CHAPTER TWELVE

Asteroids

Chapters 12 through 16 feature in-depth examples that will provide recipes for the game feel of the games they describe. To develop these examples, we'll apply the classifications we built in Chapters 6 through 11, breaking down well-known existing games into input, response, context, polish, metaphor and rules. The idea underlying these breakdowns is not simply to clone these games, though that's certainly possible using this information. The idea is to better understand the hundreds of tiny implementation decisions that gave these games the feel they have. In many cases, these decisions seem counterintuitive—artificially changing gravity at the highest point of a jump, for example. All these little decisions and relationships together, though, are what make these games feel good.

The creators of these games did not follow a particular methodology. They noticed something that bothered them about the feel and tried different implementations until it felt better. Coming at it from a more structured place, we can look not only at the decisions they made inside their specific games, but make generalizations about why this worked the way it did and how it can be applied to all games. That's the idea, anyway. We want to understand the principles underlying these decisions outside of the context of specific games.

With that in mind, we're not interested in specific implementations in specific languages; you could achieve the same feel coding in Actionscript, C++ or Python. It really doesn't matter. Also, throughout these chapters I will reference specific examples. I highly recommend you go to http://www.game-feel.com and download the examples, experiencing the difference in feel at various points throughout the example. These things are better felt than described. For each demo, the idea was to expose the important parameters, enabling you to feel differences in game tuning without having to program anything. I recommend going to http://www.game-feel.com/examples/ and downloading each applet so you can follow along. These things are best felt.

At the start of every example section, I've provided a URL to a naked version of the system, with every parameter tuned to zero. Consider these an exercise in tuning. The pieces are all there; if you want a challenge, I suggest trying to recreate the feel of each game from this all-zero tuning.

The Feel of Asteroids

Asteroids redefined the meaning of "video game." It was the iPod of its time, as synonymous with video games as Apple's ubiquitous product is with digital music now. When the game was released in 1979, it sold more than 70,000 units, obliterating all previous sales records, including those set by Space Invaders the previous year. Space Invaders, itself a game so popular that it caused a national coin shortage in Japan, was left in the dust. Why was this? Why did Asteroids dominate? How could it handily overtake a game so overwhelmingly popular? The answer hinges on the game's unique feel.

Asteroids was, in essence, a rebalancing of the formula set down by the progenitor of all good-feeling games, Steve Russell's Spacewar!. As such, it featured programmatically simulated inertia and discrete tracking of velocity, acceleration and position of the ship. Pressing the thrust button added force into the simulation, accelerating the ship forward in whatever direction it was currently facing. Turning the ship was a much simpler affair, overwriting the ship's orientation to rotate it left or right.

The combination of this detailed simulation for position and the simple, direct rotation gave Asteroids both crisp precision and a flowing expressivity. It was as though the ship was always on the verge of being out of control, but it never actually was. The player's job was to steer and tame it. This feel was not novel. There had been numerous attempts to bring the Spacewar! feel to arcades, including Cinematronic's Space Wars and Atari's own Computer Space. The insight of Lyle Rains and Ed Logg in designing Asteroids was to find just the right combination of rules and spatial context for the tuning of the ship's movement. The spinning asteroids provided just the right context, dominating the screen space in a difficult but not oppressive way. They were just the right shape, size and speed for the motion of the ship, providing close shave after close shave and round after round of fluid, expressive excitement. The rules were simple, encouraged a very clear and particular set of skills and rewarded continued play.

By comparison, the feel of Space Invaders (Figure 12.1) was stiff and rigid. Its motion was limited to a small region at the bottom of the screen. Its left-right steering changed only the ship's position, and did so rather slowly. It is an enjoyable game but in terms of feel, it does not compare to the rich flowing motion of the ship in Asteroids.

William Hunter, curator of the excellent video game history Web site thedoteaters .com writes, "I was never much of a Space Invaders fan. … But Asteroids, with its cool ship inertia and frighteningly close shaves between the rocks, is simply a masterpiece of design and programming. While the games that had inspired Asteroids, such as Spacewar! and Computer Space, had pioneered the concept of inertia in video games, the feeling of actual physics being played out in Logg's creation is another big draw of the game."[1]

[1]http://www.thedoteaters.com/p2_stage2.php

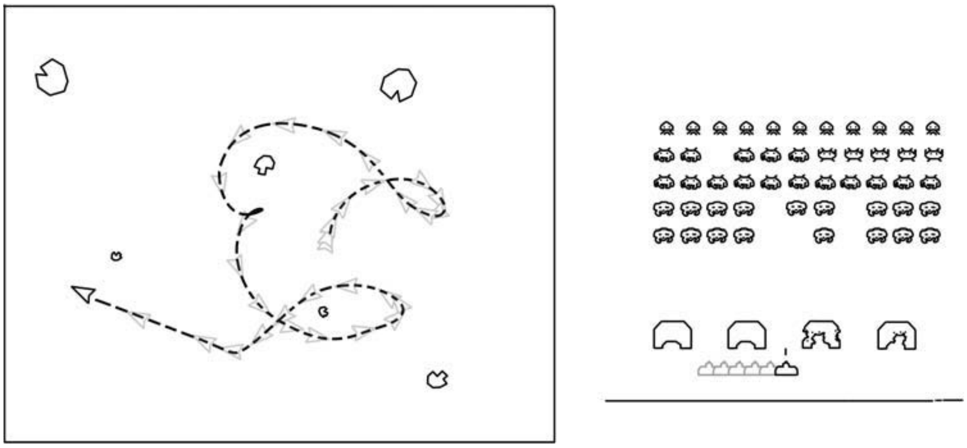


FIGURE **12.1 Compare the movement of the ships in Asteroids and Space Invaders.**

There are still games being made today that emulate the feel of Asteroids, such as Shred Nebula and Geometry Wars. The feel of Space Invaders, however, is all but gone from modern mechanic design.

Input

Asteroids' input space consists of five standard buttons (Figure 12.2). The buttons are far apart, making control a two-handed affair, though all five buttons can be pressed simultaneously. Though the possibility exists, there are no chorded moves in Asteroids.

Physically, the Asteroids cabinet is big and bulky, made out of wood. The surface in which the buttons are embedded is nice and smooth, made out of molded plastic; it feels good. The buttons themselves are big and springy and make a satisfying noise as they click down. When pressed, they become almost completely flush with the surface of the cabinet.

Each button has two states and sends the usual Boolean signals of ON, OFF and HELD.

Response

The incoming Boolean signals modulate parameters in the game in the following ways:

The Rotate Left/Right buttons rotate the ship along its axis clockwise and counterclockwise (Figure 12.3).

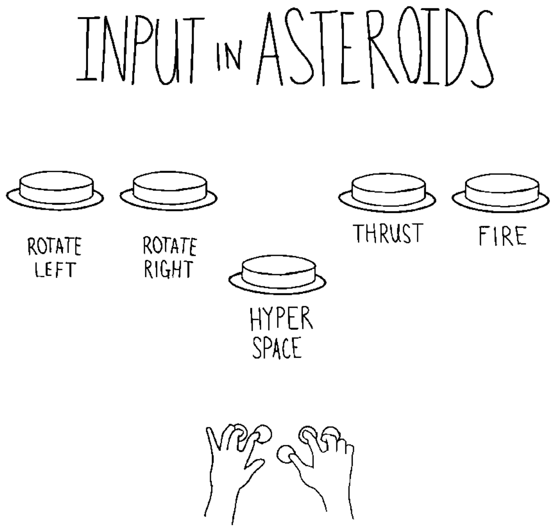


FIGURE **12.2 Five two-state buttons are the inputs for Asteroids.**

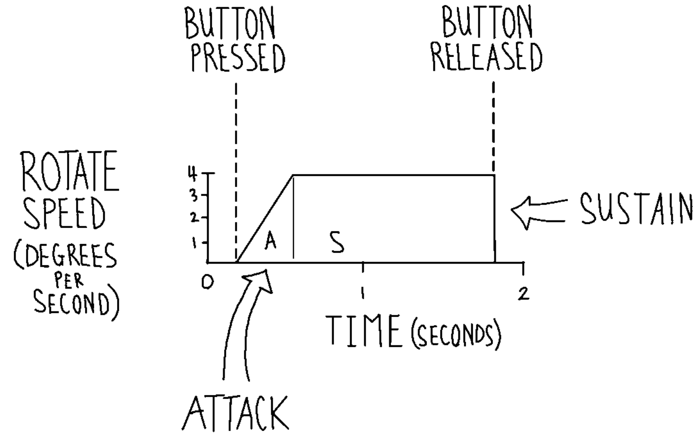


FIGURE **12.3 Attack, delay, sustain and release for the rotation in Asteroids. There is a very slight attack value.**

The Thrust button adds force along the local forward direction of the ship, limited by a maximum value. This progression is highly floaty, taking around three seconds to reach sustain and even longer to release (Figure 12.4).

The Fire button launches a bullet along the local forward direction of the ship, limited by a timed delay. The bullet inherits the velocity of the ship. Only four bullets can be on screen at one time.

The Hyper Space button sets the ship's position to a new, random value.

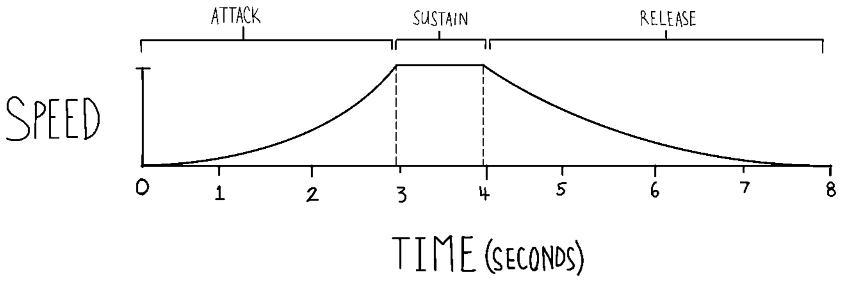


FIGURE **12.4 The ADSR envelope for thrust in Asteroids.**

Simulation

To create the feel of Asteroids, the ingredients are one ship avatar, two varieties of flying saucers, screen wrap and a healthy sprinkling of asteroids. Since the asteroids and alien spacecraft exist mostly to provide spatial context for the motion of the ship, we'll hold off discussing their behaviors and motion for now and instead focus on the ship.

Playable Example

Open example CH12-1 and try to recreate the feel of Asteroids. The necessary parameters are there, just zeroed out.

The ship avatar in Asteroids has two basic motions, accelerate and rotate (Figure 12.5). The rotational motion is crisp and precise while the acceleration is loose and sloppy. In both cases, the motion's frame of reference is the ship itself, a "local" motion.

The most important relationship to the specific feel of Asteroids is the decoupling of rotation from thrust. This is achieved by storing separate velocity values for the ship and for the thrust that gets added to it. If these values are not separated and the thrust velocity overwrites the ship velocity directly, the feel is more like a squirrely remote-controlled car than a smooth, flowing spacecraft.

Let's build the feel up from the beginning. First, we need a ship that rotates. The rotation of the ship in Asteroids is simple. If one of the two rotate buttons is held down, the game adds a small value to the ship's orientation in the corresponding direction, clockwise or counterclockwise. There's no simulation in the code, no acceleration to speed up or dampening to slow down. If the button is held down, the ship rotates. If not, it doesn't. However, there is a very slight Attack value applied to the input signals as they come in (Figure 12.6).

