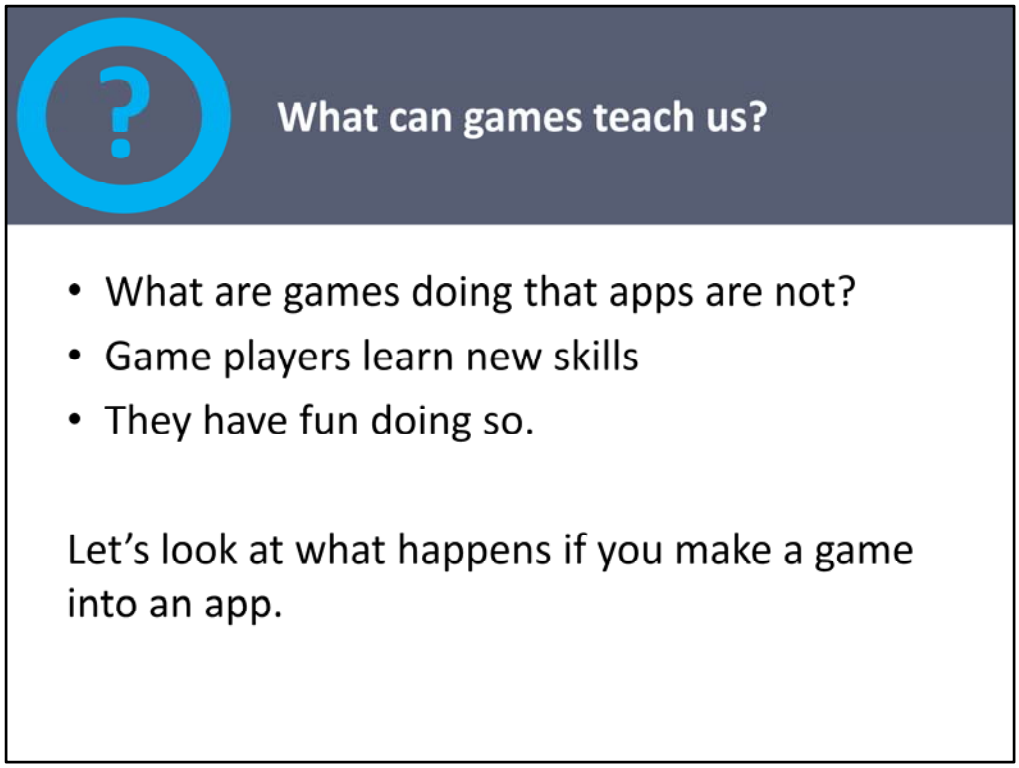
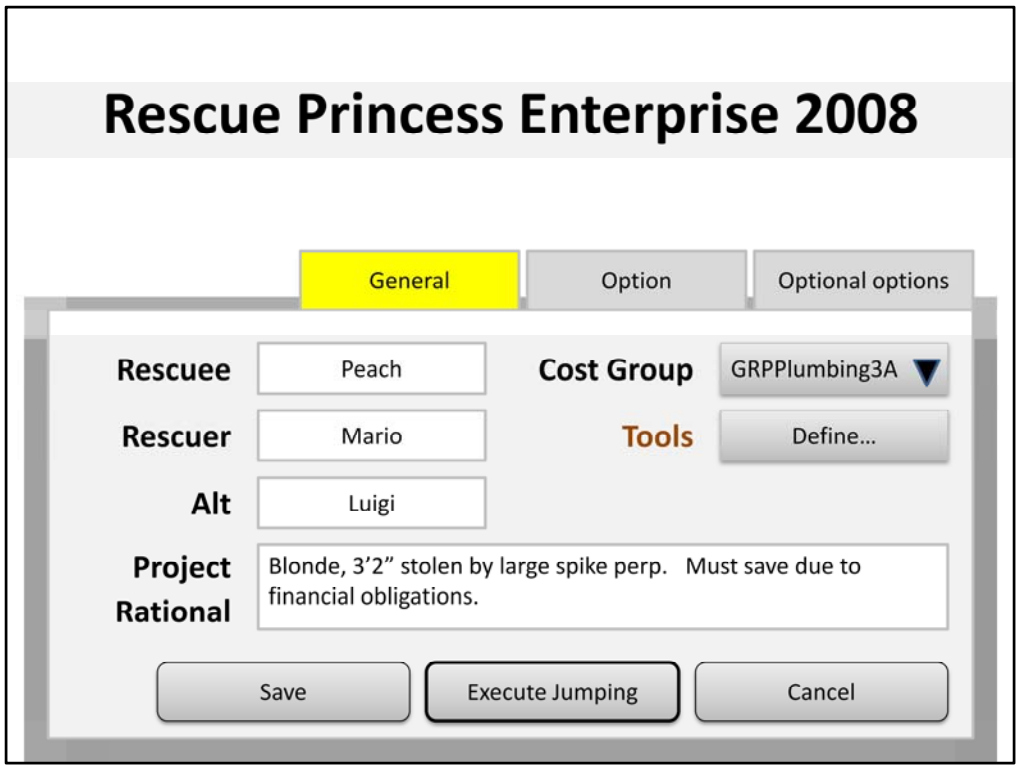
Target audience: Interaction designers, Introductory game design

This talk is about building learning and fun into your applications. If you've ever wondered how games work and how they can help you build better apps, this is the talk for you. Along the way, we'll discuss a new philosophy of interaction design that is already creating a major competitive advantage for innovative application developers.



