CI).

More about git Git is a code versioning system that stores all the code in a repository. If one of the files on the machine is not necessary for executing the project, you can use the `.gitignore` command, which will ignore that file when moving all of them from your machine to the repository.

You can find more information about GitHub at https://try.github.io/levels/1/challenges/1.

A Team's Responsibilities Regarding CI

To implement CI, you need all your team members to meet these responsibilities:

- *Keep the code updated*: Every time the code needs to be edited, it's important that the previous changes are already integrated, especially if they are bug fixings.
- *Do not commit broken code*: Broken code breaks the build and makes it a problem instead of a solution.
- *Do not commit nontested code*: If the code is not tested, there is no guarantee that it is working, and if it's not working, you won't have a game working.
- *Do not commit when the build is broken*: When the build is broken, the last commit should be reversed and fixed. If it's not fixed, future modifications will remain broken, and it won't be possible to identify future problems.
- *Do not abandon the build*: Do not abandon the build until the build of the game is ready. Keep an eye on it because if it breaks, it's necessary to take measures.

So, what about the advantages of using CI in game development?

In addition to the many CI tools that are available and compatible with game development engines, the following are advantages that are important from the time-saving perspective:

- Nonprogrammers can test the current version of the game.
- A press release version for promotion is available.
- The publisher can follow and identify the status of the game.
- You can guarantee that the game runs on different platforms.
- It's easy to get the final version of the build.
- It's easy to fix any bugs that appear in the final version.

Code Versioning

Code versioning gives you several benefits because each stage of the code is disposed in a system that allows you to identify the strengths of the versions, revert, and divide the code. The following are some advantages of code versioning:

- *You changed the code and realized it was a mistake*: Just reverse it. If the commit has already gone to build, it's a best practice to revert and redo.
- *You lost the code and the backup version is too old*: Just commit frequently. If commits are made often, you won't have any problems restoring lost code.

- *You need to check the differences in code versions*: Use `git diff`. Many versioning tools allow you to check the differences between the code in progress and the original. Another advantage is that you can check out the modification log, allowing you to identify when a feature was added.
- *You need to identify which change broke the code*: Code versioning facilitates the identification and fixing of faulty changes. You just identify in the CI platform which commit broke the code. Furthermore, this is related to the previous topic because it includes the concept of verifying differences.
- *You can divide the work over the code*: Work division is important in large project so that the development can be done concurrently, not in sequence.
- *It allows you to follow the development of the code*: The visualization of the project progress allows the development team to check the progress of the game. It also allows outside people to check how things are going.
- *It allows you to keep different versions of a product*: Many times it's necessary to keep the build from different platforms. Continuous integration allows you to do that in a simple way. One way to do it is by using branches.

It's a best practice to use comments on commits that identify which card generated the modification and what was done. When a branch is created, it should identify which card the branch belongs to and what it will do.

All these difficulties can be solved with versioning. My preference and suggestion is to implement versioning through Git, such as with GitHub.

For Unity, remember that the steps are similar to other tools. For instance, here are the steps to version your code:

1. Use `gitignore.io` to generate your version, avoiding sending hundreds of unnecessary files to the build. Watch out for 3D models ending in `.obj` because you must exclude `.[Oo]bj` models from this file.
2. In Editor Settings, under Asset Serialization Mode, select Force Text Flag. This prevents the Unity files from being sent as binaries because that may create merge issues on GitHub. It's complex to do a merge of numerical sequences.
3. In Editor Settings, under Version Control Mode, select Visible Meta Files. The meta files are important for the Unity project to locate its parts.

Automated Build

The Unity engine has an internal system that allows for automated builds, called the Unity CloudBuild (https://unity3d.com/learn/tutorials/topics/cloud-build/introduction-unity-cloud-build). Unfortunately, other engines don't have such a strong feature as automated builds, and their build systems used to be somewhat precarious.

For the cases in which the automated build feature is not part of the engine, I recommend using an existing system, such as Jenkins (https://jenkins.io/) or Travis (https://travis-ci.org/). Both Travis and Jenkins are popular and versatile.

In addition, there's TeamCity (https://www.jetbrains.com/teamcity/), which is optimized for C#. For other software, there are other sources, such as Go CI and Snap CI.

Of these, Jenkins is probably the most popular for doing automated builds in games; it's a simple platform to use to implement jobs, and it has an active community.

Most important, its code is built in one single step. Depending on your proficiency level in CI, it's possible to build several platforms in a single automated step.

Important parts of process automation are the uniformity, safety, and the guarantee that all tests are going to be run in the same way. This applies both to unit tests automation, which can be run before the build, and to the functional tests stage. For those who use Unity, I suggest creating unit tests with NUnit and NSubstitute and creating functional and integration tests with Unity Test Tools (https://www.assetstore.unity3d.com/en/#!/content/13802), which, despite rare updates, is a powerful tool. Make your build self-tested and fast.

Summary

In this chapter, you learned about the value of CI, as well as how and where to use it. You also saw how to use CI in games. Most important you learned how CI can be very helpful in a market that typically presents a lot of bugs. With this information in hand, you can start to think about designing and building a game.

CHAPTER 8

The World Between Design and Build

In Chapter 2 I talked about the first steps of using lean and briefly mentioned design and build. In this chapter, the goal now is to explain in depth these concepts and how to use them in lean game development.

Generally, *design* is responsible for predicting what the software is going to be (from the artistic and visual parts to the user experience), and *build* refers to how you are going to develop the idealized design and what is and isn't possible to do.

To help distinguish design from build, you can think of it like this: "Design makes the recipe, and build prepares the meal." It's about the difference between planning and doing. Design's concern is with utility, and build's concern is with the technical specifications and tools.

A Little Bit of Design

Formulating the game design involves a series of iterations and can also be a moment to reflect on whether it's worth continuing the project. However, this is not its main function.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_8

An important task during design is to search for information about the needs and the utility of the project. Many times, it's necessary to ask random people what they think about the project and/or create board games that try to simulate, as precisely as possible, the mechanics and the features of the game to be developed. This allows you to assure that the game is viable and has potential.

An efficient design allows for a minimum waste of resources and time. Design is the step in which the project's elements will be prioritized and the team will try to eliminate strict specifications because they are costly to modify. Hence, everything that can be decided on in another iteration must be left for the next iteration. Remember that leaving something for later doesn't mean you will not do it; it means that you should not get stressed now about something that shouldn't be solved yet, that cannot be solved, that is blocked for any reason, and so on.

For the design to be a success, it's necessary to think of many elements of the game, including the following: the game set, the characters, the features and mechanics that will be in the game, the kind of experience you want users to have with it, the plot, and its gameplay. Think about a *Castlevania*-style scene in which a character is playing piano: is it worth it to make a super-complete animation or something minimally realistic about the character playing the piano? You make important decisions in the design stage that can affect the build stage.

A Little Bit of Build

What about the build? The *build* is the step in which all information solidified in the design stage is used to start thinking about the development process. You must remember that, in games, developing also includes creating artwork and sound effects. If these elements were decided on during design, they are not something you should worry about at this time.

If the solutions suggested in the build stage are not good, you will realize in the next iteration because the concepts need to be developed in the ideation, hypotheses, and design phases. That is, during the build, you can discuss some style and development specifications, and, if needed, you can step backward. It's important that development keeps the pace of the code and is continuously integrated because it's a creative process that changes over time in which feedback is important.

For instance, take a game that has a scene in which the character turns his back and starts to play the piano. Say the artist got excited and animated the whole scene of the character playing the piano, but the fingers are never seen. That would be easily prevented if the artist would have used prototyping and continuous delivery; also a lot of memory would have been spared. In the build, you can define which unit tests are going to be done, clarify how are the functional tests going to be developed, and observe whether some programming pattern emerges. You basically specify how all of these things are going to be delivered, automated, tested, and integrated.

Pretty Beautiful, But How Is It Done?

To start the build, you have the MVG0 iteration, which is a board game. This is when you see if the mechanics and the features of the game are interesting, fun, and coherent. The game doesn't need to be perfect, or even close. Creating a board game usually demands a lot of creativity, but it gives all the team members a chance to visualize what you want as a product.

The board game I conceived has a board divided into a grid. The bottom is made up of noninteractive spaces (the ground) and spaces that can lead the player to death (holes). The enemies' behavior is generated by a random value (given by dice) at each turn.

The main character can move freely accordingly to the dice, but a jump costs one extra dice unit. This is the same as in *Mario* when you determine that killing enemies implies falling over them; so, if the character stands over the enemy, he dies.

Enemies can move from -3 to 3 positions horizontally. To win the game, three lives are enough.

Creating the board game requires several tiny cycles that you can do in a lean way and finish quickly, with no great artistic complications. Figure 8-1 shows the MVG0 prototype; I can assure you it was fun to build and provided plenty of game hours.

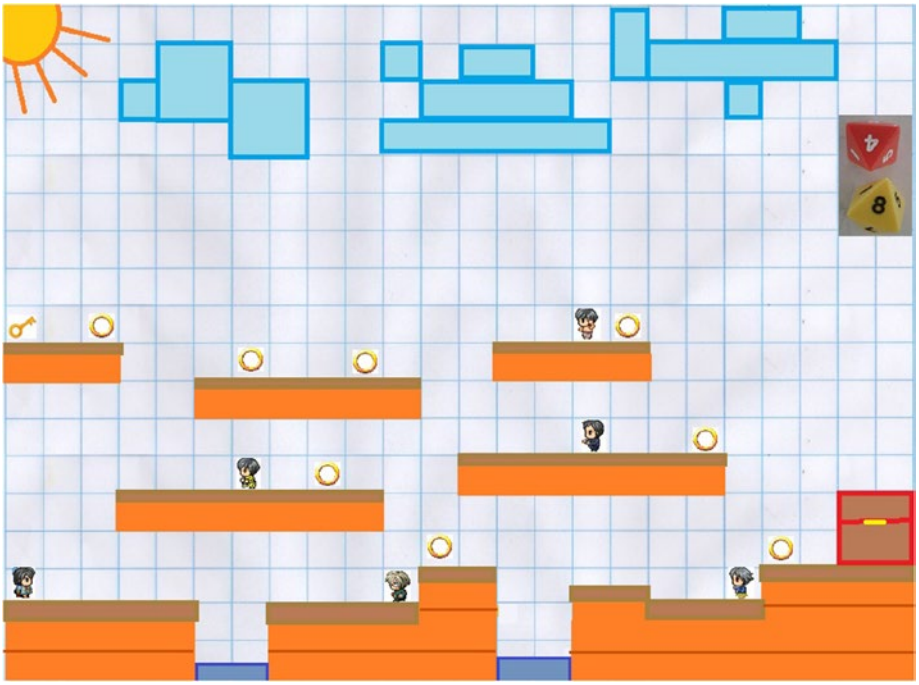

Figure 8-1. *Board game prototype*

The board game model had eight prototypes that formed four MVGs, and each iteration corresponded to a prototype. So, I had to do at least eight iterations. There was the initial hypotheses, but I'll have to devise ones for the next iterations, with new ideas.

In the first stage of design, you define the narrative of the first MVG so that the PO can guide the game vision and indicate how you want the mechanics of prototype to look. Figures 8-2 through 8-5 represent the idealizations of how mechanics work in the example and some conceptual artwork for MVG1's prototypes.