The ship does not go directly to full speed rotation from a standstill. There's a short ramp-up in speed, an Attack, over about a third of a second. It's subtle, but

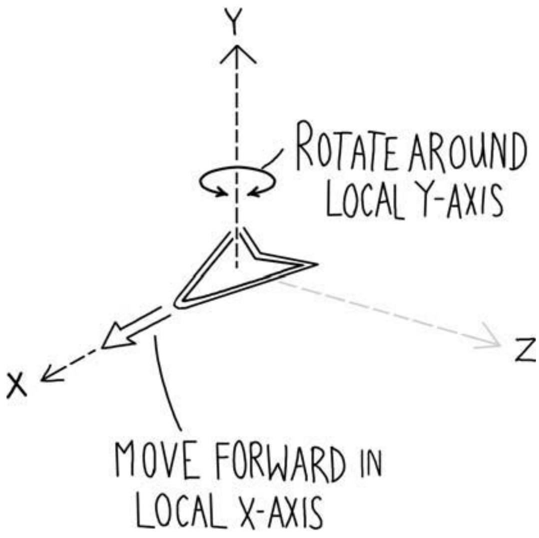


FIGURE 12.5 **Dimensions of movement for the ship in Asteroids.**

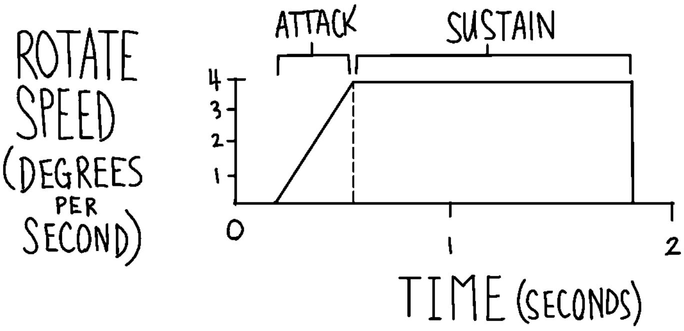


FIGURE 12.6 **The attack takes only a quarter of a second, but its absence is noticeable.**

without it, the rotation feels stiff and robotic. An interesting thing to note is that, perceptually, it's just as though the ship had a very slight inertia that had to be overcome. From the player's point of view, it seems as though the ship took that third of a second to ramp up to full rotation speed.

Playable Example

To experience this subtle difference, open example CH12-2 and click on the "Raw Input" checkbox.

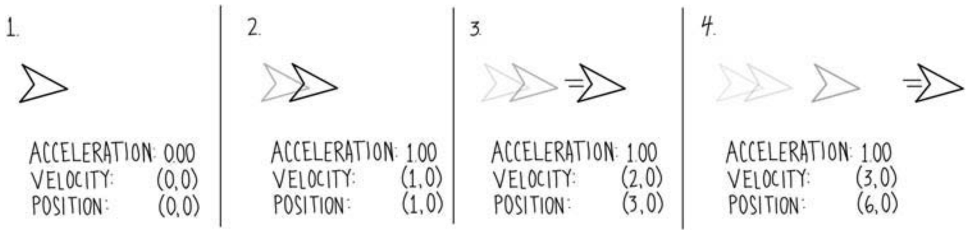


FIGURE **12.7 The different values that drive the thruster movement of Asteroids changing over time.**

Next we want the ship to move forward in response to the Thrust button. This motion will be relative to the current heading of the ship, which causes the left and right rotation to control the direction of the ship. If the forward acceleration were relative to the camera or to some other object in the world, the ship would be stuck moving in one direction, which would not be much like Asteroids at all. Now, if we set the position of the ship the same way we're setting the rotation, by overwriting it directly depending on whether or not the button is held down, the feel is stiff and inorganic. It's crisp, precise and responsive, but moves so differently from any object encountered in everyday life that the motion feels jarring and unsatisfying.

Playable Example

To experience this, click on the "Mode: Set Position" checkbox in example CH12-2.

This is clearly not what we want. The first big thing that's missing is static inertia. In Asteroids, the ship speeds up gradually to its maximum and continues moving at that speed indefinitely until another force acts on it. To get this sense of inertia, we'll separate position from velocity and have the Thrust button modulate acceleration instead of modifying position directly. In each frame, the acceleration value is added to the velocity value, which then updates the position value based on how far the ship has moved (Figure 12.7).

Playable Example

To experience this, click on the "Mode: Translate" checkbox in example CH12-1.

This is the feel of a frenetic racecar gripping the road with perfect sideways friction. It carves circles, without any hint of the desired Asteroids floatiness. The motion is interesting, even aesthetically pleasing, but it feels totally different from Asteroids because the rotation and thrust are now inextricably interconnected.

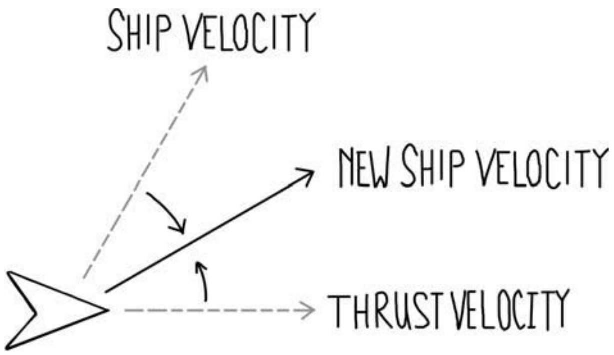


FIGURE **12.8 The most important relationship to the feel of Asteroids is the ship velocity to the thrust velocity.**

Turning changes heading instantly, every frame, while the lack of dampening causes the ship to go on forever without hope of slowing down. To arrive at the feel of Asteroids, the thrust vector must be separate from the ship's velocity. When the thrust button is pressed, instead of setting the modified ship velocity directly, a new vector is created using the ship's heading as its direction and the thrust speed as its magnitude. This is the thrust vector and when the thrust button is held, this vector is added to the ship's current velocity (Figure 12.8).

Great success! This change, between adding and overwriting velocity vectors, makes all the difference. We now have a simulation that can be tuned to the precise feel of Asteroids.

Playable Example

To experience this, click on the "Mode: Asteroids" checkbox in example CH12-2.

The final things to note are the limit on velocity, the screen wrapping effect and the very low dampening force.

An arbitrary maximum is applied to the ship's velocity vector. This value may change the feel slightly if it is set very high or very low, but primarily it serves as a catchall to prevent the ship from having runaway speed. If you're interested in changing this limit to see the difference, it's the "Max Velocity" parameter.

The screen wrap effect is achieved by detecting whenever the ship's position is greater than the size of the screen, then setting its position to the opposite side of the screen. For simplicity, this is done for the X and Y edges of the screen separately. The screen wrapping, like the Max Velocity parameter, is mostly a pragmatic measure. Without screen wrap in place, the motion of the ship avatar carries it off screen in a matter of seconds.

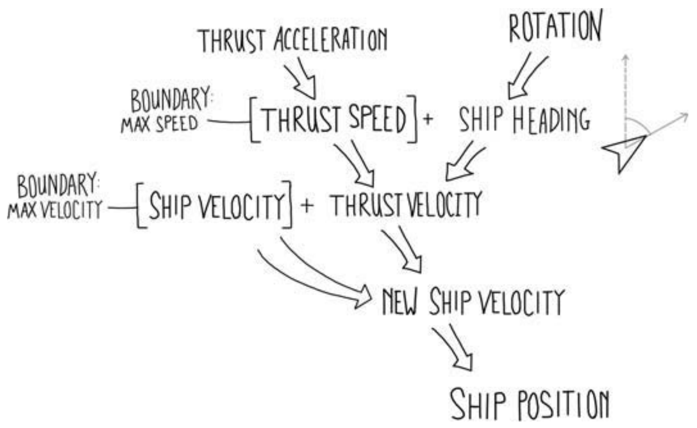


FIGURE **12.9 The relationships that create the feel of Asteroids.**

Finally, there's very little dampening on the motion of the ship. Once accelerated, it will keep going at speed for more than four seconds before coming to rest. This low friction produces a "spacey" feel. Though we Earthbound schlubs have never experienced frictionless motion, when we see it in a game it reads as space-like because of our exposure to news footage of astronauts and science fiction films such as "Apollo 13" and "2001: A Space Odyssey."

The final set of variables we're tweaking are these (their relationships are shown in Figure 12.9):

- Ship Rotation—How quickly the Y-axis orientation is changed clockwise or counterclockwise per frame
- Ship Position—The position of the ship in absolute space, expressed as an X position and a Y position
- Ship Velocity—The current direction and speed of the ship in absolute space
- Thrust Speed—The current thrust value
- Thrust Acceleration—The amount the thrust value will increase over time as the thrust button is held
- Thrust Velocity—A vector representing the force that gets added to the ship velocity when the thrust button is held. It gets its direction from the ship's current heading (which can be different from Ship Velocity) and its speed from the current Thrust Speed value
- Max Thrust Speed—Limits Thrust Speed to a hardcoded maximum value. Speed can not be greater than this amount

To summarize the simulation of Asteroids, when the game receives the signal for "thrust button held down," a force is applied along the ship's forward-facing axis. As the button is held, that force is increased according to the acceleration value, to a predetermined maximum. Whatever direction the ship happens to be facing, the thrust is applied in that direction as an additive vector.[2] It doesn't simply overwrite the previous velocity vector of the ship, but rather gets added to it. This is crucial because it decouples the rotation of the ship from its thrust. This separation of thrust from rotation is the most important part of the feel of Asteroids. It enables the ship to rotate freely regardless of its current velocity. It gives the impression of frictionless motion as well as creates a slightly manic feeling of being constantly out of control. It's only when the thrust button is held that the orientation of the ship affects its velocity, and even then it's additive. The end result is highly floaty: when the player changes direction by rotating, it will take almost three seconds before the velocity of the ship is in line with its heading. In the case of the context and rules of Asteroids, this floatiness is both desirable and awesome.

Context

The only thing that really needs to be said about the asteroids in Asteroids is that they provide just the right spatial context for the ship's movement. The larger asteroids dominate a large amount of the screen, but they move very slowly and are easy to predict. Smaller asteroids take up less screen space but are much more difficult to dodge because they move much more quickly. In all cases, the ship moves faster than the asteroids themselves but because its motion is so wild and squirrely and because the constraint of screen wrapping applies equally to both the asteroids and the ship, every asteroid on the playfield feels unsettling. For me, it feels like being an experienced ice skater at a crowded public rink. When I go to a public skate, I can skate much more quickly than just about everyone on the ice because I played hockey as a kid. But I can't predict when people are going to fall or turn or if they're suddenly going to cut across me to head to the cocoa machine. As a result, I limit my speed and try to give everybody a wide berth. Even though I can stop and turn quickly, I don't have enough control to stop myself from running over or into someone's mom if she biffs it in front of me. Playing Asteroids feels a bit like an extremely crowded kids' night at the local rink. Except for the shooting and the subdividing asteroids, of course.

In terms of functionality, the asteroids are imparted with a random velocity at the beginning of the game. When shot, they break down into medium-sized asteroids and have an additional force applied to them in a random direction. They inherit the velocity of the larger asteroid that spawned them, though, so the likelihood of their speed going up rather than down is very high. The same thing happens when they split into the smallest asteroids. There's no particular insight here; the asteroids, as they get broken down, further clutter the field and become more difficult

[2] A vector is a combination of speed and direction. For example, driving 40 mph to the west would be a vector, where 40 mph is just a speed and west is just a direction.

to maneuver around. The ship moves very quickly and turns rapidly, but the turning never seems quick enough to truly react and get out of the way of an asteroid unless you've planned ahead.

The flying saucers are much more difficult to deal with than the asteroids, but they provide fundamentally the same function. They move in unpredictable patterns, going mostly horizontally but randomly moving up and down. And, of course, they shoot back. The closer you are to them, the more likely you are to be shot, so dealing with them feels a bit like poking a hornet's nest with a long stick.

Generally speaking, the feel of Asteroids is as much defined by the things to be avoided as it is by the motion of the ship itself. The constant, inescapable danger of the asteroids is compounded by the wily flying saucers, and the screen wrap means that you can never escape. These dangers give meaning to the quick, slippery motions of the ship, defining every subtle twist and turn and making it feel almost more out of control than in control.