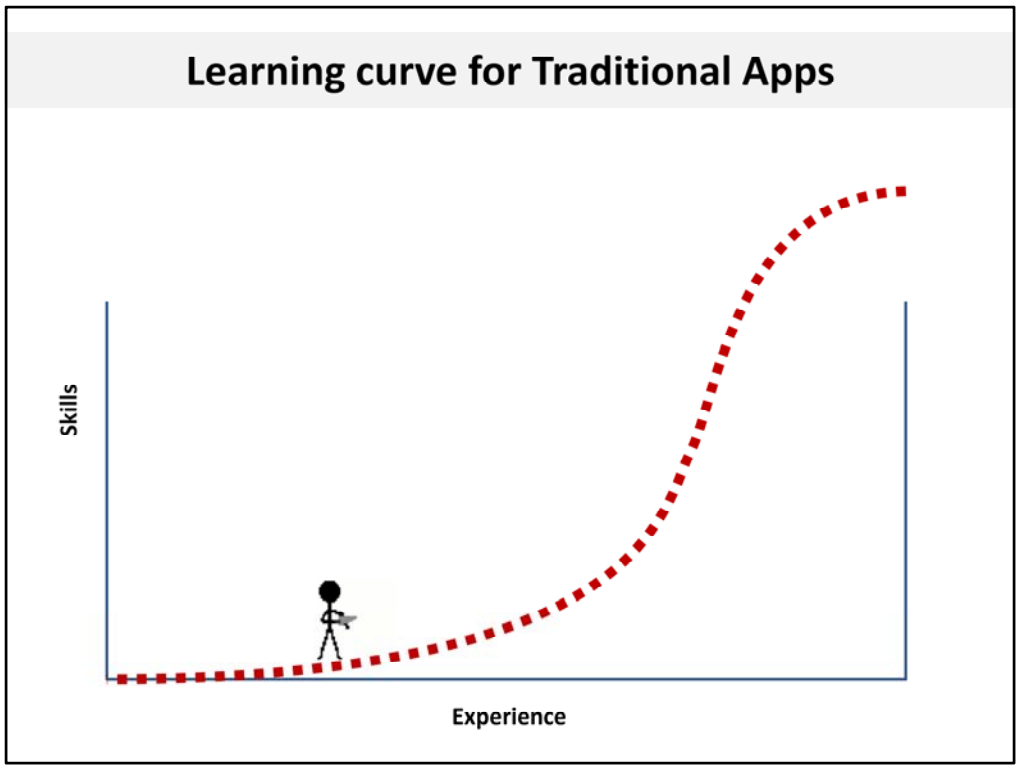
Games are really good at rescuing princesses. I've yet to see an application that does the job half as well. So in my quest to explain how games are different than apps, I decided to turn things upside down.

What follows are my attempts at making a princess rescuing application.



Here's my first attempt: Rescue Princess Enterprise 2008. You have everything and the kitchen sink necessary to rescue a princess. There's even an Execute Jumping button.

This is an example of a typical app. It's defining characteristics is that it focuses on giving the user utilitarian tools to get the job done. The result is a focus on features. Have a problem? Add a feature. The result is this massive, complex Swiss army knife. If you want to do something, it is in there. But you need to know how to use it.



Here's the learning curve for your typical app. On one axis is the user's cumulative experience with the app. On the other are the useful skills that they've accumulated.

Skills are tools that you know how to use. The Jumping button in our mockup was a tool. Knowing when and how to use the jumping button is a skill that involves a complex functional mental model inside the users pulsating brilliant brain.

Notice it takes a while to build up competent skills in a traditional app. There is so much complexity that comes from feature piled on top of feature, it is easy to get confused. You can spend 12 months gaining a basic level of competence in Photoshop.