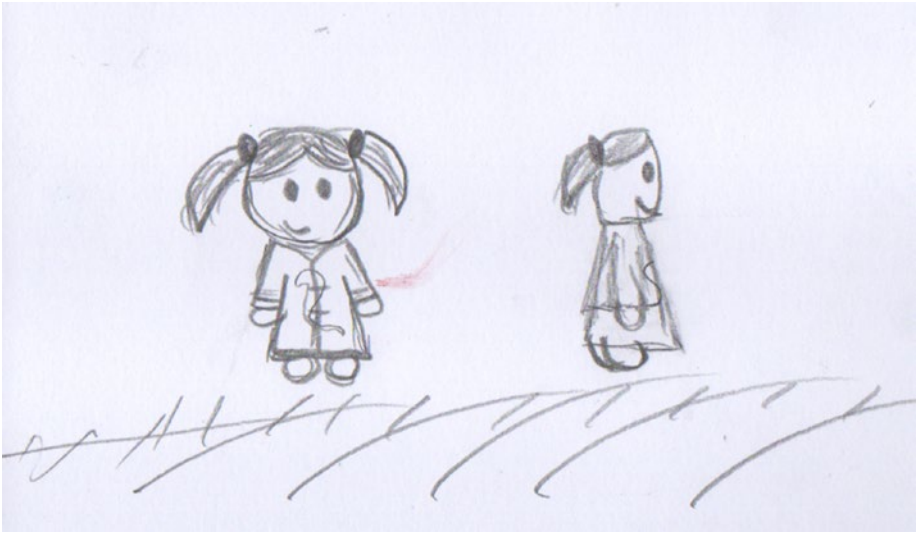

Figure 8-2. *Original sketch of character (artwork by Diego Kim)*

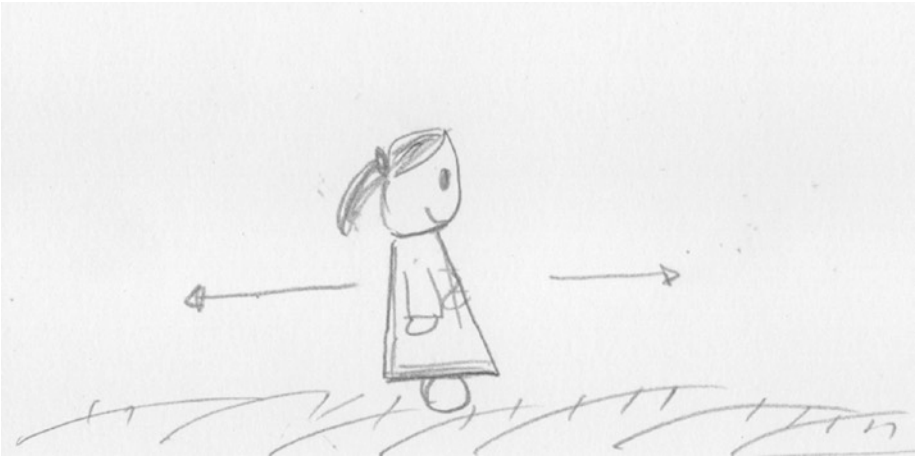

Figure 8-3. *Original sketch of the movement mechanics*

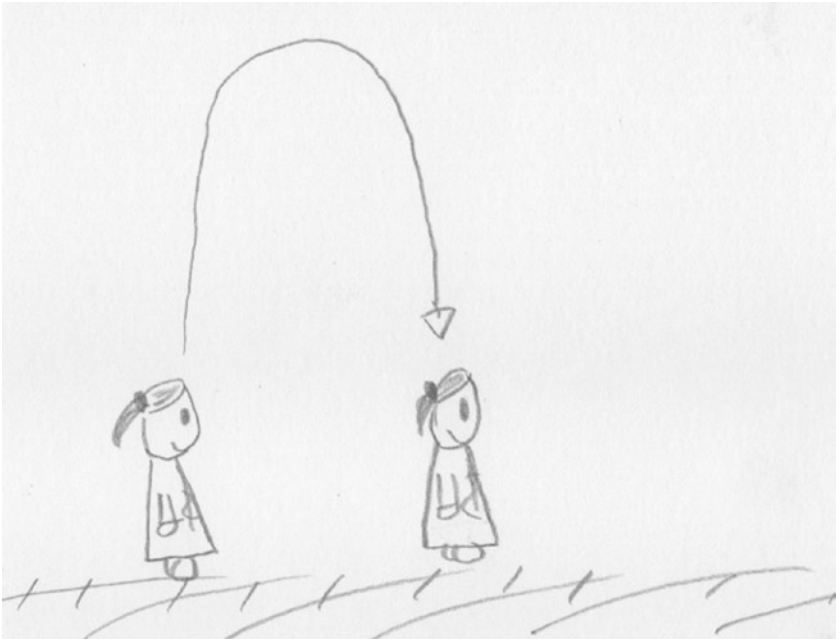

Figure 8-4. *Original sketch of the jump mechanics*

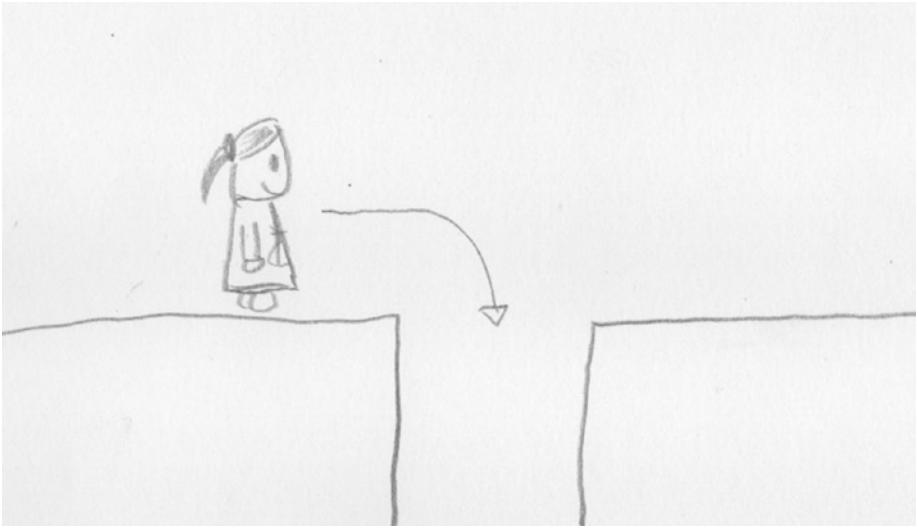

Figure 8-5. *Original sketch of the fall mechanics*

Figure 8-6 shows the pixel art resulting from the sketches.

Figure 8-6. *Character sprite sheet (artwork by Diego Kim)*

What about the build? How are you going to do the tests? What tools are going to be used? How are animations going to be made? And the sprites? How much time do you project this iteration to take? How is everything going to be integrated? How will you deliver? Who are you going to pair up? Where will you do the code versioning? Is what you have is still valid? Does it work? Is it worth migrating? These are the questions that Figures 8-7 and 8-9 answer.

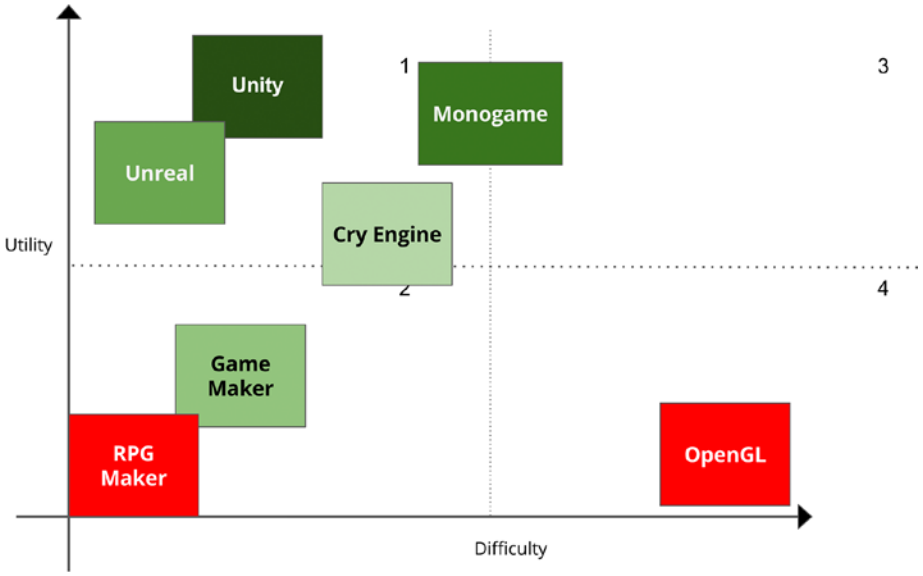

Figure 8-7. *Game development tools*

Tests are explained in the tables of Chapter 9. Figure 8-7 shows the tool preferences for game development: the dark green color indicates the favorite tool, the light green indicates the least favorite, and the red one means tools to be excluded. Figure 8-8 shows preferences for integration tools. Also, I decided to use the pixel tool to generate the sprites shown in Figure 8-6 and animate via Unity.

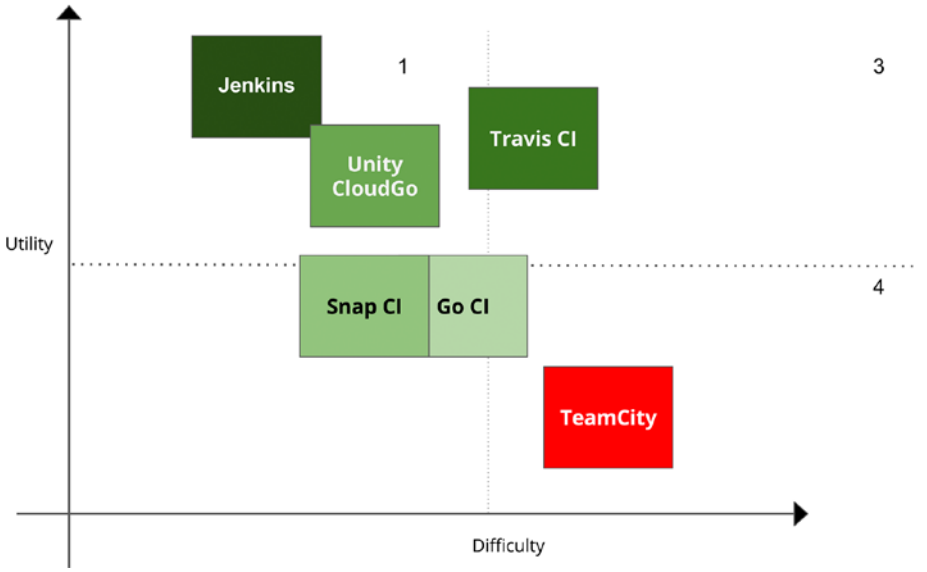

Figure 8-8. *Integration tools*

Figure 8-9 represents the idealization of the possible stages of continuous delivery. The code versioning was done via GitHub. I didn't have any valid existing code for this game; however, I used some old experimental code.

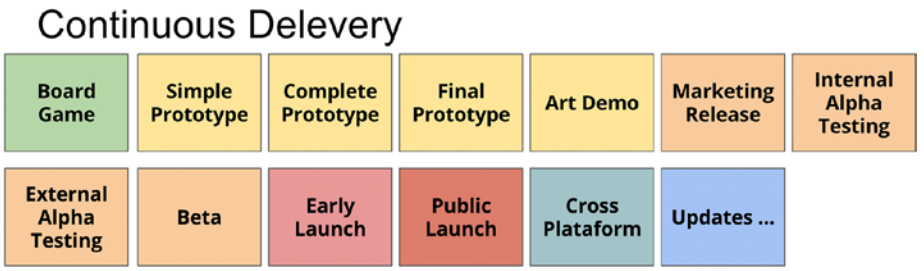

Figure 8-9. *Continuous delivery sequencer for the game*

Sprites Sprites are 2D layouts used to create characters and game environments; usually, they represent 2D objects. The most popular sprites nowadays are made with pixel art. A set of sprites to generate different animations is called a *sprite sheet* (Figure 8-10).

Figure 8-10. *Example of sprite sheet. Source:* `https://www.codeandweb.com/texturepacker/tutorials/how-to-create-a-sprite-sheet`.

A good sprite editor is Piskel (Figure 8-11). It allows you to generate sprites pixel by pixel, with many colors and tools (`www.piskelapp.com/`).