Polish

Without a lot of processing power to spend, it would be perfectly reasonable for Asteroids to lack for polish effects of any kind. Instead, the Atari engineers rose to the challenge magnificently, with a masterful cohesion across visual and sound effects. Specifically, there is an excellent, consistent relationship between the visual size of objects and the sounds they make. For example, when a large asteroid is shot, it makes a deep, booming sound. A medium asteroid's explosion sound is higher pitched, and smaller asteroids are higher still. Similarly, the large flying saucer makes a lower pitched noise than the small one. Being the smallest objects in play, bullets make the highest pitched sound of all. Of all the sound effects, the thruster firing is the lowest and most rumbling, conveying the sense that it is a comparatively powerful device.

Other subtle but effective polish effects include a spray of particles when an asteroid is destroyed, the ship disintegrating into its component parts when destroyed, and the subtle but effective flashing of the vector line to indicate rocket flame. Because of the limited processing power, each individual effect is drop-dead simple. They harmonize so well, though, that the net effect is a powerful sense of the physical properties of the ship, flying saucers and asteroids. This is a great example of how cohesive, self-consistent effects can be more effective than gaudy, splashy ones that are applied willy-nilly.

Metaphor

As a metaphorical representation, Asteroids is very simple. There appears to be a space ship, but it's more Flash Gordon than NASA. The asteroids and flying saucers reinforce this science fiction theme. The treatment is highly iconic. It's not approaching any sort of realism but it's also not venturing into the realm of the abstract. Each item—the ship, the rocket flames, the bullets, the asteroids—is iconic. They clearly are meant to represent a particular idea. Because the treatment applied to these objects is simple and consistent, there are few expectations being set up for the player. The theme is outer space, so the frictionless feel of the ship is certainly not clashing with the metaphorical representation, but neither is it inextricably linked with it. A car-like physics such as the one we experienced earlier might not seem so jarring because the visualization is so simple.

Rules

The main rules affecting the feel of Asteroids are those related to collision and destruction of the ship. At the start of the game, the player is arbitrarily given three lives. Running into anything—bullet, asteroid or saucer—destroys the player instantly and removes one of those lives. This serves to make the ship seem exceedingly fragile and to make extra lives seem like the most valuable commodity in the game. This sense of value also hooks into the points system: because you gain an extra life for every 10,000 points you score, destroying asteroids feels gratifying and worthwhile. A large asteroid is worth 20 points, a medium is worth 50 and a small is 100 points. This creates a nice value scale for the destruction of asteroids and provides constant incentive to destroy them.

What's really set up, though, is an awesome risk-reward relationship with the tiny flying saucers. Flying saucers are simultaneously the most dangerous and difficult to kill objects and the most valuable. For a large saucer, you get 200 points. For the small, extremely difficult to hit saucer, you get 1,000. So there's a lovely risk/reward tradeoff that happens, because you get so many more points for destroying this tiny flying saucer than for blasting another set of mundane asteroids. When you see one come across, even though it is shooting back at you and is an unpredictable, tiny target, you focus on it and steer toward it because there's the promise of a huge number of points, which moves you that much closer to getting an extra life.

Of course if you lose a life in the process of trying to shoot this damn thing, the point is moot. So there's this circumspect little calculation that happens in your brain about risk and reward. Is it worth it, is it not? How many lives do I have? Do I have a lot? Do I need the points? How close am I to getting extra life? And so on. This sense of value, risk and reward affects feel by driving the player closer to the tiny saucer. In so doing, they learn a whole new set of skills and experience just how out of control the ship is relative to the saucer's quick, precise motions and shots.

Summary

Asteroids was groundbreaking and hugely popular, in large measure because of its unique feel. Applying our taxonomy of game feel, it's easy to see why.

The input device was satisfying, though it only used Boolean on-off buttons, and it mapped well to how things moved in the game action. It also required both hands and five fingers, ensuring the player was challenged (but not too much) and engaged.

The response mapping was clear, simple and easy to follow.

But a lot of the "secret sauce" of Asteroids was in the simulation. Thrust is separated from rotation, creating the most important part of the feel of Asteroids. It creates the loose feel that is so crucial to the feel of the game.

In terms of context, the asteroids in Asteroids provide just the right spatial background for the ship's movement. The constant, inescapable danger of the asteroids is compounded by the wily flying saucers, and the screen wrap means that you can never escape. These dangers give meaning to the quick, slippery motions of the ship, defining every subtle twist and turn and making it feel almost more out of control than in control.

Adding to the overall feel, just the right amount of polish was used in Asteroids—not overdone, not underdone. Visual and sound effects are simple, but cohesive and self-consistent, making the most of the processing power available at the time.

The metaphor—outer space—is simple and iconic, setting up easy to exceed expectations of how "real" spaceships ought to behave in the mind of the player.

Finally, the rules in Asteroids are exceptionally well done, challenging the player to increase his or her skills in anticipation of greater rewards.

Overall, all of these elements were balanced beautifully to create a simple but wildly popular game. The man who conceived Asteroids, Lyle Rains, and the creative visionary who programmed and designed the game, Ed Logg, did everything right. It's no wonder Asteroids was such a hit in the United States and became Atari's best selling game of all time.

This page intentionally left blank

CHAPTER THIRTEEN

Super Mario Brothers

Super Mario Brothers was a breakout hit for video games as a medium.

In 1983, things looked a littles bleak for the future of digital games. “Video game” was a dirty word to retailers, and arcades were shutting their doors with frightening speed. Atari had flooded the market with inferior product, culminating with the much-lampooned E.T. The Extra-Terrestrial cartridge. Consumers lost interest, retailers lost money and doomsayers decried the fiery end of the video game fad. Enter Nintendo and its “Entertainment System.” Improbably, a young industrial design graduate was about to change video games forever.

A quiet, unassuming man who is “very content” with his modest salary and seems genuinely bemused by his worldwide celebrity, Shigeru Miyamoto was an unlikely candidate for “world’s most acclaimed game designer.” As he flashes his trademark smile and casually explains his original sketches for Donkey Kong, you get the sense that he is as excited today about the idea as he was more than 20 years ago. Because there was simply no one else in the company available, Miyamoto was tapped by Nintendo president Hiroshi Yamauchi to create the game, Miyamoto’s first. In the most emphatic and real sense, the future of Nintendo hung on the unproven industrial design graduate and his “Stubborn Gorilla.” Against all odds, the game became a smash hit, in a stroke saving the ailing Nintendo and establishing Miyamoto’s reputation.

While it was the first major hit of the burgeoning “platformer” game type, Donkey Kong still felt very stiff. The character in Donkey Kong, Jumpman,[1] could run left and right, climb ladders, and, of course, jump. His jump followed a specific, predetermined arc and he only ran at one speed. On or off, full speed or complete standstill. There was no gradual acceleration or deceleration and no control over the jump once you were in the air. It was a step forward—a charming, playful game with appealing characters, bright colors and detailed animations—but it still felt very stiff. Miyamoto knew that his games could feel much better. After a successful Donkey Kong sequel, he turned his attention to refining the movements of his now-Italian, now-plumber character in Mario Brothers.

[1]The naming of Mario was, apparently, a conciliatory gesture to the irate landlord of Nintendo of America’s warehouse, Mario Segale.

Mario Brothers was different. This time, Mario jumped much higher, though his trajectory was still unalterable once he'd left the ground. The first hint of the powerful feel that was to come was in Mario's left and right movement. Instead of movement being binary-state (full speed or stopped), when the joystick was pressed, Mario now had three states: stopped, walking and running. As a result, he now sped up gradually, and the player could make quick tapping motions on the joystick to make small adjustments to his position. Likewise, once the joystick input stopped, there was a slight slide as Mario came to a halt. Mario now had inertia. This smooth feel was used in games like Asteroids and the venerable Spacewar! but had not yet found its way into a character-based game about jumping over obstacles and gaps. Mario Brothers was a modest hit, coming as it did at the end of the arcade era.

In 1986, all the elements came together. Super Mario Brothers combined a loose, fluid feel with a powerful character-driven metaphor and a charming, surreal treatment. Instead of one extra state inserted between standing and running, there were hundreds. Mario now accelerated gradually, without perceptible switches, up to his full speed. When the input stopped, he slid gradually to a halt. The game felt intuitive but deep: sloppier and more imprecise than Donkey Kong, but better for it. Somehow it felt more "real." It took the world by storm. The first truly universal hit video game, Super Mario Brothers sold more than 25 million copies worldwide, far and away the greatest selling game of all time. In a 1987 survey, Mario was more recognizable to American children than Mickey Mouse.

Miyamoto understood game feel not in terms of simulation but of simplification. First, he regarded the feel of a game artistically, as a composite aesthetic experience. At a time when the field was dominated by engineers who, in the tradition of Steve Russell, drew on complex, literalistic metaphors like the gravitational pull of black holes or landing a spacecraft on the moon, Miyamoto brought a refreshing, naïve perspective. He simply wanted to make fun, colorful games about whimsical characters that felt good to play. Second, he designed games holistically, taking into consideration both software and input device. (To this day, Miyamoto designs controllers as well as games, a rarity among designers, especially now, with the death of the arcade.) Finally, Miyamoto understood the power of metaphor and how it affected players' willingness to learn and master a complex system and their emotional attachment to it.

Miyamoto had intuited just how powerful the tactile, aesthetic feel afforded by instantaneous reaction to user input could be. Super Mario Brothers felt great, a shining example of possibility for virtual sensation.

Now the big question: just how was this feel created? How does one build a game that feels exactly like Super Mario? Like many questions surrounding game feel, this is a surprisingly difficult question to answer. Just thumb through this chapter and you'll see; even for a game as simple as Mario, there are a huge number of tiny but ultimately important decisions that must be accounted for. Individually,

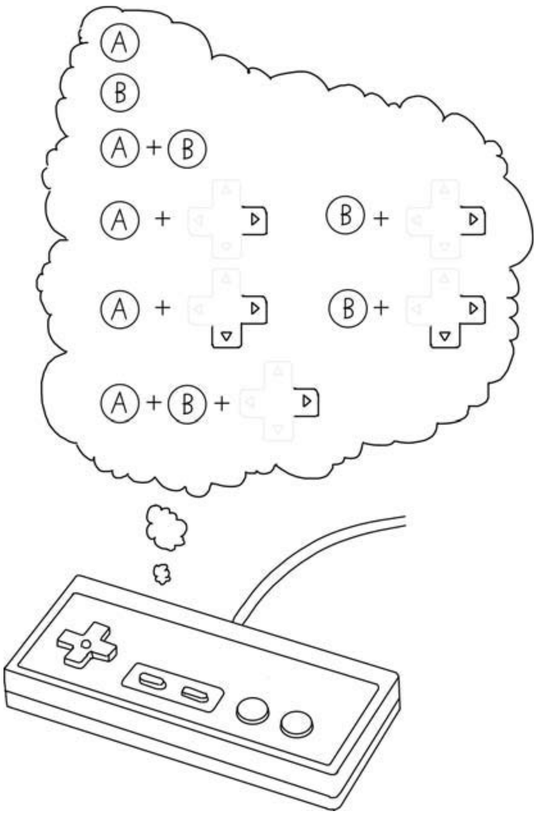


FIGURE **13.1 The simple, but classic, NES controller.**

they often seem trifling and bizarre. Taken as a whole, they lead to the feel that sold more than 25 million copies.

Input

As an input device, we have the NES controller. The signals it sends are very simple, as we have said, and overall it has very little sensitivity as an input device. It feels pretty good to hold and use and is composed of a series of standard two-state buttons. One of its great strengths is its simplicity. When you hold it, it's almost impossible to press the wrong button since there are so few for each thumb to deal with (Figure 13.1).

Button	States	Signals	Combination
A	2	Boolean	B, any direction
B	2	Boolean	A, any direction
Up	2	Boolean	A, B One other direction at a time, except down
Down	2	Boolean	A, B One other direction at a time, except up
Left	2	Boolean	A, B One other direction at a time, except right
Right	2	Boolean	A, B One other direction at a time, except left

Each button sends a binary signal. Taken alone, this input data can be interpreted as “up” or “down.” When measured over time, the signal can be interpreted as “up,” “pressed,” “down” or “released.”

That’s it. Not much more to say about the NES controller; as an input device it is among the simplest, most effective ever created. The plastic on the front is smooth and porous, the buttons springy and robust, and the overall package feels solid.

Note that we’re using the keyboard to control the examples presented here, which will change the feel of control by allowing left and right to be pressed simultaneously and because this input uses multiple fingers instead of a single thumb.

Response

There are two avatars in Super Mario Brothers, Mario himself and the camera. Mario has freedom of movement along a 2D plane, X and Y, as shown in Figure 13.2.

Since the Mario avatar doesn’t rotate at all, there’s no distinction between local and global movement.

The camera (Figure 13.3) is indirectly controlled by the player via the position of the Mario avatar and moves in only one axis, X. Interestingly, it can never move to the left.

A Recipe for Mario

The feel of Super Mario Brothers lives primarily in the main Mario avatar.

If you want to create a game that feels exactly like Super Mario Brothers, the first thing you need is a rectangle. This is how the game views the object that millions of

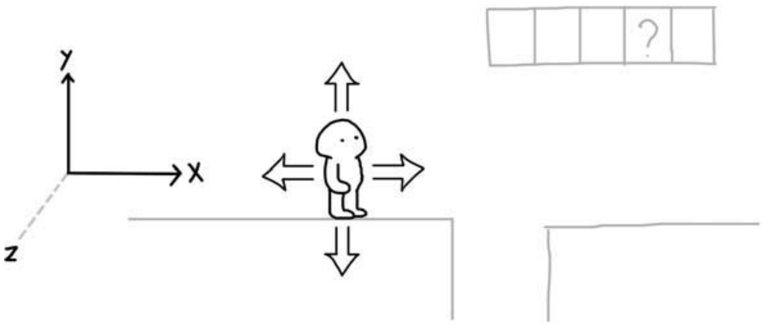


FIGURE 13.2 **Mario moves in two dimensions, X and Y.**

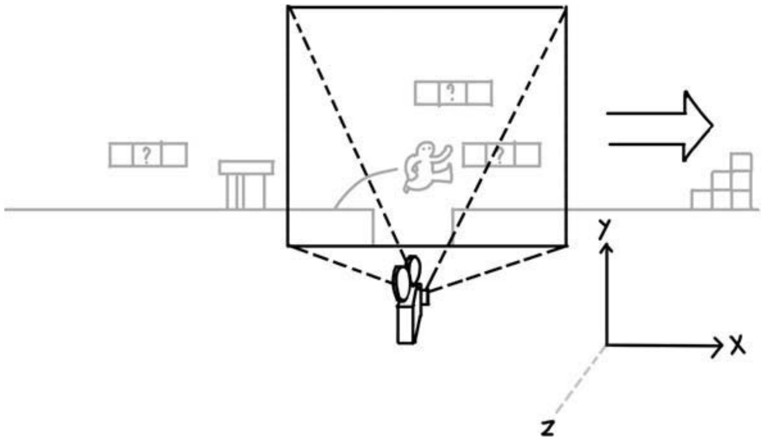


FIGURE 13.3 **The camera in Super Mario Brothers moves in one dimension: the X-axis.**

us know and love as Mario. He's simply a rectangle. More specifically, he's a series of points that form a rectangular shape, but for our purposes it's reasonable to call him a rectangle. So let's start with our rectangle, sitting motionless in the center of the screen (Figure 13.4).

Playable Example

Open example CH13-1 to follow along. To begin, there is no motion, all the parameters are set to zero and the avatar is a blank rectangle in the middle of the screen.

The next obvious step is to have the rectangle move. The way Mario's simulation functions, there are two distinct subsystems at work, the horizontal (X-axis) movement and the more complex vertical (Y-axis) movement. To start, let's focus on the

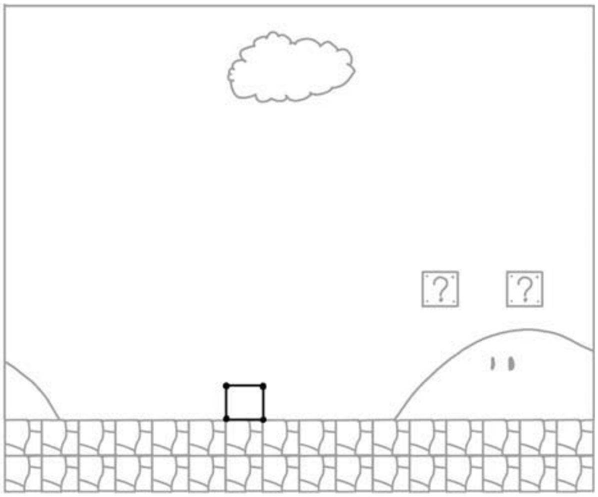


FIGURE **13.4 The shape of Mario: a series of points forming a rectangle.**

horizontal movement. It's the interplay of these two systems that gives rise to the expressive, fluid feel of Mario, but they are kept mostly separate as far as the simulation is concerned.

Horizontal Movement

All horizontal movement in Mario is mapped to presses of the left or right directional pad buttons. The signals coming in are simple Booleans and they can't be pressed simultaneously because of a physical constraint imposed by the input device itself. As a result, at any given time there will be only one relevant signal coming in from the input device: left or right. The simplest way to map this input to a response in the game would be to store only a position for the rectangle. When either the left or right signal was detected, the rectangle's position would change by a certain amount in the corresponding direction. As long as the left button was held, the rectangle would move some distance per frame in the corresponding direction. This is the way that Donkey Kong works, changing position when the joystick is held in a direction. This is not, however, how the horizontal movement in Mario works. Figure 13.5 shows graphing of Mario's movement over time versus Donkey Kong (from Chapter 7).

Now, I like the feel of Donkey Kong. I think it's rather charming. It's hard to argue, however, that it's more expressive than Mario. Mario feels fluid and responsive while Donkey Kong feels stiff and robotic. Look again at the movement of each over time; you can see just how many more places it's possible for Mario be, how much more expressive potential there is for the player. The primary reason for this lies in the simulation. Super Mario Brothers has something resembling a

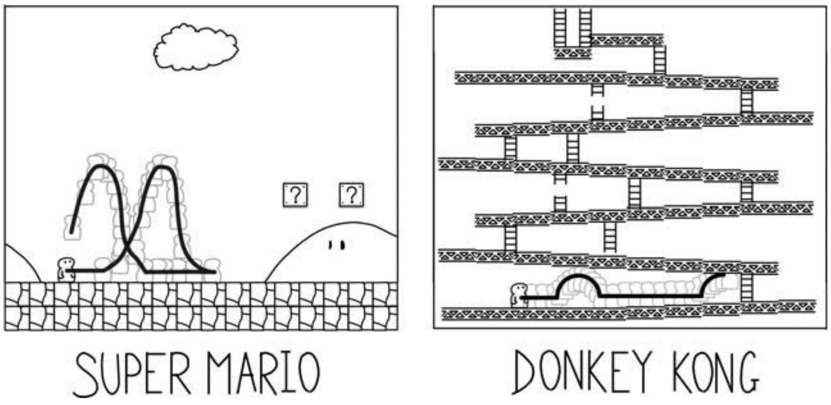


FIGURE **13.5 The movement of Donkey Kong and Mario.**

simulation of physical forces. It's a sort of first-year college physics level of simulation, but there are in fact stored values for acceleration, velocity and position. So while it's simple by the standards of modern physics simulations, it is a model of Newtonian physics. It may not be very accurate—the programmers only had 8-bit numbers to work with—but it is modeling things in a certain sense. It's not totally fake.

Donkey Kong has no such simulation. The character has a position and two states, and that's it. When you press the right button, the code simply takes the current position and adds a value to it. This new position gets drawn to the screen and becomes the current position and so the motion continues. The player moves at a constant rate if the joystick is held in one direction or another. There is no period of acceleration between standing still and running full speed. Likewise, when the input stops, there is no deceleration. Put another way, Jumpman's speed can only ever be equal to, say, five units per second or zero units per second. There is no in between. Donkey Kong takes the sensitive, expressive input of the joystick with all its states between center and fully expressed and clamps it down to a simple on or off response. Figure 13.6 shows the short attack phase of the movement in Donkey Kong. The movement starts the frame after the input is received, but there is a sensation of a slight attack because it takes some time to push the joystick from off to on.

Mario's horizontal movement incorporates separate values for acceleration, speed and position. When the signal for "left" is received, it applies an acceleration in each frame rather than feeding directly into position. For each frame when a directional button is held down, the acceleration value adds a certain amount to the velocity value. The velocity value in turn tells the rectangle how it should change its position. Instead of the boxy non-curve shown in Figure 13.6, the change in Mario's position looks like Figure 13.7.

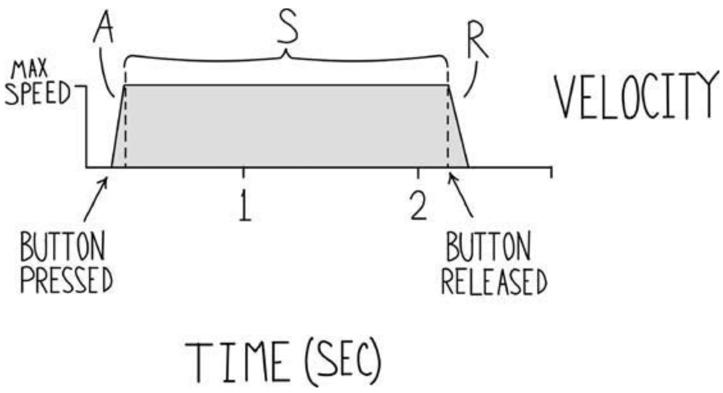


FIGURE **13.6 Donkey Kong's movement is very stiff.**

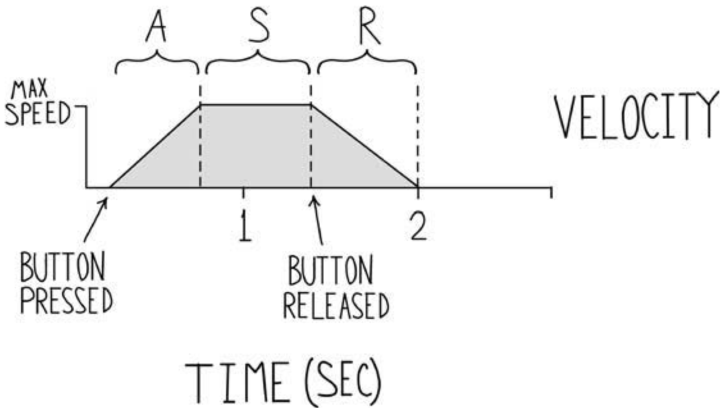


FIGURE **13.7 Mario's movement has a gradual ramp up to maximum speed, making it much more expressive.**

Playable Example

If you're following along in example CH13-1, you can experience what we've constructed up to this point by setting a high max speed value (try 2,000) and turning all the jump values and the deceleration value to zero. If you want, you can also switch the metaphor to rectangles.

Now steer back and forth by pressing the A and D keys. Notice anything? The speed value quickly runs away and becomes way too large. We need to clamp that down, put a limit on it. This limit is the max speed value, and it applies unilaterally to both left and right movement.