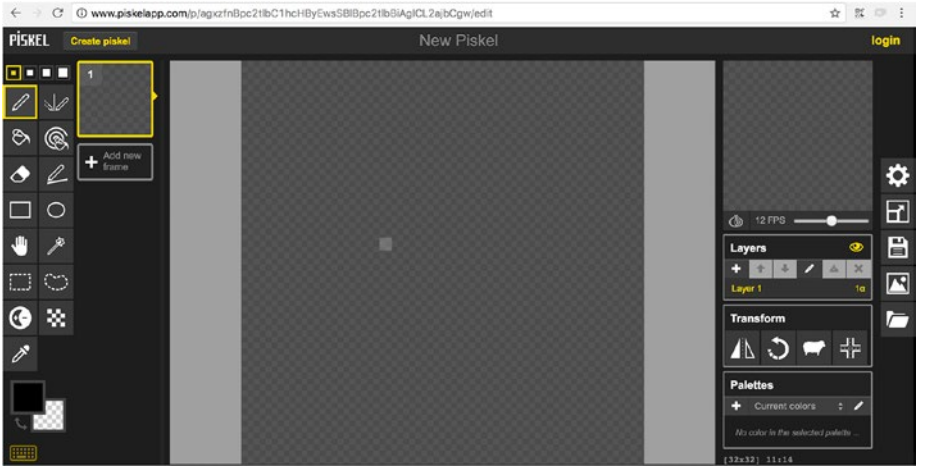

Figure 8-11. *Sprite editor Piskel*

Summary

Now you know the fundamental differences between design and build. You saw the expectations for each stage and one interesting example of concept usability. With everything defined, you can now move on to the most pleasurable part: coding and testing.

CHAPTER 9

Test, Code, Test

One of the most important steps for a game is the code and how to test it. Although a game is artistic software, it still needs to be coded and tested. Therefore, this chapter will help you understand what kinds of tests can be performed, how to elaborate on test cases, and how to automate this process for code and art.

What is the best way to start this process?

Game design is done, but how do you turn game design into a game with the least number of possible bugs? Here you can use software development techniques such as test-driven development (TDD) and extreme programming (XP). It's not absolutely necessary to follow the development techniques mentioned, and adaptations are always welcome, but it's important to use them as guidance in development. One of the most important lessons you can get from them is the fact that testing is important. That's why you need to test from the start. Let's look at this now.

Testing Types

When you think of development, a series of questions probably arise regarding testing. For this specific feature, how will you do tests? What is the simplest test? What will be the step-by-step process for the test? What about test automation? How will that be done?

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_9

All those questions must be asked and planned in the build stage so that the coding and testing stage goes as well as possible. The following are some types of tests:

- *Unit test*: This type of test checks one unit, one feature, or one routine developed by the programmer. A good test would be to assure that, for example, by pressing the X button, the character will receive a jump command. These are the classic assert tests in TDD examples.
- *Functional test*: This test assures that the functionality of a class or namespace is satisfied.
- *Component test*: This type of test runs a class, a package, or a small software program that involves the work of many programmers or a team of programmers. A good example is when a state machine is tested in several ways.
- *Integration test*: This is a test between two or more classes, namespaces, packages, components, or systems. It starts when there are two different classes/namespaces interacting. A good example would is when Mario falls over a Koopa to guarantee that the Koopa will become a shell and "roll away."
- *Regression test*: This is a continuous integration test to look for defects not found earlier with different versions. It's a sort of contract test that looks to assure that a great set of features and its integrations will work no matter what the version is. In games it can assure side effects are controlled.

- *System test (game)*: This test runs the software in its final configuration. It tests all the possible solutions and interactions. It's something like an end-to-end test, in which you create a bot that allows you to test all possible game features (a technique used a lot to create gameplay).

Test Cases

Let's begin with a simple example that does not apply specifically to the game market but that most developers have to face at some point. Imagine that your team has to develop a feature for a company that uses a bank system. The given feature is the validation of the credit card number to assure that the number entered is valid. Table 9-1 presents a case.

Table 9-1. *Test Cases for Features*

Feature	Test the Validity of a Credit Card
Input	A string representing the credit card number and two numbers to represent the month and year of the expiration date.
Tests	1. Check whether all the bytes in the string are digits.
	2. Check whether the month number is between 1 and 12.
	3. Check whether the year is later than the current year.
	4. Check whether the four first digits of the card are from a valid issuer.
	5. Check whether the system is valid in an external system.
Output	OK, or error with message indication that the card is not valid.

The following are the test strategies recommended for developers:

- Test each requirement or feature, ensuring that the entire codebase is tested and ensuring the quality of the code. In addition, tests must be established in the build stage so they favor development and enable what must be done during development. Extra tests are also welcomed.
- Start with base tests, which are fundamental units for the code's operation. Then move on to more detailed and specific tests that represent more complex components and features.
- Each line of logic code must be tested. This is the only way to assure the continuous integrity throughout the game's development; it's also a great way to create documentation.
- List all the errors in the project to monitor your performance and improve your technique. This is a way to accomplish personal development.
- Write the tests before starting to code. Besides being a best practice in programming, this technique almost guarantees that you don't write more code than necessary.
- Look to make clean tests. Summing up, this consists of keeping the test and the code comprehensible and simple—to a level of KISS.
- Review your work.

Keep It Simple, Stupid (KISS) is a general principle that values the simplicity of the project and recommends that all unnecessary complexity is discarded (https://en.wikipedia.org/wiki/KISS_principle).

Coding Game Artwork

Coding can involve much more than doing the actual coding to make a software application work. Coding can also mean creating because the artwork needs to be developed at this time and will also be tested. Because the artwork goes through a creative process that cannot be test-driven, it's a little more difficult to understand how to apply the TDD principles to artistic areas.

In addition, many times the artistic creation process is something very subjective, so different people will respond in different ways to different stimuli. A solution for this is to focus on the smallest artistic units possible.

Let's look at the *Jujuba* example. To test the game's code, you don't need great art; you just need a set of blocks and other geometrical shapes that can give you the feeling that you are playing with a character.

Another important point is that the creative process shouldn't be interrupted. Although I agree that artistic inspiration shouldn't be interrupted, many games have suffered from artistic block. An example is *FEZ*, an indie game developed by Phil Fish. The point that I am emphasizing here is that not every step of artistic creation must be free in all senses.

In this case, you already know who you want to reach because the audience was set with the definition of the personas. Furthermore, you must choose your priorities, which is a great way to allow your brain to think about what you want and what you can do with what you conceived.

So, what about an artistic design driven by tests? First, you should remember that you have a clear separation of priorities in the prototype; thus, you shouldn't be worried with what you still haven't done. At the first stage, you should simply worry with the prototyped shape of what you want.

Because the first prototype tends to be shorter than others, perhaps you can simply make a humanoid with long hair and basic. clothes just covering the body? You don't have to attend to details or think of what would be the best way to portray a character; just provide yourself with a good starting point.

Do you want to test this prototype? As art is subjective, you can test by asking for feedback and explaining that it's a prototype. List the similar feedback and check whether the information matches your priorities. If it doesn't, maybe you'll use a parking lot (explained in Appendix A). If it does, you can apply the necessary changes, and from now on, you'll have a harmonious, elegant, artistic, and tested prototype.

What about using automated tests on your artwork? Unfortunately, our neural networks aren't capable yet of doing these kind of tests. Who knows, in the near future, maybe Google's deep minds will be pioneers in this field.

Coding the Game Software

You should now have clear goals of what should be coded first: your prototype. To do that, you can use the following tables as a guide for how every step should be done. Remember, in the build stage, you decided which tests were going to be made for each one of the features you were creating for the first prototype.

Although I have already defined the stages of prototypes, they can be broken into others and restructured using the feedback and ideation process. The features for the prototypes are walk, jump, and fall, as listed in Tables 9-2 through 9-4.

Table 9-2. *Developing the Walking Feature*

Feature	Character Should Walk Toward the X-Axis
Input	Keyboard inputs.
Tests	1. Check whether a specific key is being pressed. 2. Check whether the direction keys (left and right) are being pressed. 3. Check that when the left direction key is pressed, the character moves to the left. 4. Check that when the right direction key is pressed, the character moves to the right. 5. Check whether the other keys, when pressed, don't move the character (in many cases, you can exclude the keys ASWD). 6. Check whether the character collides with the end of scenery.
Output	If the new position is as expected, the test passes; otherwise, it fails.

Table 9-3. *Development of Walking Feature*

Feature	Character Should Jump
Input	Keyboard inputs.
Tests:	1. Check whether the spacebar is being pressed for jumping.
	2. Check that when the key is pressed, the character moves up in one position.
	3. Check whether the character moves softly from the bottom to the top (check for two or more points in the first part of the jump).
	4. Check whether the character moves softly from the top to the bottom (a complete jump).
	5. Check that when a direction key is pressed, the jump is in the same direction.
	6. Check whether the jump makes a parabola (five points at least).
Output	If the new position is as expected, the test passes; otherwise, it fails.

Table 9-4. *Development of Falling Feature*

Feature	Character Should Fall When There Are Holes in the Scenery
Input	Character collides with the scenery.
Tests	1. Check whether the character collides with parts of the ground.
	2. Check whether the character collides with air blocks in the scenery.
	3. Check whether the character falls when there is a hole in the ground.
	4. Check whether the character falls by making a horizontal launch affected by gravity from the ground.
	5. Check whether the character falls by making a horizontal launch affected by gravity from an air block.
Output	If the new position or new state is as expected, the test passes; otherwise, it fails.

The examples presented in the tables correspond to the tests created for the first prototype of the *Super Jujuba* game.

Each programmer can idealize differently how these tests are going to be made, but the important point is to keep the coding simple and the solution even simpler. If a step is too vague or obscure, conceiving of and running an intermediate test can be a good solution. In addition, any deeper modification, such as refactoring and class extraction, can become much less laborious after error fixing via tests.

In software development, it's common to update application versions, including games, after release. Identifying broken tests and differences between what was expected and what was found can be a great way to optimize the code, with no need to review the whole logic. Furthermore, it's much simpler to understand what the code should do when tests document what should happen. Always start with the simplest and most fundamental test to guarantee that the minimum planned task is happening.

Test Automation

Unit tests developed by the programmer are a great source for starting automation because they are fundamental for checking whether the features are working properly. Therefore, it's important to automate the process in which this is done.

A *test runner* is a tool that allows you to run a sequence of all kinds of tests from a command line. In the case of Unity, the test runner provides integration between tests and the visual output, allowing to the programmer to verify what is going on. Figures 9-1 and 9-2 show a visual

test runner and an example of the commands you can run. The following are some test automation components you should know:

- *Test manager*: This component manages the operation of the software tests. Also, it keeps a record of the test data and expected results.
- *Test data generator*: This component generates data that must be tested by the software. It can use predefined patterns or random data required for specific stages of the game.
- *Expected data generator*: This component's main function is to generate all possible data for an automated test, such as end-to-end data.
- *Results comparator*: This component's main function is to compare new results and old ones.
- *Report generator*: This component generates reports on test results.
- *Dynamic analyst*: This component sees and counts function calls made by the code.
- *Simulator*: This component simulates user interactions with different types of systems, such as testing different platforms.

```
>Unity.exe -runEditorTests
-projectPath PATH_TO_YOUR_PROJECT
-editorTestsResultFile C:\temp\results.xml
-editorTestsFilter Player
```

Figure 9-1. *Test runner from the command line. Source: https://docs.unity3d.com/Manual/testing-editortestsrunner.html.*

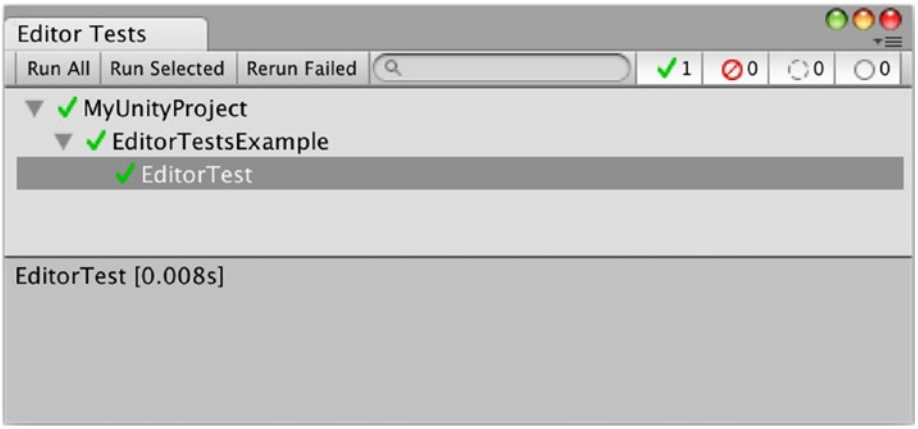

Figure 9-2. *Command-line test runner for Unity. Source:* https://docs.unity3d.com/Manual/testing-editortestsrunner.html.