The other thing you'll notice at this point is that once set in motion, the rectangle will not slow down or stop. You can reverse direction, counteracting speed in one direction with acceleration until such time that the movement switches from right to left, but you can never come perfectly to a halt again. For this, we need a separate value to decelerate Mario back to a standstill. This is the deceleration or slowdown value. If Mario's running forward and suddenly I stop touching the controller, Mario will slide gently to a halt. The rate at which he comes to a stop is its own variable, unrelated to how fast you sped up. Now, with the deceleration value included, we're getting warmer, close to the horizontal movement feel of Mario.

The last piece of the puzzle involves the B-button. When the B-button is held, Mario "runs." The change is caused by a mapping of the B-button to changes in the simulation. Under the hood, when the game detects that the B-button is held, it changes the values for acceleration and it changes the max speed. When the B-button is held, the rate at which the character will accelerate is increased and his max speed is increased. In this way, the B-button is sort of a state modifier, mapped only to a change in the parameters of the simulation, not to a particular force.

This feel was different from games of the time featuring as it did two different accelerations and two different max velocities for running. The expressive power of such a seemingly simple change is in the interplay between the speeds. The perception of increased speed is created by the contrast. The run seems fast only when compared to the walk. Change both values up or down, and the run still seems like a run. What's important is the relationship between the two values. As long as that relationship is maintained, the impression of speed that comes from the contrast between walking and running will remain. This is very interesting; it is the relative relationships between speeds—rather than the speed values themselves—that seem to be most crucial to the feel.

Another interesting result of this change is just how much expressive power it lends to the horizontal movement. If you're at a standstill and you hold down B and start running, the curve describing the acceleration will be different—it's using higher values. Likewise, because you can press the B-button at any time, it's possible to feather the button and adjust speed very precisely. Try it now in the applet; start running and then try tapping B or holding it at various points in the acceleration to see how many different speeds there are between going from standing to walking, from walking to running and from standing to running. The number of different possible speeds is huge, adding surprising expressivity just with this one small change in response.

Finally, horizontal acceleration in the air is different than on the ground. This brings out a further contrast, one between acceleration speed on the ground, in the air and when running. When you're on the ground, you accelerate at a certain rate, which is different from the acceleration in the air. It's simply a different number that gets applied as a force each frame. As soon as you enter the air state, the horizontal acceleration number changes. Note that pressing and holding B has no effect in this instance—in the air state, all horizontal movement happens at the same rate. Also, this is acceleration, not speed, so you can be running as you jump and still

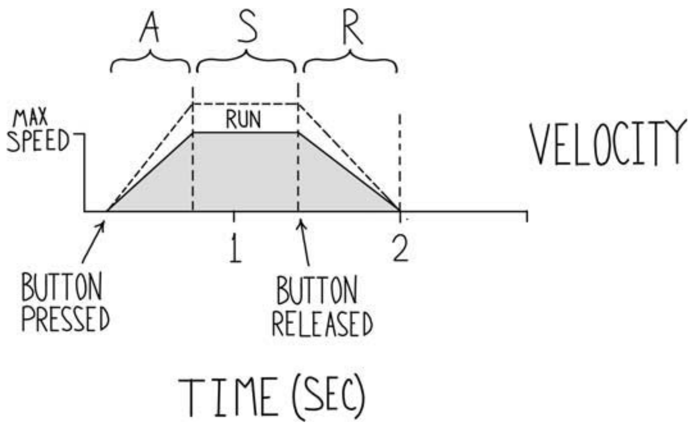


FIGURE **13.8 Horizontal movement in Mario.**

retain that speedy horizontal movement. What's reduced is your ability to modify that velocity by accelerating one way or another.

To sum up, the important values to the feel of Mario's horizontal movement are:

- Acceleration left
- Acceleration right
- Max speed
- Deceleration
- Running acceleration left
- Running acceleration right
- Running max speed—deceleration always remains the same
- Air acceleration left
- Air acceleration right

Note that deceleration remains the same regardless whether the run button is held or not. At this point, we have a rectangle which will accelerate gradually left and right to a maximum speed, slow down again gradually to a standstill, and accelerate faster to a higher top speed if the B-button is held (Figure 13.8).

Vertical Movement

The rectangle's vertical motion, the jumping of Mario, is a more complex series of relationships than those governing his movement on the ground. To start, there is a constant application of gravity. When you're moving left and right, gravity is always pulling the character down. That's simple enough: a constant downward force applied to the character. But this gravity force is variable. At the moment the jump

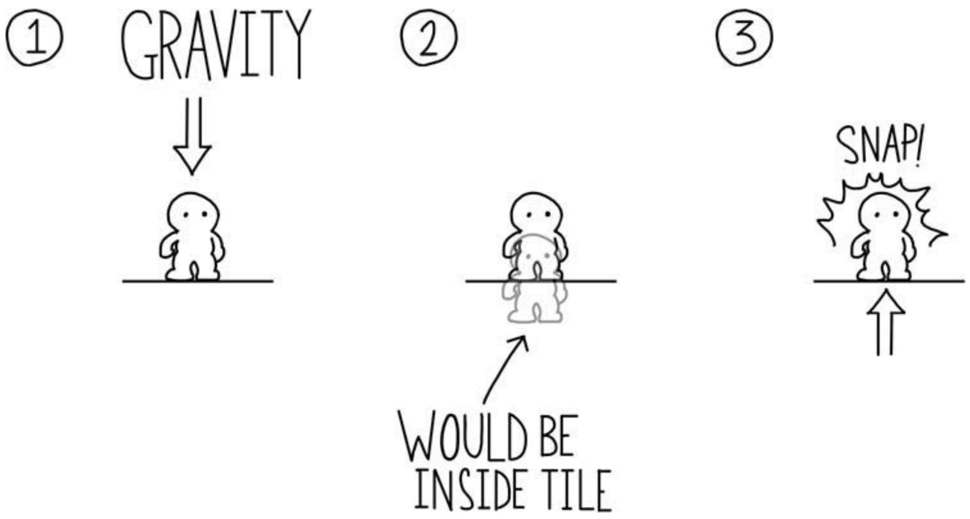


FIGURE **13.9 The collision code snaps Mario back outside of (but up against) any solid object, gravity must otherwise push him through. This happens every frame.**

button gets pressed, Mario is instantaneously imparted with a certain upward velocity, which counteracts the constant downward pull of gravity, launching him into a soaring, graceful arc. This upward velocity is gradually reduced as gravity takes hold again. At the apex of the jump, when velocity reaches 0, gravity is raised artificially by a factor of three, pulling Mario back to the ground with a much greater force than the one he overcame to get airborne in the first place. This artificially inflated gravity has a cap, however, as do all forces in Mario—there is a terminal velocity that will limit its downward motion. In addition to all of that, there is an artificial sensitivity created in the amount of time the jump button is held down. A tiny tap on the button will yield a small hop, while holding the button down extends the jump. This time sensitivity also has a range; there are minimum and maximum jump heights, enforced by limits on the minimum and maximum amount of time the jump will accept inputs.

If this sounds surprisingly complex, it definitely is. Let's step through each of the individual rules and interactions one at a time.

First, the rectangle needs a constant downward force. This is the force of gravity, constantly pulling the rectangle back to the ground. This force is applied all the time, even when the rectangle is pressed against the ground tiles. In each frame, the collision code looks at the position of the rectangle and the forces applied to it. From that, it infers what the rectangle's position would be in the next frame. If it would be inside a tile that's supposed to be solid, it snaps the rectangle's position to be flush against that tile. Though there is a gravity force applied to the rectangle in all places and at all times, the collision code keeps Mario from falling through the world.

Next, the upward force. At the moment the jump button gets pressed, the Y component of Mario's velocity gets instantaneously set to a high value, which counteracts gravity, launching him into the sky. If this force were applied constantly as long as the button was held, the rectangle would fly forever into the sky.

Playable Example

To experience this, try changing the "max jump force duration" value to 10 seconds or so in example CH13-1.

To mimic Mario, the rectangle needs to gradually slow down, losing upward force as gravity takes hold. This is not a hard-coded set of values but is instead the result of two relationships.

The first relationship is between the initial upward velocity and gravity, which reduces that value by a small amount every frame. The initial upward velocity has a limited duration because gravity is always reducing it. After the initial burst of force, there's nothing to keep the rectangle moving up. As a result, gravity will take hold and gradually slow the upward momentum until the rectangle is no longer moving upward. The result, assuming the character is moving horizontally at the time of the jump, is a graceful arc. This also has the effect of making the jump feel immediate, responsive and quick because the moment the input is received there is big, visible response.

The second relationship is between the time the jump button is held and the upward velocity. The jump is time-sensitive. A slight tap on the button will yield a small hop, while holding the button down longer will extend the jump. This time sensitivity also has a range constraining it; there are minimum and maximum jump heights, enforced by limits on the amount of time the jump will accept inputs. In terms of actual measurable response, if you hold the A-button for the full duration of the jump, you'll get a jump that's about five tiles high. If you tap the A-button as quickly as possible, the jump will be shallower, with a height of as little as one and a half tiles. The result is an expressive range of jump height, anywhere between one and a half tiles and about five tiles in height, which corresponds to releasing the button somewhere between one frame and about half a second (Figure 13.10).

The effect is a sort of early out for the jump. The player can choose to release the button early and in so doing accomplish a shallower jump.

At this point, with a time sensitive jump in place that applies the maximum jumping force as long as the jump button is held, the rectangle will jump satisfactorily. The only problem is in the height of the minimum jump: with this implementation, it is still very high. After all, the maximum jump power is still being applied to it for a certain amount of time. So the expressive range between the shortest jump and the tallest is very narrow.

To make this jump feel right, we'll need to artificially clamp the jump force in a way that may seem like something of a hack. Rest assured, though, this is the

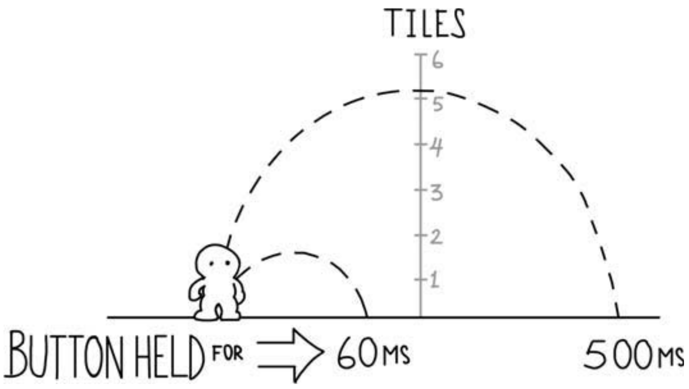


FIGURE 13.10 **Jump height in Mario depends on how long the button is held, but only to a certain point.**

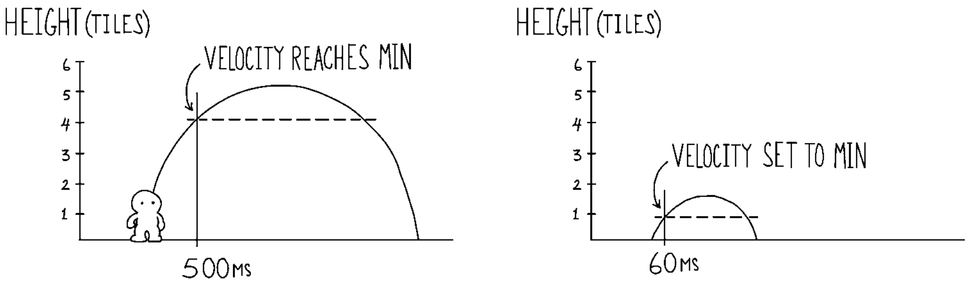


FIGURE 13.11 **Upward velocity is artificially set lower when the player opts out by releasing the button early (thus getting a much smaller jump).**

keystone to getting the good-feeling Mario jump. The hack is this: if the game is in the jump state and it detects that the jump button is no longer being held, the game then checks to see if the Y velocity, the jump force, is above a certain threshold. If it is, it will artificially set the Y velocity to a specific, unchanging, lower value. The value is close to zero but is not actually zero. Weird, eh? Even if you only tap the jump button for one frame or two frames, it still receives the full upward velocity. It's just that when the button is released earlier, it artificially sets the jump force to a lower number before it allows the jump to take its course. The effect is that you float just a little bit more upward from the point of release and always fall with that nice, flowing arc (Figure 13.11). It feels a lot better than the wide variation of arcs you would get otherwise.

An example with actual numbers will help visualize this more clearly. When the button is held down for the maximum time allowed (giving the maximum jump height), the progression of upward velocity might go something like the jump shown in Figure 13.12.

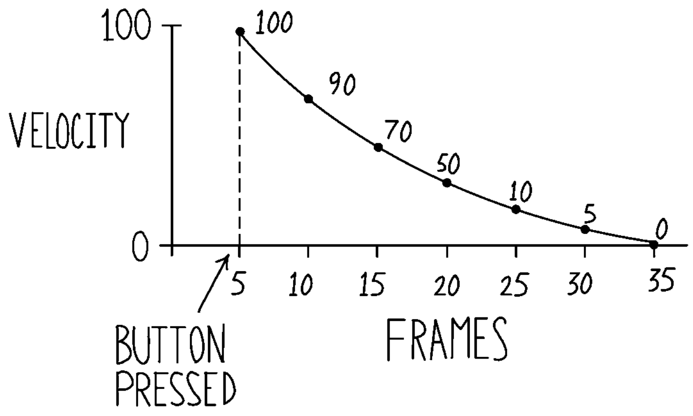


FIGURE **13.12 Falloff in upward velocity over time.**

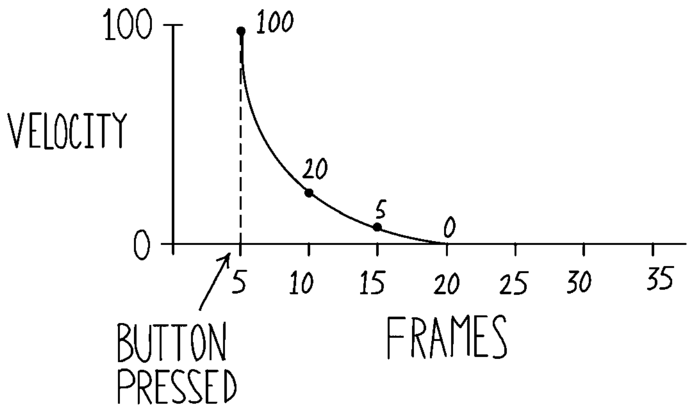


FIGURE **13.13 Falloff in upward velocity when the player opts out early.**

So on the first frame after the button is pressed, the jump force is 100 in the Y direction. This large force will propel the rectangle upward quickly. In the next frame, the force will have been reduced only slightly by gravity, to something like 90. In each subsequent frame, the force in the Y direction is lowered only slightly until eventually the maximum jump time is hit. At this point, the input is no longer important, and the jump force falls off gradually to zero, at which point the rectangle begins to fall back to the ground (Figure 13.13).

Contrast this with the shortest possible hop, where you're effectively getting a jump at the minimum power, which is much lower than the velocity of the full jump. It will give you the full power jump (100) at the first frame, but by the second or third frame, the jump force has already been set to the lower hard-coded value (20).

Instead of setting the value to 90, 80, 70 and so on until the full height jump arc has been completed, it sets it right to 20, the velocity at which the jump stops listening to button input regardless. Similarly, if you hold the button for half of the range of the jump, it might go 90, 80, 70, 60, 50 then to the preset 20. The result is that jump will always have the same arc, especially at the end. The duration of the button just changes how high (and consequently how far) it will go.

Now back to the jump in progress. If you'll recall, the rectangle has only reached the apex of the jump, the point at which vertical component of velocity is reduced to zero (by gravity acting on it over time). Now it has to fall back to the ground. Interestingly, after the tipping point, gravity changes. If you're running along and you press jump, gravity is the same normal value it always was. You're imparted with a negative Y velocity and it sends you upward, temporarily overcoming the weaker pull of gravity. The upward force will only be added for a certain amount of time, though, as gravity gradually takes hold and slows you down until you have no upward force. Once you reach the peak of your jump, instead of just allowing the natural pull of gravity to bring you back to the ground, the gravity is artificially increased, sucking you back down to the ground. It applies this stronger gravity whenever you're falling, whether you've just reached the peak of a jump, walked off the edge of a platform or bumped into an overhead block (which sets your vertical velocity to zero).

Try setting the fall gravity to the same level as the negative gravity and see what happens. The jump seems to take far too long and you begin to feel as though you've been out of control of the character for far too long. The impression of weight is also affected, making the rectangle seem far lighter than it ought to. In a word, it feels weird.

The final piece of the fall is a clamp on the possible falling speed. Just as the horizontal speed has a maximum, so too does the vertical. When you're falling, there's a terminal velocity that is definable by code. It's hard-coded. If you don't limit it, you can really feel the difference. When you jump from somewhere high, you'll get going really quickly before you hit the ground and it'll feel weird. If instead you set this number very low, it will feel like opening an umbrella in a cartoon or something, where you can actually feel the artificial clamp take hold. You can tell it's not the natural arc of the fall. Try both—setting the terminal velocity to something very high or very low—to get a sense of it.

At this point, we have a rectangle that has gravity applied to it and will jump to different heights when the jump button is held down for shorter or longer periods. These periods are limited by maximum and minimum time values, however, which define a certain expressive range of possible jump heights. The arc of the jump will always feel the same, however, because if the game detects that the jump button is no longer held while the jump force is being applied, it will artificially set the value to a lower value. This value never changes, so the end of every jump will have approximately the same arc. When the arc has completed, an artificially high gravity value pulls the rectangle back to the ground quickly and efficiently, decreasing the time during which the player has reduced control of the character and enhancing the perception of weighty, close to real-world gravity (even if the character is leaping around like a flea). Finally, the speed at which it is possible for the rectangle to fall is clamped to prevent it from getting too high and feeling unnatural.

To sum up, the important values to the feel of Mario's vertical movement are:

- Gravity
- Initial jump force
- Minimum jump button hold time
- Maximum jump button hold time
- Reduced jump velocity
- Falling gravity (about three times normal gravity)
- Terminal velocity (maximum falling speed)

Finally, there is one small crossover between vertical and horizontal movement. In Super Mario Brothers, if you press jump while you're moving faster than the normal walk speed, you'll get a small extra jump boost. The initial jump force will be slightly higher, so the height of the overall jump will go up slightly. If you're anywhere between the full, running maximum speed and the normal walking maximum speed, you get a little bit of extra jump velocity. Jumping from a standstill, you can reach a height just under five tiles. If you get a running start (holding B) you can just about get over and land on a five-tile-high surface. This height boost is not commensurate to the speed at which you're moving at the time of takeoff; you get the height boost or not, depending on whether you're over the normal max speed when the jump button is pressed.

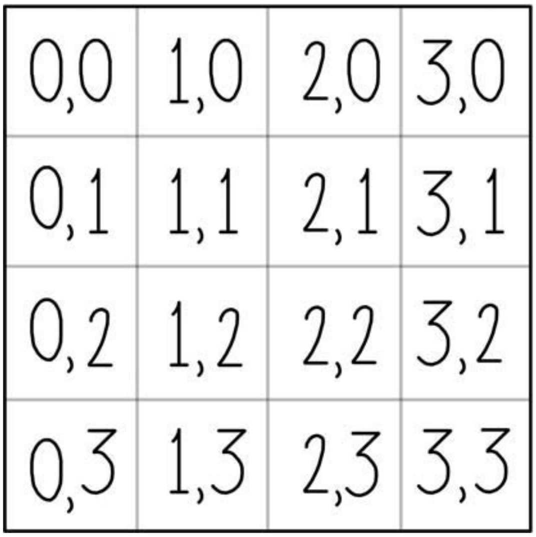
Collision and Interaction

Next comes collision, where Mario's world becomes solid.

Mario is a rectangle, one tile high. A tile is a nice way to simplify the layout, position and properties of things in a 2D game. Instead of having to store detailed positions for each object, we can create a grid of tiles and reference their position with simple two-number combinations. Typically, tile (0,0) is in the upper left hand corner of the screen. The tile below it is (0,1), the tile to the right of it is (1,0) and so on. If we store a list of all these tiles, we can easily find out where a tile is in relation to any other tile (Figure 13.14). If, when you make your list, you specify the type of tile it is—sky or brick or pipe or whatever—you can then check to see whether or not Mario can pass through that tile.

So, to make Mario collide with things, we look at his velocity. By knowing his direction and speed, we can tell which tile he will be in the next frame. If the tile is, say, a brick tile which Mario is unable to pass through, we place him on top of it instead of allowing him to go through (Figure 13.15).

This is pertinent when you have, say, an air tile below Mario in one frame and a ground tile the next. If you know this, you know that Mario should be falling, but



0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

FIGURE **13.14 Tiles in Mario.**

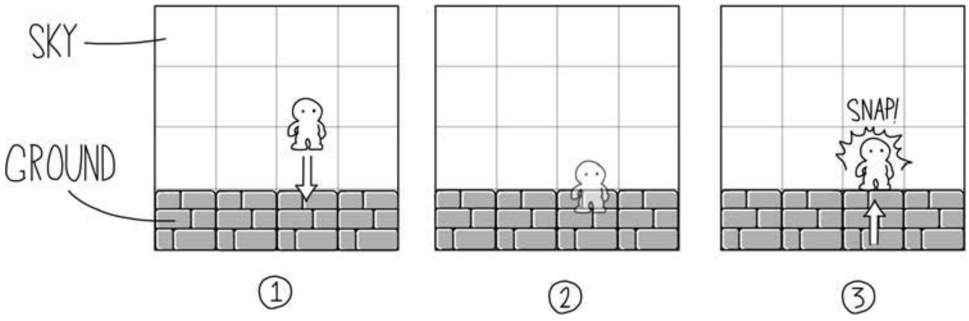


FIGURE **13.15 Instead of going through the tile Mario is placed on top of it.**

that he's just about to land back on the ground again. This is how simple collisions are done. Without getting into unnecessary detail, the feel of a simple tile-based collision system like this is very slippery. Because there is no simulation of friction—the dampening that slows Mario back down again —applied based on what material he's sliding across, the feels is that of a bar of soap sliding across wet tile. He won't get caught or hung up on anything, and the overall sensation is very loose and sloppy. In the case of Mario, this is a huge part of the appeal. This is, of course, not always the case. To create the satisfying carving motion of a car or bike turning is to simulate the friction between tire and ground. But Mario essentially has no friction. Being pressed against a wall of blocks does not slow his jump force; he's free to slip slide across everything.

So now we have a rectangle that slips and slides across the environment, never getting stuck and always feeling perfectly solid. Only the rare spring platforms feel as though they have a bit of give. The rest of the objects in the world are like polished marble, and Mario himself is equally unyielding and smooth.

The next little particular that the rectangle needs is to have its Y velocity set to zero in every frame. As noted earlier, gravity gets applied to every frame, even when Mario is colliding with the ground. In each frame, his collision places him up a little bit, back on top of the block, and in each frame his Y velocity gets set back to zero. When falling, the gravity force will set the Y velocity to something very high (the fall gravity) to pull the rectangle downward. If you don't set that Y velocity to zero when he's in contact with a tile, a great deal of force gets built up and "stored" in a weird way. The collision still keeps you out of the ground but if you walk off the cliff, you plummet down because you have this huge negative velocity built up. It feels really weird. Instead, you want your Y velocity to be guaranteed to be zero when a fall starts so gravity will pull the rectangle gradually and appropriately to the ground.