Summary

In this chapter, you learned what to do before coding, namely, defining the tests and the development policies. In addition, you saw some great tools so you can use to implement best practices in tests. Now you need to focus on getting what you have already integrated by learning about metrics.

CHAPTER 10

Measuring and Analyzing

I have talked a lot about how to iterate and that you need ways to measure and analyze your results. Therefore, the first step to measuring the results obtained by your iteration is via feedback in all areas (from the team, users, market, clients, and media).

But what form does the feedback take, and how can you measure it? In addition, how should you analyze it? And what should you do with your analysis? The goal is, in fact, to maximize learning through iterations, thus developing with more quality and safety at each iteration.

Feedback

Feedback can happen in several ways, but the main ways, especially in games, are about reacting to products and about evaluating the performance of one individual on a task. These types of feedback aim to improve the product and to improve a person's performance, respectively.

Feedback can be given continuously or when relevant situations take place (I do mostly this kind of feedback). It's important to keep a culture where feedback is expected and valued because its goal is to improve personal development or product quality. For that, it's necessary to always be seeking feedback, whether among clients and users or co-workers.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_10

One of the ways to get product feedback is through Twitter accounts. A common example in the game development community is to hold several active accounts, in which people post photos and comment on new releases and, at the same time, get feedback from clients and users. Another way is to encourage the whole team to actively participate in gathering feedback during the game development, whether from the PO, from business and marketing analysts, or from other developers.

As for the team members, it's important to keep the feedback polite and honest. Try not to discuss personal matters, and use common sense. When taken seriously, feedback allows people to improve their performance and fix minor errors that we all make.

More on Feedback

The following tips about feedback were taken from Felipe de Morais' blog (https://medium.com/@felipedemoraes) with permission and translated from Portuguese.

What's Feedback?

Feedback is an important tool to understand how people around you perceive you, which behaviors you can improve upon, and which ones you should keep and even emphasize.

Feedback is a suggestion to reinforce or improve a certain point.

Good feedback is based on three main points: the data noticed about a fact, the story you built around it, and the argument you had to get to the conclusion. Let's see an example to make things clearer:

Felipe Says: "In the update you gave in the meeting today, I think you were not clear enough. It seems you were distracted."

Julia Says: "Thank you, Felipe. I hadn't noticed that. I'll be more careful."

How to Give Feedback

The following are some important points to consider before giving feedback to a particular person:

- *Start the conversation*: Look for a calm place to talk, such as a coffee house. Avoid unnecessary tension.
- *Ideal environment*: it's important to have the discussion alone. More people participating in the conversation than necessary can generate tension among them.
- *Set the objectives of the conversation*: Be clear why you are having this conversation and why you believe it will help the other person.
- *Describe the feedback clearly*: It's always good to remind the person about the situation that generated the feedback and explain the conclusion you came to in the moment. This is the moment to talk about what you need, and remember to always tell how the feedback could help the other person.

- *Validate the feedback*: Check whether the person agrees, and open some room for questioning. There's no need for others to agree on what was said, though.
- *Avoid bad vibes*: It's time to have empathy and sense how the person is reacting to the feedback. If the person takes it badly, remind them that your goal was to help them.
- *Give the feedback sooner rather than later*: The longer it's been since the incident that you're discussing took place, the smaller the chance of the person remembering what happened, and thus the feedback will have less impact.

Other Ways of Measuring

A common way to get feedback from companies specializing in the game business is to allow those employees to play a demo. These people, besides having a positive publicity impact, can give recommendations and feedback. What you do with this information is what makes the difference in your game.

It's important to remember that, during all stages of development, it's necessary to measure how the product is doing. Therefore, you need to look for ways to get information about quality; merely remembering the developer's vision of how the game should be is usually not enough.

- *Have empathy*: Many times, you are going to have conflicts between the vision of the PO and the vision of the team. In these moments, it's necessary to understand why each part has divergences in their visions.

- *Collect opinions and ideas from team members*: Although they are developers, team members usually play games. Their insights and suggestions can be positive for the development of the game.
- *Offer testing sessions of demos*: Many times people wait for hours in line to test demos. A good way to measure your game is to release demos at events and analyze how people react to the game, in addition to collecting feedback.
- *Validate hypotheses*: You'll learn see more about this later in the chapter.
- *Analytics*: There are several ways to collect data. Currently, using this data to improve the game experience is a great way to measure the game. In this case, I have two examples:
    - *Call of Duty*: *Black Ops 2* was a game in which I saw a lot of modifications on gun performance and creation of new weapons with better balance of features after the initial release. The developers decreased the damage that the Fal gun did and created a new weapon on DLC (extra content for download).
    - Another game that comes to my mind is Rockhead's *Starlit*. It's a great example because it managed to keep my son playing for hours. In this game, the Rockhead team observed the average performance of players in certain regions of the game. When it was easy, the team would make the game more difficult; when it was difficult, the team would facilitate or improve the game experience. In my opinion, this is one of the best ways of measuring.

- *Analytics* is the discovery, interpretation, and communication of significant data patterns. It can be used to describe, predict, and improve the development of businesses and games. For game development, it is a powerful tool to understand the behavior of players, providing valuable insights about them and the game (Figure 10-1). You may find the following sites helpful:
  - *Game Analytics*: `https://github.com/GameAnalytics`
  - *Unity*: `https://www.assetstore.unity3d.com/en/#!/content/6755`

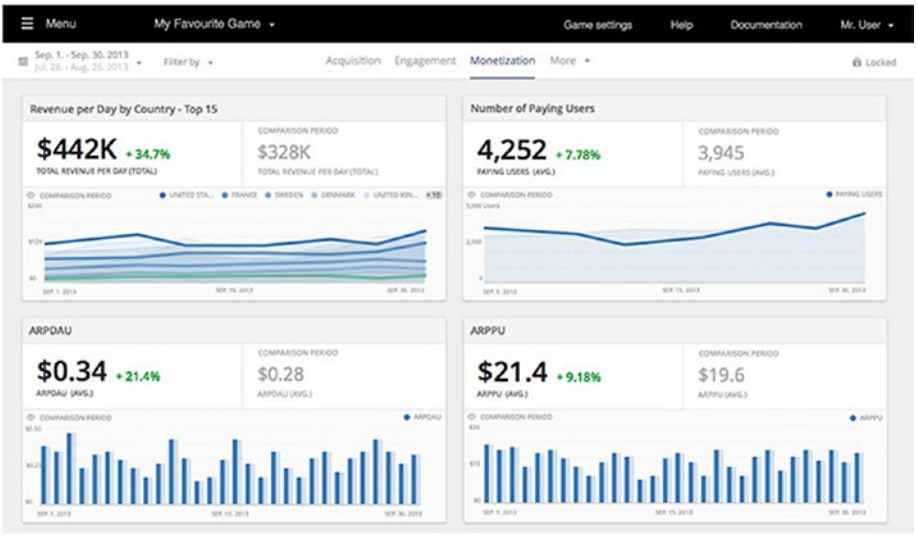

Figure 10-1. *Example of game analytics*

Measuring Through Hypotheses

Some chapters ago, we talked about hypotheses and how they can help you measure and test your minimum viable games (MVGs) and prototypes. However, up to now, you are not using your hypotheses to get you results. Results go way beyond simple data, because data can be collected partially; you need to be searching out good data and avoid negative data, which will affect the data's results.

You need ways to measure your hypotheses so that they give you information that generates knowledge, whether for good or bad, to provide you with future insights. In fact, you will turn your ideas and insights into hypotheses to be evaluated in the future.

Basically, hypotheses are for evaluating results since they're the ideas that will be tested and can be modified only in the next iteration, which will occur after analysis and learning. Using all the information and feedback collection you did, you can test hypotheses without selecting data and with keeping your judgment as neutral as possible.

I think this is an important moment to remind you that the goal of game companies is not to make money but to provide great experience and entertainment. So, if your hypotheses are based on financial metrics, there's a big chance your game will fail or will not be innovative and fun. A result of evaluating hypotheses is knowing your clients and knowing the possible impact the game has on the market. Validating hypotheses is dealing with risks because it much easier to develop when you think that everything is right than to develop while observing the fails. The hypothesis model so far is the one described as follows:

We believe that by:

building the feature *[name of the feature/prototype]*

for the *[intended audience]*,

we will achieve *[expected results]*.

We know that we are successful when we have *[the expected market sign]*.

Well, you know the audience you want to impact, the outcome you want to get with the MVG or with the prototype, and the metric you set as an expected sign of this success. With this information, you can determine whether your target audience has given you the expected sign of your outcome through the feedback and analytics, for instance. In addition, sales data and downloads can contribute to the feedback. However, they tend to be cloudier regarding the target audience since many users are not used to making their personal data public or they might even use false data. Comments on the game page or on Twitter accounts can also be a powerful tool to validate hypotheses. But, with these measures, whether they are positive, negative, complex, or simple, how can you analyze your metrics and hypotheses?

Analyzing

When analyzing, you need to remain impartial. The more information and results that are available from the metric validation, the more important and relevant the analysis will be. While the importance of analysis increases with the quality and quantity of data, the complexity and the advantages increase as well.

Therefore, when you analyze, you must classify the types of feedback, data, and metrics you used to develop your hypotheses so you can keep adding to those categories in the future. The measures resulting from metrics, from MVGs and prototypes to features and to mechanics, allow you to identify points that need to improve.

After this stage, you can separate the categories by relevance or impact on the game development. This allows you to analyze how much revision is needed and where you need to improve the product/game. It's also important to remember that much of the feedback is disconnected from the necessities of the game or the team. The feedback can be useful many times but not at the moment. Thus, it's worth it to leave the feedback in the parking lot (see Appendix A) to be used later.

During the analysis, it's important to compare the feedback with the stories and personas generated previously, in addition to being clear on what information you want to get from every group of analysis. The metrics of hypothesis signs allow you to verify if the MVG/prototype, the feature, or the mechanic is adequate, improving the game for the next stage and generating a continuous improvement process.

Measuring Your Hypotheses of the Female Audience Reach

Let's look at the example from Chapter 5:

"We believe that by building a strong nonstereotypical female character, with interesting functionalities, we will be able to reach the female audience, who will get excited by our game. We know that we are successful when lots of girls start to comment in our feed."

This hypothesis is way beyond mechanics and features. It introduces aesthetic concepts of the game and an audience usually left aside, especially in Japanese games. Is promoting the character and the game at the same time a good way to be successful? You can verify this is working if when the newly released game does not have many downloads from women, but later, after a few speeches, videos, and tweets about it, you have a legion of fans.

This would be a proper metric if that was your intention with the hypothesis; however, you are seeking to hear direct feedback and generate discussions. If your feedback and discussions don't appear in the metrics, you may have to rethink what you got with the hypothesis.