The opposite case (when the character jumps and hits a tile above) also needs the Y velocity set to zero. If you hit the ceiling and the Y velocity is not zero, you'll stick to the ceiling (as you do in Super Contra). Mario still wants to move up every frame. The collision will stop him from moving through the tile, but it will have no effect on his upward velocity. To get the right feel, you have to set Y velocity to zero whenever a collision is detected, whether it's from above or below.

Another little behavior particular to the original Mario is the lack of height boosts for bouncing off an enemy's back. When you hit a turtle's back, it sets you to the same state as if you'd released the jump button after the minimum amount of time. The rebound off a turtle or Goomba is a tiny little hop, as with the minimum strength jump. This changed in later games such as Super Mario World, where holding down the jump button while bouncing off the back of an enemy would give you a massive height boost, much higher than that afforded by a regular jump.

The only objects in the game that give Mario a height boost are the springy platforms. But you must jump just at the right time. It's wicked hard, though, because the window for doing it properly is extremely small. In the later games, simply holding down jump while bouncing off an enemy or jump platform will give you the extra boost, which to me feels better as an implementation.

To wrap up the movement of the primary Mario avatar, there is one particularly rare special case that's worth mentioning. If you get a mushroom and you're too tall to walk under a one-tile-high brick, you can slide under it. If you're small, you can just run through. But if you're big Mario, you have to build up a head of steam and slide in there. The case the game has to deal with is: what do you do when you stop ducking? In some games, you just can't stop ducking—you're forced to remain in the duck state. Mario enables you to stand up, which puts you in a weird, unique, single case game state where you can't move or do anything (even duck). It pushes you right, locked out of input, until you're free of any collisions. It's a bit stopgap, but I guess it covers a weird case that Mario can get into.

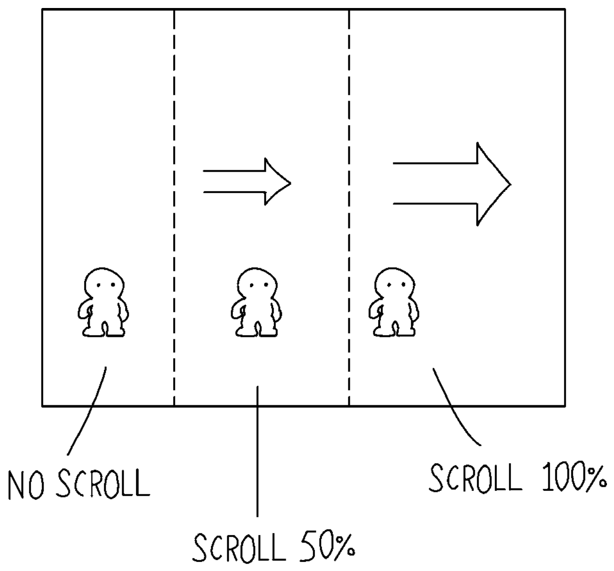


FIGURE **13.16 Camera scrolling zones in Mario.**

Finally, we come to the movement of the camera, which functions as a second, if indirectly controlled, avatar. Clearly, the feel of Mario includes the motion of the camera and the game would not feel quite correct unless the camera moves properly with the avatar. First, the camera moves only to the right. Once it has moved right, it cannot be moved back to the left. To me, this feels a bit oppressive and it's unclear whether this was a technically motivated decision or a design one. Either way, it has the effect of cramping up the screen and encouraging constant forward motion. If the player goes to the left, the fact that the camera does not move feels abrupt, halting and unyielding. When moving to the right, the screen scrolls at the same speed as the character. The only small, important thing for feel is the small zone extending from about 25 per cent of the screen width from the left edge to the center. Inside this zone, the scrolling speed is reduced. The effect is a gradual, though rough, speeding up of the camera as the character accelerates from a standstill (Figure 13.16).

And there you have it! This is the simulation that creates the feel of Mario in all its dirty, exhaustive details. What strikes me about it is just how seat-of-the-pants many of the decisions and particulars are. Stuff like manually setting the Y velocity to a lower, artificial value when the jump button is released early and swapping out the normal gravity value for one three times higher seems like it would have the opposite of the intended outcome. Why do those particular changes make the game feel better? I'll attempt to address such questions at a more general level in the Principles of Game Feel section. For now, though, I recommend fiddling around with the final, composed applet a bit. Really dig in and change some of the parameters.

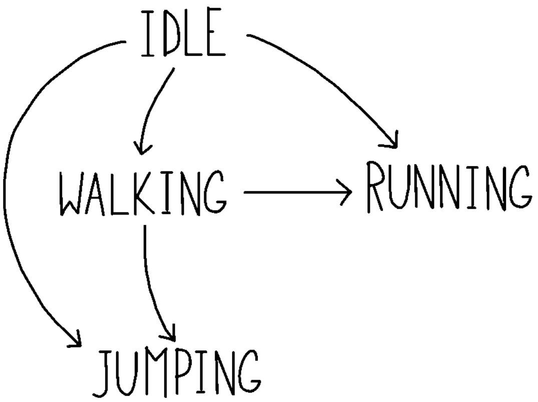


FIGURE **13.17 States in Mario—stuck and dead are special cases.**

At the end of this chapter, I've listed the tunings for Super Mario World and Mario 2. See if you can arrive at them on your own, just by tweaking the numbers.

States

States are the final piece of the Mario feel puzzle from the standpoint of someone trying to construct a perfect-feeling Mario stand-in. By state I mean a specific set of instructions about how the game will respond to input. When a game has more than one state it means that the same input might result in a different response from the game, depending on what the character is doing at the time of the input. A simple example from Mario is illustrated by different horizontal acceleration in jumping state versus the running state. In the running state, the acceleration is a very high value, speeding Mario up very quickly from a standstill. In the air state, that value is drastically reduced. Effectively, the change in state maps the same input to multiple responses in the game. As long as the player has a way to conceptualize a change in state, such as when Mario leaps into the air, the fact that the result for a particular input has changed is not jarring or distracting. In fact, it offers more expressive potential.

The states in Super Mario Brothers are idle, walking, running, jumping, stuck and dead (Figure 13.17).

For the record, this is how things are organized under the hood in Mario. More or less. As long as the relationships are maintained, the implementation is more or less irrelevant, but structuring things this way certainly makes it easier to get the desired feel.

Context

Our rectangle is now happily sliding around. It feels great to leap around and run and change directions. Right? Well, perhaps. Returning to example CH13-1, run around a bit. Notice anything weird? Right, there's nothing in the level. It's an endless field of blankness. Time to add context.

A level design trick that originated with Super Mario Brothers or was at least used heavily in it is the "Fortune Favors the Bold" approach. Running full speed to the right and attempting to quickly adjust to changes in the environment makes the game much easier by virtue of the level design itself. The levels are set up to encourage (and in many later levels, necessitate) this kind of bullish charge-ahead play. It's simply how the level is laid out: the jumps and obstacles are easier when the player is moving at the maximum speed possible. Since that speed was predictable once the mechanic was complete, it was possible to design the levels to match.

It seems to me that this was the intended experience of the game, as a whole. Even the camera avatar's behavior seems to encourage this behavior, by constantly blocking the path behind the player. You can't go back, the game seems to say, so you might as well go forward as quick as possible.

As a general rule, the spacing of the objects in Super Mario Brothers is about four tiles high. To accurately recreate the feel of Mario, you need a level that's built this way. This is because while the character can actually jump close to five tiles in height, jumping to an object four tiles high is much easier and drops the character onto the platform just at the shallow part of the jumping arc. Jumping to a platform that's just below the maximum height of the character's highest jump just feels better.

Playable Example

To experiment, try placing platforms at various heights in example CH13-1. There are platforms of various heights to jump up to. Notice how the four-high platform matches the jump best. Like Goldilocks' porridge. Best reference ever.

The relative imprecision of the Mario mechanic makes it necessary to give the player plenty of room for overshoot, in both the horizontal and vertical directions. Part of that is making most platforms' height about four tiles apart. The other part is stretching all platforms in the game horizontally. Now try removing some sections of ground in example CH13-1 to see what happens if you construct a level that has mostly single-tile wide platforms to land on. Again, this hooks into the overall design goal that the player should be running full speed to the right at all times. Tiny, single-tile platforms encourage a more plodding approach, where each jump is considered carefully before it's executed.

Another aspect of the context that plays heavily into the tuning of the rectangle's motion is the movement of enemies in the game. This is something we don't normally think of with respect to level design, but the motion of the AI characters actually has a huge bearing on the way the mechanic will feel. Their motion dominates certain areas of the screen space, changing the player's ideas about the spatial topology. An area enclosed by two tiles suddenly becomes a bit of a death trap if there are two turtles wandering around in there. Similarly, jumping up a set of stairs becomes an entirely different interaction when there's a hammer brother at the top. For a striking example, look at the first level in which 10 (the floating cloud guy who tosses those red spiny bug-turtles) appears. Most of the level is flat with almost no skillful jumping required, but the entire thing feels hurried and oppressive because of the ceaseless shower of enemies raining down from on high.

Enemies, and this is no accident, move along the ground at approximately the same rate as Mario's maximum walking speed. You can sync up with the speed of the enemies' movement if you drop out of running. The result is that you can jump into the midst of a group of enemies, walk temporarily for a precise amount of time and then jump out again without feeling too out of control and without plowing into the enemies themselves.

Challenge is also defined by context, as it is in most heavily spatially oriented games. In the later levels, the jumps become a bit wider, requiring more specific, precise landings. To further ratchet up the difficulty, you must do a large number of these precise jumps in rapid succession. At the same time, there are far more enemies. Where once there was one Goomba, there will be three in a row followed by two turtles. Then things like bullet bills and hammer brothers begin to appear, and the meaning of even the simplest jump and motion changes. It may seem an obvious point, but it's interesting to examine the ways in which challenge is constructed via the addition of enemies. There's almost always an optimal path that, if the player's running full speed, will take him or her through unscathed without too much hassle. The game feels like a course to be run and perfected rather than like a space to explore slowly and methodically.

To get the feel of Mario, you need blocks that are four tiles high and are spaced far apart horizontally. Pits in the ground are in a range from two blocks wide to six, with six feeling quite risky to jump across. The difficultly ramp up happens based on adding more of these wide, difficult jumps. They require a great deal of precision and force the player to do them in rapid succession. In addition, the later levels become littered with increasing numbers and increasingly fast and unpredictably moving enemies. These enemies serve, by their motion, to dominate areas of the screen space and make it feel more unsafe and oppressive. In the game's later levels, no place feels safe, and it seems a mad dash of survival to reach the end.

Indeed, there are lots of one-tile-wide blocks on 8-1 and 8-2, as well as lots of single-tile pits you can run across. In addition, many pits get as wide as five or six tiles, which leaves very little margin for error. Technically, Mario can jump 10 horizontal tiles at a full run, but it's almost never used in the game because it's so difficult to do.

Polish

Up to this point, what we have is a bunch of rectangles moving and sliding around. To examine the various polish effects, let's turn on the character-based treatment. So as not to infringe, I've created "Scarfman" as a Mario stand-in.

Playable Example

Open CH13-2 to see the rectangle replaced with a character and the various tiles and enemies given some kind of representation.

First, animation effects. The character has a run cycle. When the motion of the rectangle is activated, the visual representation now sitting on top of it plays back a series of frames. It's a very short and simple series of frames, but it's an animation nonetheless, and it conveys some new and different things to the players about the nature of the object they're controlling.

One of the crucial parts of the feel of Mario is that the playback of the frames of this running animation are synced up perfectly with the simulated motion underneath it. This was and is a big deal where feel is concerned; even today many games don't successfully accomplish the sense that the avatar is truly and accurately representative of the simulated object underneath it. This is typically called "foot slip" by animators and is a particular problem in video games because of the unpredictable, participatory nature of avatar movement.