- The character was empathic with the female audience but not enough to engage manifestations.
- You are not available to get feedback.
- There was a failure in communicating the intention.
- You didn't reach your audience.
- The kind of game was not adequate to your audience (I think that every game category can be adaptable to all audiences, though).
- The audience loved it, and the game is a success! (This never happens!)

You must analyze every feedback comment seriously, whether it represents 2 percent or 100 percent of the total. However, you have to prioritize the most common and the most relevant from the team's point of view and, as much as possible, try to analyze the other feedback.

Measuring Your Hypotheses on Basic Features

> "We believe that building the features of walking, jumping, and falling into the game will result in a game with strong basic mechanics. We know that we are successful when we have positive feedback from manual tests and from third parties through the demo we will have."

This hypothesis is a little bit deceiving because you know that many games are based on this mechanics and can be successful unexpectedly. Of course, game testers will focus on features from the technical point of view, so, as much as they have fun, their goal is to see whether the game works. This is relevant information because if from the testers' point of view, the game is not working, you should take action regarding it.

Now imagine sharing a game demo with random people. Would this give you more benefits? If so, a good way to do this is to send demos to YouTubers to make reviews, even if the demos have different names. Choose a serious YouTuber, and the person can review the game from an entertainment and quality point of view. With these situations, you can come up with the following analysis:

- The features of the game are bad and must be fixed.
- The playability of the game is bad, and the project must be rethought.
- The features are OK, but the game is not fun.
- The game is fun but very hard to play.
- The critiques were great!

Thinking like this, you can get a set of possible signs on your hypotheses that allow you to go to the next stage with clear measures such as feedback on game mechanics and features and analytics on the player

performance through different sections of the game. These measures allow you to analyze strong and weak points of features to be validated by hypotheses and allow you to think through problems, solutions, and points of action. With points of action in mind, you can go to the next stage: ideations over action to be taken.

Summary

At the beginning of this chapter, you learned about the core concepts of feedback, mainly understanding how to give feedback. Feedback is important in lean development because it can be the core measure to iterate on your project. Also, you saw some different ways to measure game results, especially analytics tools. You saw how to validate and get results from hypotheses. Lastly, you saw some examples of analyzing with the hypotheses developed in previous chapters.

CHAPTER 11

Creating Ideas for Iterating

Before iterating, it's necessary to come up with ideas about what you need to improve for the next stage and which feedback can be used to improve the game comprehensively. After measuring and analyzing, you need to use all the generated information, such as action items. This stage is called *ideation*, which is similar to inception but much shorter and focused. Also, in the ideation stage, you have much more information about the game because it is already being developed. The stage can validate the current hypotheses and generate ground-breaking ideas for developing the game. Ideation can also generate new hypotheses and make it possible to rethink the design and build steps in a way that the game development process will be quicker and better.

Action Items

Each action items needs an explanatory title and a description of what it is about to put it into context. Furthermore, you need to present what generated the item, including the involved feature, the base hypotheses, and the identified sign. To conclude the action item, it's interesting to have a brief description of the reason for creating it, and even possibly assigning someone to do it.

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3_11

During the process of generating ideas, it may be useful to bring up the inception concept. However, this time it can be shorter and more focused because you already have the minimum viable products (MVPs). At this point, ideation works for improving the concepts of MVPs, as well as generating new hypotheses and perfecting all the concepts of the game in general.

The ideation process consists of brainstorming with the previous data and generating adaptation propositions for MVPs, without losing the focus and the vision of the game. It's important to keep your judgments impartial at this point and to think "out of the box," perhaps using techniques that stimulate creativity. The ideation stage is one of the best ways to find solutions for the problems identified in the measurement and analysis stages. During this stage, the recommendation is to have the most diverse profiles because that's the only way to gather a proper variety of ideas.

To organize the suggestions, you can use an *ideas menu*, which is a synthesis of the ideas generated so far, with the goal of ordering the insights and making them visible and comprehensible to everybody. A good way to deal with the problem is to use a *sketching session*, which is a way to get the team more involved in design decisions to create a deeper understanding and develop more consideration of the end users; it also helps to define the problem and propose improvements for features.

SKETCHING SESSION

The *[game/feature]* is *[what is intended]*.

So that *[players/users]* can *[goals of users]*.

We note that *[current problem/limitation of the current system]*, which makes *[reason of being a problem]*.

How can you *[what to solve]* so that *[benefit of solving]*?

Example of a Sketching Session

In the previous chapter, the following hypothesis was used to generate metrics:

> *"We believe that building the features of walking, jumping, and falling into the game will result in a game with strong basic mechanics. We know that we are successful when we have positive feedback from manual tests and from third parties through the demo we will have."*

We described a series of possible analyses, but let's focus on two of them in a sketching session.

- The features are OK, but the game is not fun.
- The game is fun but very difficult to play.

The Features Are OK, but the Game Is Not Fun

The game is simple and focuses on movement, demanding attention regarding the position, so the players can have a new experience in a 2D game.

You learned from the feedback that, currently, the playability and mechanics work well, but the game is not fun, so the players aren't overly interested in the game and won't promote it. How can you make the experience more fun so that players have fun and promote your game?

ACTION ITEM

Title: Improving the game experience.

Description: The features of the game are OK; however, the game experience is not fun, and it's necessary to ideate on solutions.

Context: Based on the learnings you received via the test feedback, the game is not interesting to play, although it's working very well.

Analysis: The tests on demos showed you that, in spite of the good playability, it's necessary to improve the game experience, adding aspects that provide more fun.

First Ideation

The goal here is to ideate so that the game gets more fun. By reading an action item, you can get more details on what happened, but let's suppose that the game doesn't present any difficulty. Thus, some of the possible ideas involve including new elements in it. Some suggestions may be enemies, collectibles, weather, powers, or any other idea.

The Game Is Fun but Very Difficult to Play

The game is a 2D platform that brings new playability experiences. It aims to allow players to experience the entertainment of a platform but with an extra touch of genius.

Note that the game mechanics are not adjusted to the scenery environment, making it impossible for the player to overcome the obstacles. How can you solve this game difficulty so it becomes effectively playable?

ACTION ITEMS

Title: Improving movement mechanics.

Description: It's necessary to make the game mechanics more real or improve them so the game experience becomes better.

Context: Based on the knowledge you gained via the test feedback, you noticed that, although they were having fun, the players had lots of difficulties in overcoming obstacles and getting used to the mechanics.

Analysis: Improving the game mechanics will make the playability experience and the narrative complement each other better.

Second Ideation

The ideation at this point aims to identify what the game difficulty is and to propose solutions. The difficulties can be poorly implemented features, irregular jumps, lack of predictability on movements, and much more.

Let's suppose that the jump implementation is irregular, making it difficult to jump from one platform to another, in addition hindering the process of eliminating enemies. Some suggestions might be rethink the jump, verify whether the implementation can achieve the desired goals, verify whether the jump design matches the level design, verify whether the level matches the different game metrics, and much more.

Rethink the Limitations on Game Development

In general, you have verified several limitations in the first stages of game development; however, many times, it's necessary to rethink the limitations. Here are some ideas:

- *Which platform is the most important now?* PC? PlayStation 4? Xbox One? Android? iOS? You should know which ones you are implementing for, which one is easiest, what difficulties the current platform brings you, and other obstacles that might come up.
- *Should you avoid something?* What kind of player are you looking to reach? For instance, should you avoid using lots of buttons for young children, or should you avoid using simple commands in the case of mobile platforms? Is the MVP about reaching all kinds of players, or should you focus on a single type?
- *How are inputs and outputs going to happen?* Are you using joysticks, a mouse, and Oculus Rift, or you will only use the keyboard?
- *Are the mechanics easily adjustable?* Were they correctly implemented, and are they easily improved in case you need them in the next MVPs?
- *Can you improve the scripts without breaking the game's experience?* The code needs to be refactorable and loosely coupled so that future changes don't break the code and the game's overall experience.

Tying Things Together

At this point, you are closing the lean development cycle. This is the moment when you are going to rethink your ideas, rethink the hypotheses, iterate for fixing the design, and improve the build. You proposed solutions for problems found, shared them with team members, and planned your next steps. Now you have to review the hypotheses, review the design, evolve the build, and so on, until you have enough iterations to wrap up your game.

Summary

This chapter was short but was a key component of the lean cycle. Here you have to think about everything you have done so far so you can generate new ideas and evolve the previous ideas to continue developing the game in the following iterations.

Book Recap

The key to understanding this book is to always keep in mind Figure 2-2, which showed the lean game development cycle. In addition, always keep in mind the seven key principles of lean, especially eliminating waste. Through some research and much insight you can have a successful inception that results in key ways to measure how your game is being developed and its general acceptance. Another important aspect of the inception is to understand the MVG and its prototypes, how they should be delivered, and to whom they should be delivered.

With the inception finished, you can start thinking of design and build, which are key concepts and which should always be flexible so they can be adapted to the current needs of the game. The design is mostly the recipe, and the build is the cooking. In design, you should think of characters,

history, gameplay, etc., while in build, you should concern yourself with tools, frameworks, engines, practices, and mostly coding.

Next it is time to start coding and drawing, the step where you should always validate your art and code quality and functionality. Keep high standards and deliver to those who can generate feedback. With that feedback, you can start measuring and analyzing to iterate again, until someone, usually the PO, says the game is good to go.

The last tip is to always keep track of team members' ideas and suggestions because they can give the project precious insights about gaming experience.

APPENDIX A

More About Games

How do you differentiate and classify all the things in the world today that we call a *game*? In fact, there are hundreds of names that can be applied to digital games, such as *games*, *serious games*, *gamification*, *advergames*, and many others. Table A-1 l some of the differences between categories. An "x" represents a requirement.

Table A-1. *Differences Between Things Called Games*

	Game Thinking	Game Elements	Virtual World	Gameplay	No Purpose
Playful games	x				
Gamification	x	x			
Advergames	x	x	x		
Simulation	x	x	x		
Serious games	x	x	x	x	
Game	x	x	x	x	x

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3

Figure A-1 classifies the differences among the types of games and the characteristics of each one, according to Andrzej Marczewski's 2015 "Game Thinking" blog post.

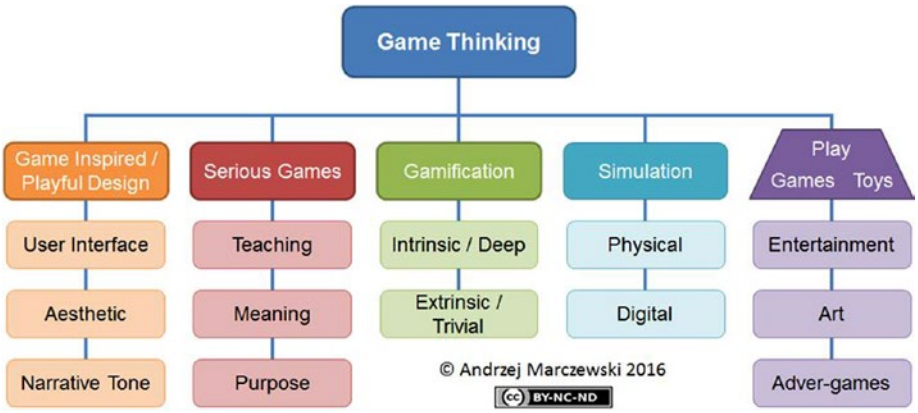

Figure A-1. *Types of games. Source: Andrzej Marczewski (2015).*

The following is a brief explanation of all these categories:

- *Playful games*: This is something that revolves around game operation and strategy but does not have strongly attributed game elements. The interface should look like a game.
- *Gamification*: The idea of gamification revolves around the concept of turning a thing into something that gives the sensation of a game. It consists of taking an idea, transforming it into something playable, and adding game elements.
- *Serious games*: These are complete games created for different purposes of pure entertainment.
- *Games for learning/teaching*: These are games created for helping to teach something by playing a real game.

- *Meaningful games (higher purpose)*: These games aim to pass on a significant message to the player, such as teaching some conflict aspect, the reality in some groups, and much more.
- *Purpose games*: These are games that return some real results. It's a group of games that can include significant games, learning games, and serious games.
- *Simulation*: This is a virtual representation of a real aspect. It does not necessarily require gameplay and usually has some purpose, usually military.
- *Games*: It's hard to define the concept of a game because the class is wide. Just remember that entertainment games can belong to several categories, including the ones mentioned here. Many people refer to pure art and entertainment games as *play games*, which can be confused with playful games. They are a form of art, and many times are digitally interactive films or narratives.

In this book, you saw how to use several techniques, such as inceptions, minimum viable products (minimum viable games and prototypes), metric analysis, test-driven development (TDD), continuous integration, and planning the design and build of the game development so there's no need to reinvent the wheel every time you are going to develop a game and in a way that the goals and the vision of what should be delivered are very clear.

Although I covered these techniques to apply to digital games that aim to entertain, you can easily apply them to any kind of software with game thinking. Thus, when developing your next game, think of how much it can contribute to the community rather than how to build it.

APPENDIX B

Agile and Lean Techniques and Tools

This appendix presents some techniques that can improve the lean and agile dynamics for teams. First you'll see some solutions for stand-ups.

Stand-Up Wall

In 90 percent of cases, a 15-minute stand-up takes much more time than that, often interrupted by unexpected situations and parallel conversations. A solution is to have the team find an online platform, or a board, in which each pair posts a card about what they're doing, the blockages, the solutions, and possible actions.

After a short meeting (preferably before lunch), the team goes through each one of the items, explaining what is blocking each pair and if there are any possible solutions. After that, the team decides the actions to be taken, sets who can help in each blockage, or assigns someone to search for a solution. Figure B-1 shows an example of using Trello for this.

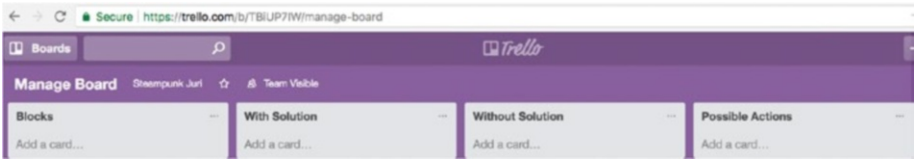

Figure B-1. *Trello board to help a dynamic stand-up*

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3

> Remember that the ideal is always to keep stand-ups dynamic and use different stand-up models instead of sticking to a single way of doing them.

Parking Lot

A *parking lot* is a way of solving parallel problems without deviating from the main subject. It can be used in any kind of meeting, inception, and so on. The goal is to distinguish items that should be solved now from the ones that you can deal later (reminder, lean avoids solving problems now that can be solved in the future).

Generally, a parking lot separates items into these categories: *should be solved now*, *should be solved soon*, and *are not going to be solved*. Figures B-2 and B-3 show examples.

Figure B-2. *Example of parking lot 1*

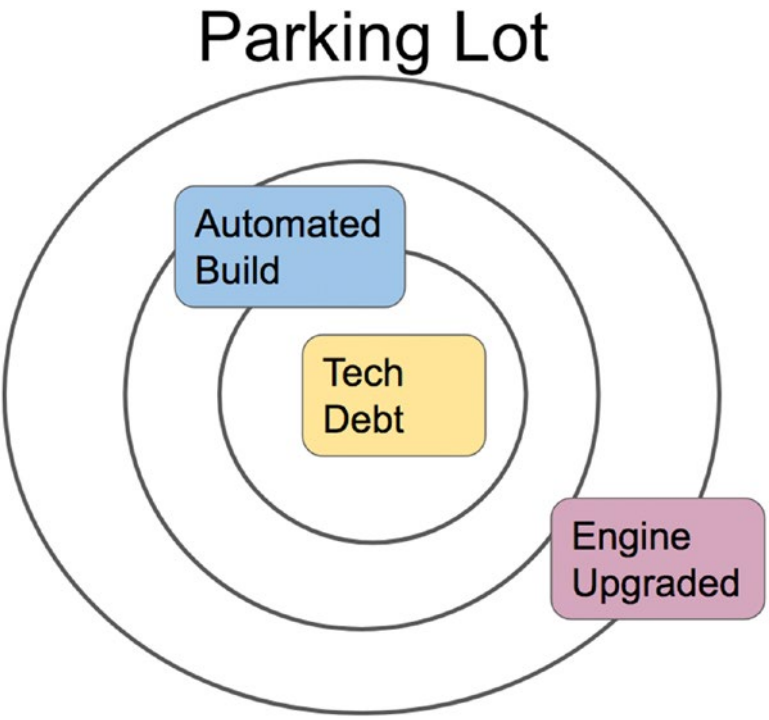

Figure B-3. *Example of parking lot 2*

The parking lot model in Figure B-2 uses the approach of putting inadequate ideas for a meeting on a board so they are dealt with at some other time. The second model does the same by categorizing them by relevance (relevance to the meeting, stand-up, team, and tools); thus, the most central card must be solved first.

Heijoku Board

Visual management aims to get the whole team to understand and see what is going on. It can be represented by Heijoku boards, kanbans, and other types of open boards so the team can visualize what is happening.

Boards make possible for the team to visualize tasks, stages, developments, next steps, or performance. Figure B-4 shows a Heijoku board applied to stages of feature development via TDD.

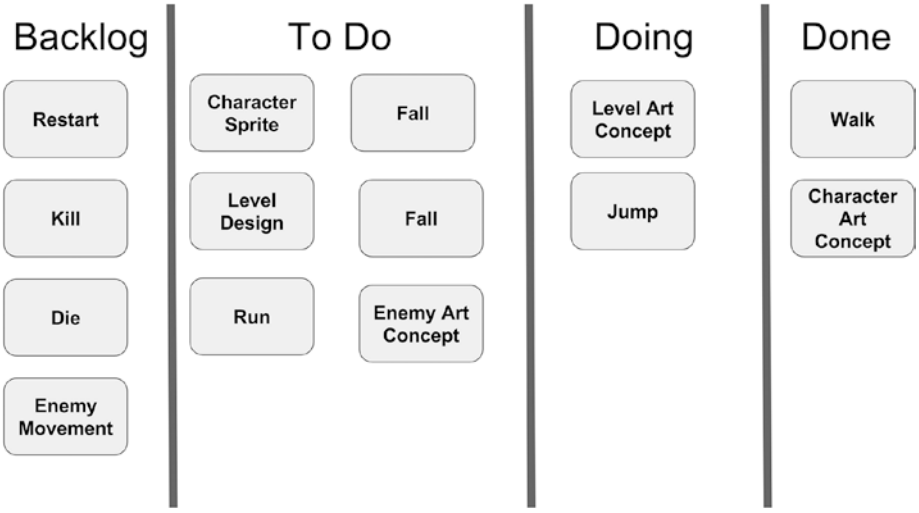

Figure B-4. *Heijoku board for feature development via TDD*

Speed Boat

Speed boat is a technique that allows the team to visualize what is important for the project, what the goal is, what is blocking progress, and what is negative for the project. It's considered a fun innovation game. The game rules are as follows:

- You draw a speed boat on a board.
- The boat is your team, your game, your strategy, or your project. Give the boat a name.
- As the goal of the boat is to go fast, it's important to ask what the optimum performance and most desirable operating conditions would be. These things form the port.

- The distance from the boat to the port represents what you should go through to get to your goals.
- The anchors represent the obstacles and blockages of the boat. The deeper they are, the more negative force they exert.
- White arrows represent the positive aspects pushing the boat.

The following are the steps for playing the game:

1. Present the game to the participants. Splitting them into pairs is usually a good idea.
2. Think of the goals and positive aspects of the project.
3. Present them to the group.
4. In small groups, think and categorize the anchors.
5. Post the anchors on the board.
6. Prioritize collectively the anchors on the board.
7. Discuss a plan of action to remove them.

Figure B-5 represents a speed boat scheme.

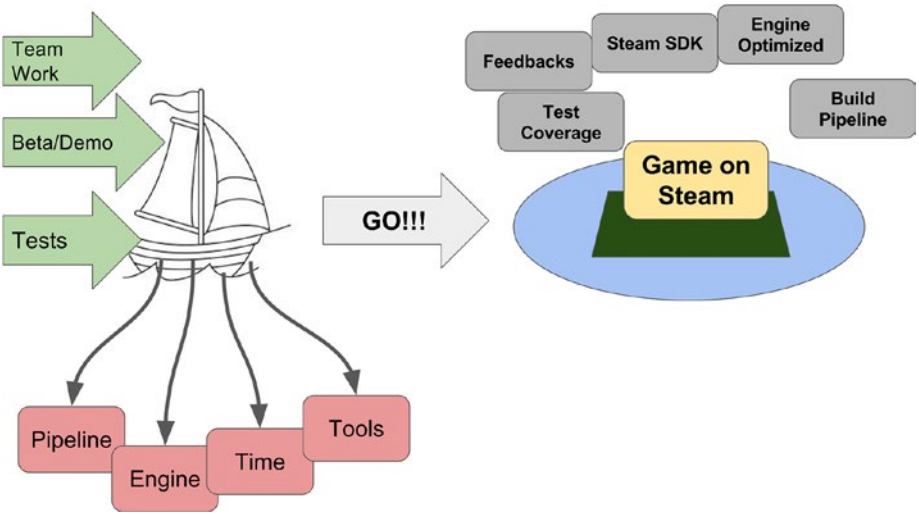

Figure B-5. *Speed boat example (Grosjean, 2011)*

Ice-Breakers

Use *ice-breaker* techniques in meetings. They're a good way of helping the team to relax before an inception, for instance. An ice-breaker's function is to erase the unpleasant mood that can come up because of the presence of several people from different backgrounds, thus relaxing the team.

At ThoughtWorks, at the start of a meeting, people introduce themselves by saying their preferable names and personal pronouns. Another technique used is that each team member has a sheet with their name and a colored pen. The goal is to go around the room asking people to draw a characteristic of a person's face, and then the cards are taped on the inception room's door. These techniques create very laid-back environments.

Happiness Radar

A *happiness radar* is a way of keeping track of how the team feels regarding the technology used, the people on the team, and the processes. The idea is that each member states how they feel (happy, indifferent, or sad) regarding each one of these areas. Then the team discusses each one of the points.

One of the ways of generating discussion after a happiness radar is through the fun retro, presented next. Figure B-6 shows a happiness radar board. It's important to evaluate the team's satisfaction level. The columns represent the rating, technology, people, and processes, respectively.

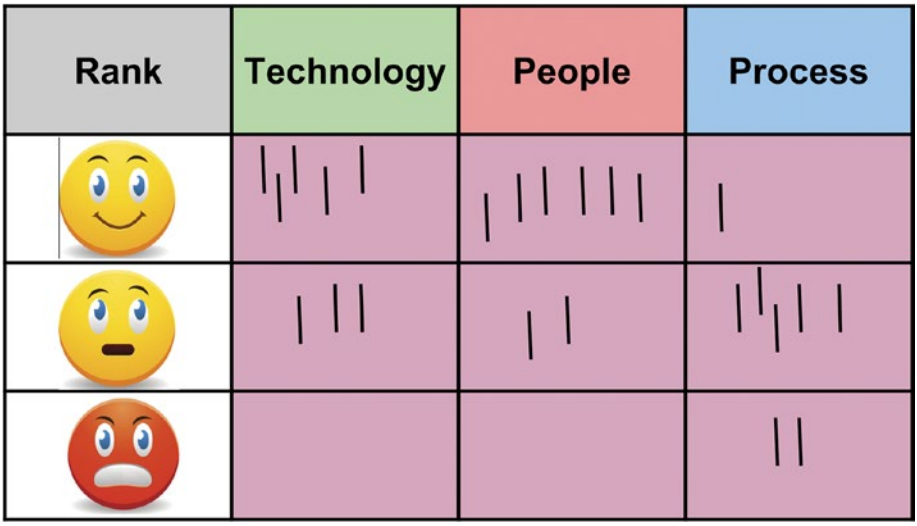

Figure B-6. *Happiness radar example*

Instructions For the current context, indicate how happy you felt (in general) regarding the presented areas.