The fact that Mario really nailed this relationship between animation and avatar movement was a big deal, even in this primitive, 8-bit context. The perception conveyed is that this is a little guy running along the ground and his feet are planting at each step because the speed of animation is perfectly matched to the speed of movement of the object.

If the underlying simulated object is moving faster or slower than the animated object, it's very easy to pick up on that discrepancy. It's a very subtle clue but it's something that humans are very good at observing. We have a lot of practice observing and coping with our immediate physical surroundings at this tactile, interactive level, so as soon as something doesn't match up, it becomes immediately obvious. The net result is that, in the player's mind, the animated character and the moving object beneath become separate entities, which makes it more difficult for the player to engage in and believe in the reality of the game world. Something is lost because you can see behind the curtain, see the simulation.

Another important animated effect is the little slide that Mario does when he's in the process of reversing direction. If you reverse direction while Mario's running, he puts his hand up and assumes a little sliding pose. In the case of the Super Mario Brothers, this is not a separate animation that gets played back. Rather, when you apply the acceleration in the opposite direction, the character slows down. This is another consequence of Mario's simulated approach; instead of having a canned set of frames played back for, say, a foot plant and direction change, the natural force of the acceleration gradually counteracts the current velocity and Mario slows to a halt, reverses direction and slowly speeds up going the other way. The animated effect occurs if and only if the player is holding down a direction which is opposite from the current direction of the character. So the underlying simulation has not changed, but the animated effect on top of it is emphasizing and enhancing the perception of what's going on physically with the simulated objects. The character is running in one direction, the player presses the other direction, the character goes into a slide until the direction change is complete, and then the character resumes the run animation (albeit at the lowest speed, speeding up gradually to match the speed increase of the simulated rectangle). To put it another way, the animation effects are there to enhance and emphasize what's happening with the underlying simulation, rather than wagging the dog the way the animations in, say, Prince of Persia did.

Another animated effect that's crucial to the feel of Mario is the jiggle of bricks. It seems like such a silly thing, but when you're small Mario and you hit a brick from underneath, it sort of jiggles. It jiggles in a very satisfying, very cartoony sort of way, but it adds a lot to the sensation of interaction. It doesn't actually affect the simulation—it doesn't move the brick, changing its permanent position—it's just a layered on animated effect that gives an impression of mass to the character. This is especially true in the contrast between hitting a brick when you're small as opposed to when you're large Mario. Small Mario causes the brick to seem loose and jiggly when it's hit, but that interaction also tells you that small Mario has a certain amount of mass. If he can jiggle a brick loose, he's striking with some force, clearly. When you're large and you can smash the bricks, it shows you that the larger Mario is quite a bit more massive. It shows you that where diminutive Mario simply loosened the brick, big Mario has the force to smash it spectacularly, causing brick detritus to rain down.

Playable Example

Check out how the feel changes if you turn off both of these effects in example CH13-2 to see just how much they sell the notion that this is a physical, malleable world.

Visual Effects

As we have defined them, there are very few visual effects in Mario. There was very little processing power to work with on the NES, so there couldn't be sprays of particles everywhere to emphasize every interaction. The trend over the years has been toward more and more visual effects in the Mario game—the recent New Super Mario Brothers and Mario Galaxy have almost ludicrous amounts of particles flying everywhere all the time. In the original Mario, everything seems clean and smooth. There aren't even modest little puffs of dust or smoke when Mario enters the slide pose to change directions. It seems as though every surface is pristine and smooth, without a hint of dust or gravel. The only visual effect of note is the broken brick particles that spawn and fly down when a tile is smashed. Again, though, look at just how much is lost if this effect is removed. It just doesn't feel satisfying to smash bricks when they simply disappear. Removing this effect removes one of the few, highly important clues the player has to derive notions about the physical nature of this world.

Sound Effects

The sound effects in Mario are paramount. I've replaced them in my demos to avoid infringing on Nintendo's intellectual property, but note their nature. The rising, slide-whistle noise for jumping roughly matches the height change of Mario as he flies upward, further harmonizing with the motion and the sensation of holding down the button in emphasis to get a higher jump.

There's one collision noise—when Mario's head hits a block or when a fireball hits a wall—that sounds like a large rubber band being tweaked. It varies slightly in pitch to stay fresh-sounding, but the impression it conveys is one of a silly, rubbery world. It fits very well with the jiggle of the blocks when they're hit by small Mario and in general convey a sense of jiggly, bouncy movement. This would seem to be selling a different impression than the smooth, frictionless collision simulation, but because of the jiggle of the bricks, it matches well and makes the world seem more alive and the physics more exaggerated than the more staid collision interactions would suggest.

The brick breaking sound is particularly satisfying—it truly does convey the sense of a crumbling stone object, even with the limitations of the NES sound board.

Metaphor

Being as iconic as it is, it's a little weird to look at Super Mario Brothers with an eye to examining how the metaphorical representation it presents affects our expectations about how things will behave and act in its world. But humor me here, let's take a poke and see what we can see.

First, let's give Mario's treatment a place on the three-axis scale between realistic, iconic and abstract. Obviously, Mario's not realistic. The treatment, such as it is, is far toward the iconic side of the diagram and indeed begins to creep up toward pure abstraction. It's very surreal. What, for example, is a Goomba? What does it represent? A turtle in Mario looks something like a turtle, but it's clearly not attempting to meaningfully convey turtleness, if that makes sense. These turtles are fast-moving, dangerous creatures. In general, the creatures and objects represented have very little grounding in meaning or reality. Their meaning is conveyed by their functionality in the game, which is to present danger and to dominate areas of space. They're not abstract shapes and lines either, though. They're definitely meant to be creatures of some kind who obey their own bizarre rules of physics and behavior. They simply tend toward the surreal. There's very little meaning imbued in these objects other than the function they serve in the game.

What does that imply for the way that we expect things in this world to behave? Well, we're not grounded in expectations about how these things should behave.

We don't expect that because Mario is slightly portly that when he comes up to a pipe or some wall that's taller than he is that he'll have to heft and sweat his way laboriously over the top. Because the treatment is so surreal, we aren't bound by that expectation. He's not a photorealistic representation of a plumber. He doesn't look like the plumber who came and cleared that hair clog from your shower. We can accept that he leaps like a flea.

In its abstractness and surrealness, Mario plays very well into the type of physical interactions and movements that it sets up. Mario flies through the air like a flea getting this gigantic, spontaneous upward force and yet needs no wind up, no anticipation, no pole to vault with. It's very organic and expressive but it has very little to do with the way that things behave in our own physical reality.

But that's okay because both the metaphor and treatment are surreal. The abstract and surreal motions of the objects and the way things feel and function don't seem odd. Even the interactions between objects, which so often seem like a block of ice sliding across a gymnasium floor, fit in just fine because of the dreamlike metaphor and lo-fi treatment. The metaphor is setting up very few expectations, so all bets are off.

The one place that Mario does actually lean into expectations is in the use of an iconic human to represent states. Because Mario appears identifiably human, we can look at him and say ah, yes, he's on the ground running now, or hey, he's in the air jumping now. When he's on the ground and running along, it's apparent to the player that he's in a different state than when he's in the air or when he's swimming. It's easy to accept that when he's in the air, he has less control, because it's obvious to the player that a different state has been entered. It's odd that there's any control in the air, as this is not the way that things work in the real world, but the visual cue effectively conveys the change and maintains a sort of logical cohesion, even if it is rather surreal. You can definitely tell that Mario, as an iconic human, is in a different state of being when he's in the air than when he's on the ground. It's a nice visual metaphor for changes in state. It uses the fact that he appears human to tie in the logic of the states and how they function.

Rules

At the lowest level, there are a lot of really interesting rules about enemy interactions that give the player clues about the physical nature of Mario and the world in which he exists, which ultimately change the feel of interacting with that world. For example, Goombas are weaker and less substantial than turtles. A Goomba is killed in one stomp and gets wiped out of existence. A turtle can be stomped and killed in one hit but leaves behind a shell. The fact that the shell remains seems to indicate that it is more massive than the body of a Goomba. A shell can't be destroyed as easily as a Goomba can be stomped. By the same token, winged turtles seem more powerful and massive than those without. You have to stomp them twice: once to knock their wings off—the wings are not very well attached, apparently—and once to knock them out of their shells. There's this sort of hierarchy of powerfulness amongst the creatures, from Goomba to turtle to winged turtle. Bowser is the most powerful creature of all; he can't be stomped. You have to drop him into lava.

The other thing all these interactions tell us is that Mario himself is quite massive. These creatures are all about the same size as him, but he can stomp anything in the game to death with relative ease. One assumes that his apartment is free of cockroaches.

At the medium level, there are three power-ups that have an immediate effect on the spatial topology. You see them and you want them for the immediate benefits they convey. So you focus in on grabbing that star, mushroom or fire flower. You're going to go out of your way to get it. Unless, of course, you already have the fully powered up fire flower, in which case the flower and mushroom become meaningless and can be ignored. These are temporary effects but each changes the feel of the game significantly. Going from small to big, you can smash bricks and you can be less afraid of enemies because if they hit you, you simply turn small again. Fire flowers enable you to run forward with impunity, dispatching enemies left and right without having to stomp them. The whole feel and flow of the game are altered because you no longer have to fear most standard enemies. They no longer dominate certain areas of space, so the game suddenly feels more open. The star is the ultimate temporary power-up, enabling you to run through enemies at leisure. With the star, the challenge is temporarily reduced to jumping exclusively. This feels more open, more free.

Finally, at the highest level of long-period rules, there are 1-ups and coins. Score in the game is mostly irrelevant, a throwback to an earlier age of arcade gaming. I never pay attention to or try to beat my score in Super Mario Brothers. Extra lives, however, I'm very interested in. Super Mario Brothers is a difficult game and you are given only three lives at the outset, leaving very little margin of error. When I see a 1-up mushroom, then, I get very excited and immediately focus all my attention toward attaining it. But the desire is tempered by the knowledge that if I die in the attempt, the effort will have been wasted. The reward is almost directly proportional to the risk. It feels like walking on eggshells. By virtue of the rules—lives are the rarest commodity and there are very few of them—it seems like the highest possible reward, so I'm willing to undertake a huge risk to get it.

Coins give a low-level sense of constant reward. Because collecting 100 coins gives you a huge reward, an extra life, it always feels like you're doing something useful as you collect them. Useful, but mundane. It's not like the excitement of the fast-moving extra life mushroom. You might go a little out of your way to collect a coin, but you wouldn't risk dying over it.

So in order for a game to feel like Mario, the metaphor should be surreal. It doesn't have to be an Italian plumber running around huge pipes and dealing with abstract, surreal monsters that vaguely resemble turtles and bullets, but putting Mario with the movement he has on a street corner in downtown New York will seem a bit off. Similarly, a photorealistic treatment will clash with the surrealism of the movement and interactions. To have that Mario feel, the treatment should be iconic, bordering on the purely abstract. Again, it doesn't have to be Mario specifically. As Scarfman shows, though, it ought to be a little guy.

Summary

Bet you'll never look at Mario the same way, eh? I know that I certainly don't after such an in-depth examination. The other major takeaway here is that game feel, even in a seemingly simple game like Super Mario Brothers, is really freaking complex. All the tiny little decisions that meld together to comprise the feel of Mario boggle the mind. Especially in the area of simulation and response to input, there are a surprising number of small but important decisions. This examination gives us a nice vocabulary for addressing things like whether the game keeps track of acceleration and velocity or whether it's simply tracking and updating position and how that will change feel.

At this point, then, you should have a very clear idea of the depth of detail that goes into creating a good-feeling game. If you're creating a game from scratch, you must be prepared. Keep your mind open to the possibility of changing any part of the system with the goal of improving the feel, the perception. As Mario proves, even things that seem hacky, such as artificially setting a jump velocity in the middle of a jump, may turn out to feel better than a more pedantic approach to simulation.