Fun Retro

A *fun retro* is a platform that allows the team to make a retrospective of what happened in order to identify what was good, what can be improved, what was missing, and the actions that can be taken. The idea is that every team member writes cards that later will become public and then voted on.

Later, the ones that get the most votes are discussed in order to improve the understanding of the team over the events that happened and come up with necessary actions. Figure B-7 shows a board with the fun retro retrospective.

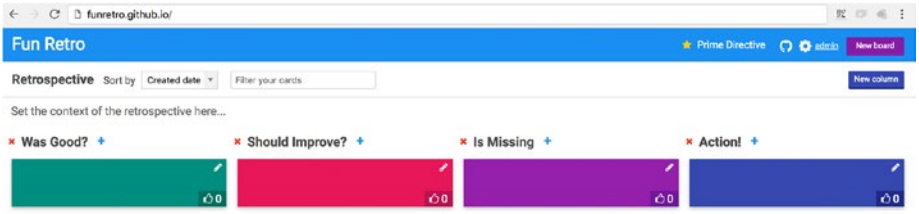

Figure B-7. *Fun retro example*

3Ls

3Ls is a method of having a retrospective that looks like the fun retro, but it's just a bit different. The name comes from "learned, liked, lacked."

This technique consists of having team members put sticky notes in three different areas of a board corresponding to each of the Ls (Figure B-8). These sticky notes are grouped by proximity and then voted on to generate discussion.

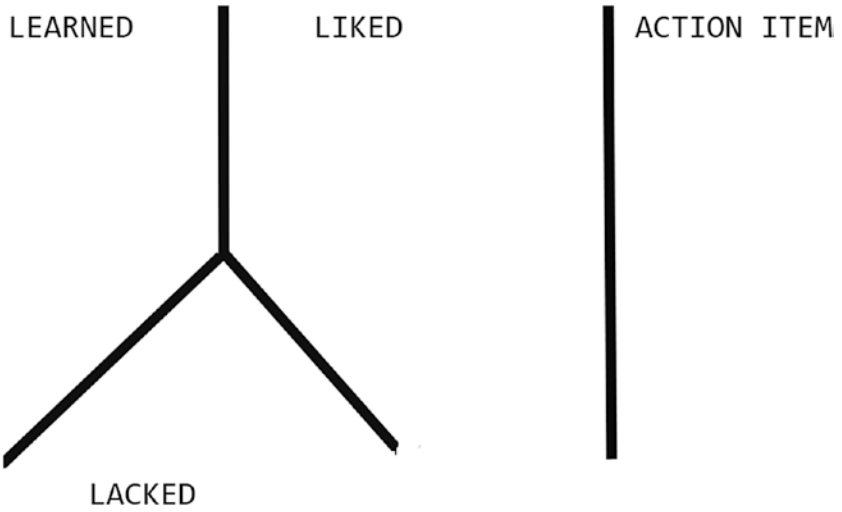

Figure B-8. *3Ls board*

Discussions can and should generate action items that are assigned to teams or individuals so that they are responsible for guaranteeing their solution. Not all action items are technical tasks, but usually they are teamwork.

360 Feedback

In the 360 feedback system, team members get anonymous feedback from people they work with. This technique can be used in teams as well.

In this system, competences, problems, and suggestions are surveyed. Feedback then is ranked to determine which ones should be dealt with first and which is most relevant. Another technique is to generate questions that should be answered as the time passes, as a way to evaluate improvement.

A good place to learn more about the 360 feedback system is www.custominsight.com/360-degree-feedback/what-is-360-degree-feedback.asp.

Roles and Expectations

It's important to help the team define the roles and expectations that everyone on the team must perform. One of the main goals is to allow the team to make clear what each member must do and, many times, how each member must act before the team.

This activity is recommended when the roles of team members are not clear or the team's health is being affected by some member that is not performing their role. Many times, it's a way of giving feedback with less risk of negative reactions.

The activity focuses on a table, as shown in the following figures, in which each team member describes their thoughts on the activities to be performed by each function within the team. Figure B-9 shows there is more room for developers. The language differences are also something interesting to see, because people from different countries were present in the activity remotely.

From \ To	DEV	BA	TL	PM	PO
DEV	Responsible for Quality, Security, Delivery	link between business and technical	soft skills, risks management, coaching		Product Vision
	Responsible for Quality, Knowledge Sharing, Delivery	Keep team developing value	Should identify the team weaknesses and help solve them	Link between product and humanity	What should we achieve
	Delivery	Business Vision and Negotiation	Leadership	Macro management	How should we achieve
	Performance, Delivery and Quality	Bridge between developers and business	Team mood	Keep the team happy	Solve problems
				Responsible for the team workflow	

Figure B-9. *Roles and expectations for developers*

As soon as all members describe what they expect from each function, everyone votes on the best description. The winning descriptions are grouped, forming a new table with all descriptions of the duties and responsibilities the team expects of each function, as shown in Figure B-10.

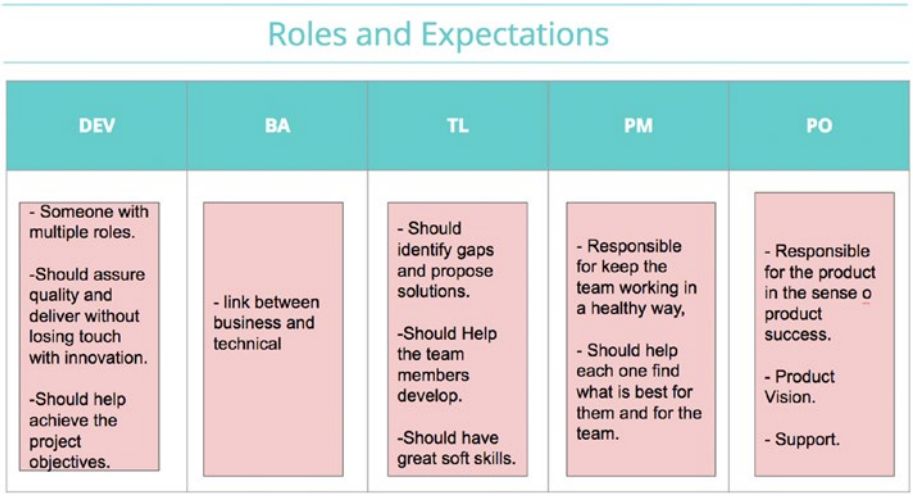

Figure B-10. *Results from roles and expectations*

APPENDIX C

Engines

Engines are software programs that integrate different game elements in a single place. One of the greatest advantages of using an engine is the ability to export games for diverse formats. Another advantage of using an engine is the ability to save well-tested elements (*prefabs*) for your next games, decreasing the need to "reinvent the wheel," which commonly happens in game companies.

Therefore, when choosing a game engine to use in your project, it's important to take into account the following: the export size and activity of the development community, the ease with which to obtain new resources, the performance of the engine regarding each type of game, and the features for practicing TDD.

It's important to remember that there's no reason to reinvent the wheel every time you are creating a new game. The following are five important tips when it's time for you to choose an engine:

- Speed of development, including collision systems and predefined physics
- Ease of configuring the player joysticks, from keyboards to smartphones (Unity is great for this)
- Good documentation and an active community
- Easy-to-understand resources
- Ease of distribution on multiple platforms

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3

This list was extracted from http://materiais.producaodejogos.com/game-engine-pdj.

The following sections give some examples of engines.

Unity

https://unity3d.com/

Some games from the *Angry Birds* franchise were built using Unity. Unity supports two programming languages adapted for games, C# and JavaScript. Unity allows you to export to most platforms and has stable versions for Windows and Mac. There's a free version for limited sales and a Pro version for a monthly fee (Figure C-1).

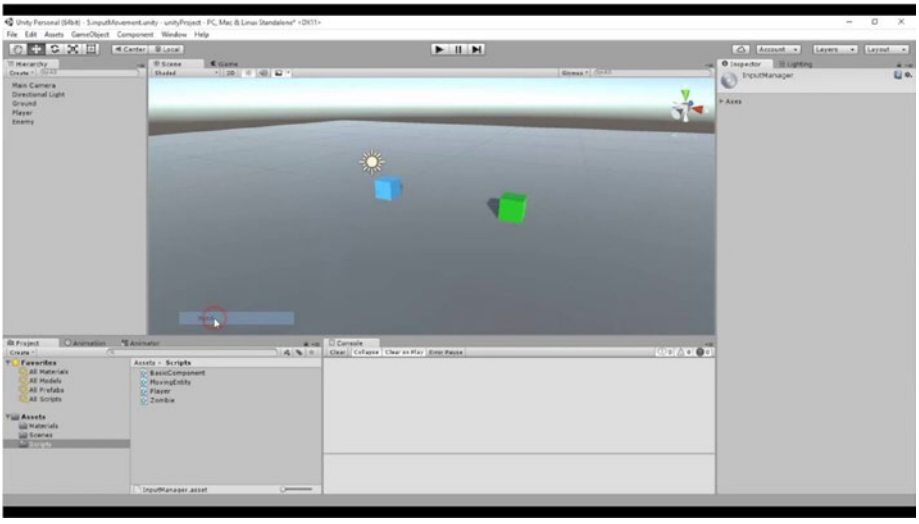

Figure C-1. *Unity editor. Source:* https://youtu.be/-eUbcA42-9I.

Unreal Engine

https://www.unrealengine.com/

The great advantage of Unreal Engine is that it's open source, with its code available on Git (Figure C-2).

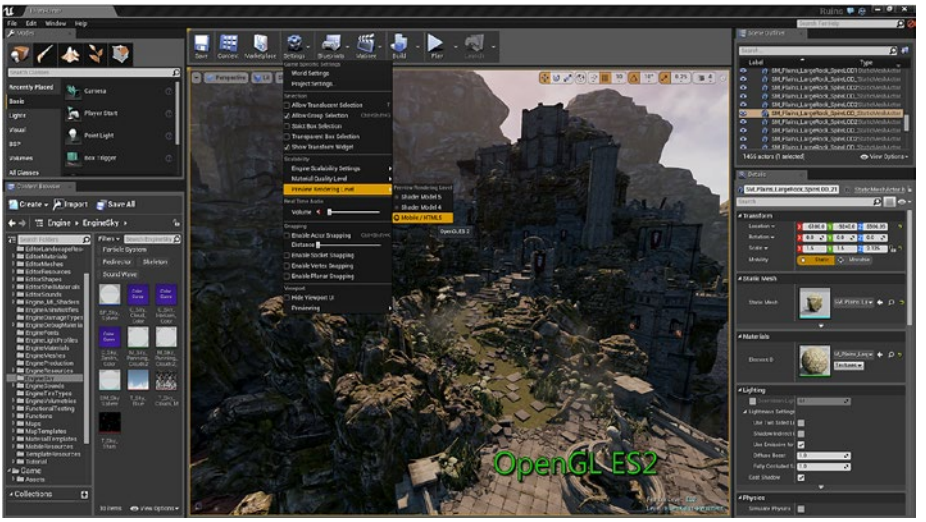

Figure C-2. *Unreal Engine editor. Source: https://forums.unrealengine.com/showthread.php?53230-Unreal-Engine-4-6-Released!.*

It's free for products with sales under $3,000; over this value, it costs 5 percent of the royalties per product per quarter. It's developed by Epic Games and supports C++ and its own visual language.

CryEngine

https://www.cryengine.com/

CryEngine was developed by Crytek for *FarCry*. Ubisoft offers an internal engine that is a highly modified version of CryEngine, called Dunia Engine (Figure C-3). It supports mainly C++ and Lua but can be integrated with C# scripts. It exports to most console platforms and recently became free. It's a good engine along with Unreal Engine and Unity.

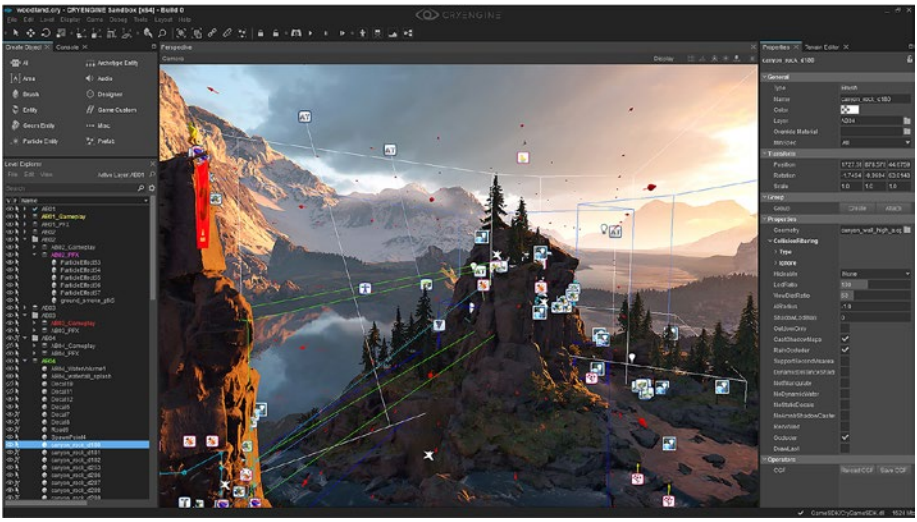

Figure C-3. *CryEngine editor. Source: https://www.cryengine.com/features.*

Construct2

https://www.scirra.com/construct2

Construct2 is a good engine if you don't know how to program and offers intuitive and easy-to-learn tools (Figure C-4). The engine allows you to produce 2D multiplatform games in HTML5.

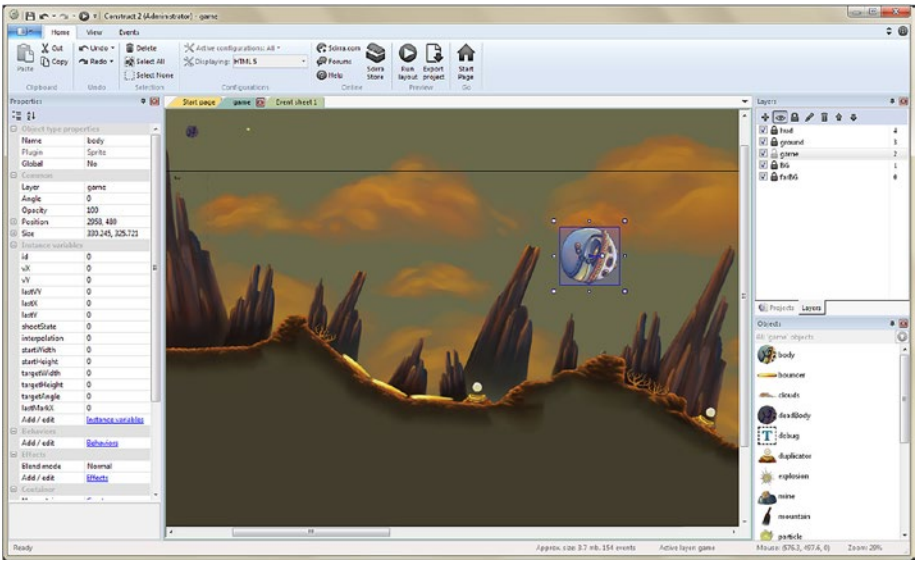

Figure C-4. *Construct2 editor. Source:* `https://www.scirra.com/construct2`.

Game Maker Studio

`www.yoyogames.com/gamemaker`

This is one of the most common engines, especially among beginners (Figure C-5). It allows you to export to several platforms, such as Steam and Windows. However, for some platforms, it's necessary to buy additional modules.

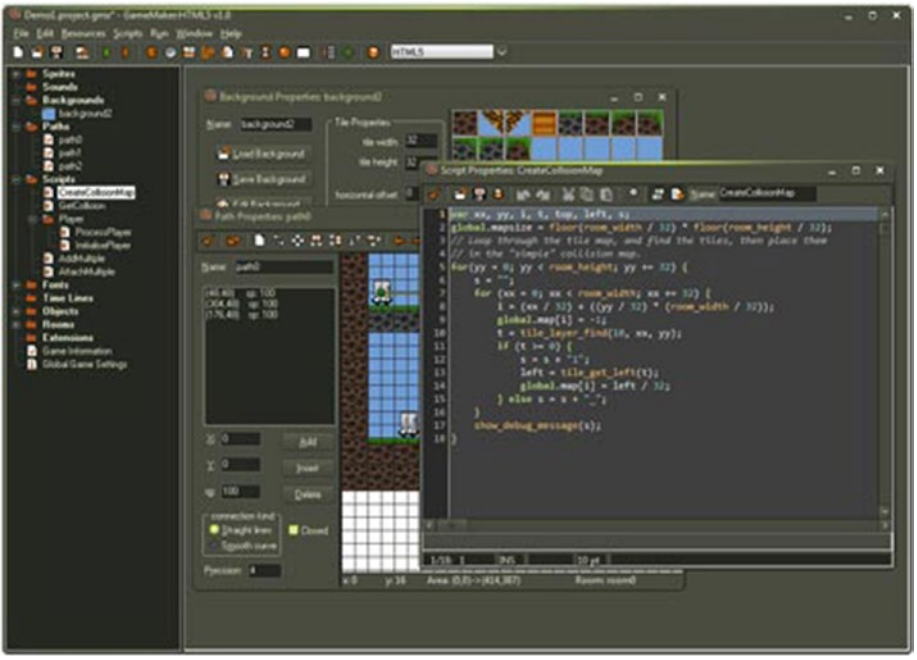

Figure C-5. *Game Maker Studio editor. Source:* `edshelf.com/tool/gamemaker-studio/`.

The engine already has some predefined resources of sound, music, textures, fonts, and so on. The free version is quite limited, but the Pro version is available for less than $200.

RPG Maker

`www.rpgmakerweb.com/`

Maybe the gateway for most developers, RPG Maker is a common and simple-to-use engine (Figure C-6). It's a classic tool for real-player games (RPGs). It allows you to export to many platforms, such as macOS, Android, iOS, and HTML5. The game patterns are similar to the *Pókemon* games for Gameboy.

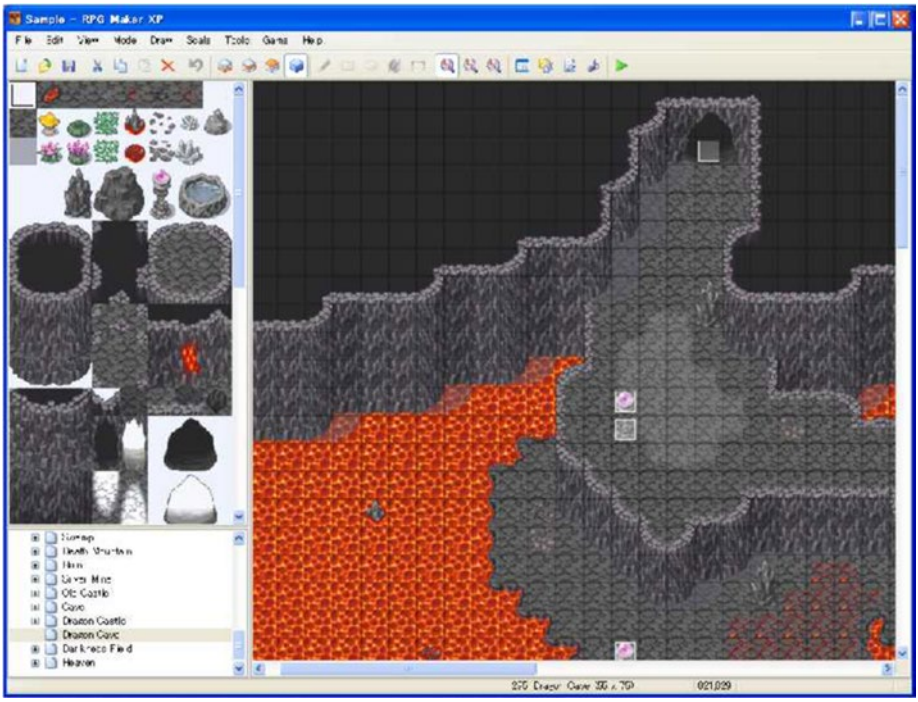

Figure C-6. *RPG Maker, one of the many versions*

Panda3D

https://www.panda3d.org/

Panda3D is an open source game engine written in Python and C++, developed first by Disney (Figure C-7). Developers use Python to program with it. The community is small but very active. Because of the use of Python, it's one of the engines most likely to use TDD, with full language support.

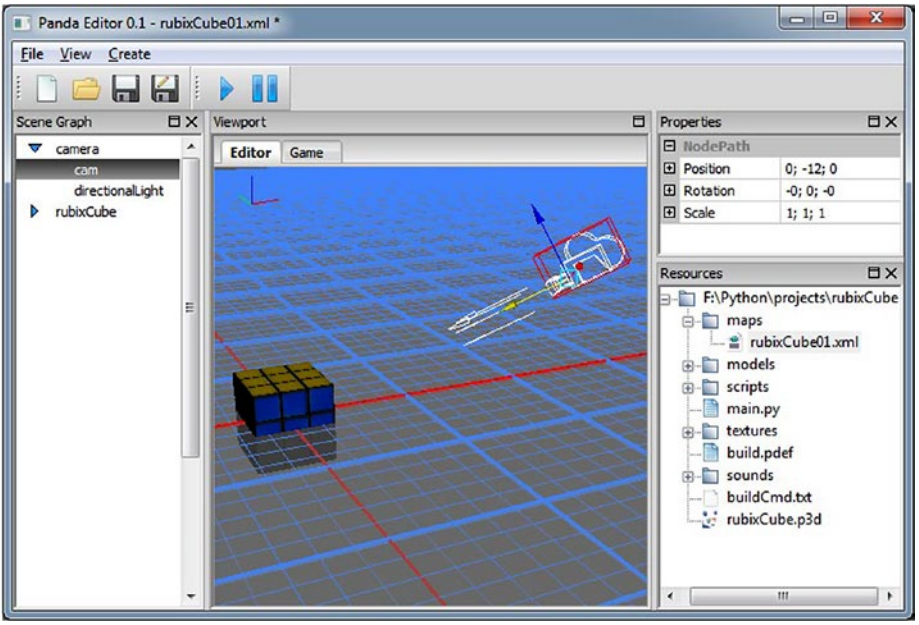

Figure C-7. *Panda3D editor. Source:* `http://thetechartist.com/?cat=7`.

MonoGame

`www.monogame.net/`

MonoGame is not exactly an engine but a game development framework created over the old game development framework from Microsoft XNA. It runs in C#, and it's primarily recommended for 2D

games that need a lot of performance and simple coding. Of all the listed engines, it's the easiest for executing TDD. It's great for both Windows and Mac games.

PyGame

`www.pygame.org/lofi.html`

This engine is similar to MonoGame but written in Python. It's great for starting out in game development and allows you to explore the abilities of TDD. It can be exported to any personal computer format but does not support mobile.

Index

J. N. Rosenfield Boeira, *Lean Game Development*,
https://doi.org/10.1007/978-1-4842-3216-3

H

I

J

K

L

M, N, O

P, Q

R

S

T

U

V

W, X, Y, Z