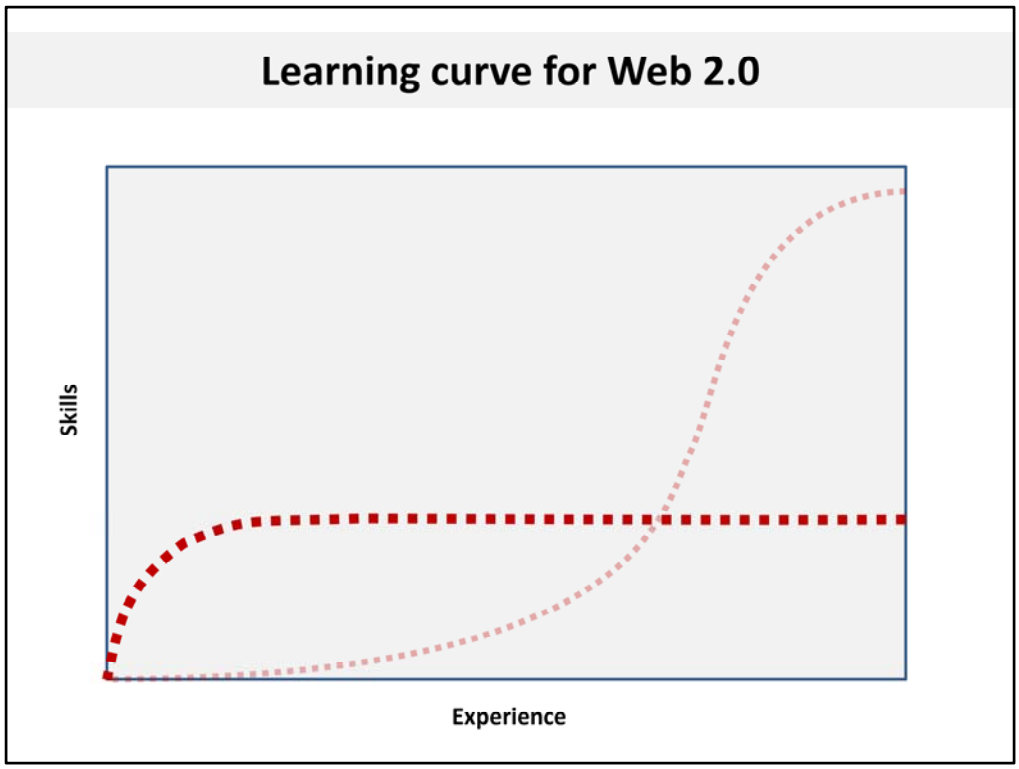
But the good news is that there is a life time worth of depth.

This initial period of learning is very frustrating. You lose massive numbers of users. I took 3 years to learn Photoshop on my own. The basic metaphor just made no sense to me when I used the trial. In this modern world where apps need people to pick them, up try them out and fall in love, this long learning curve is often the kiss of death for a new company.

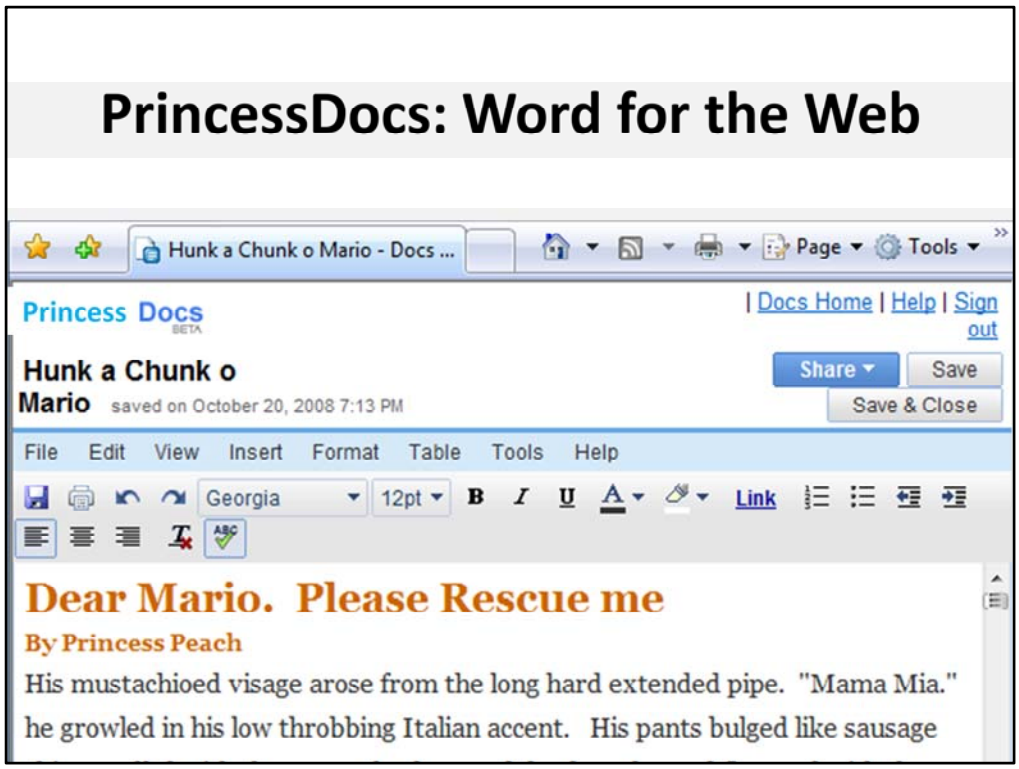
So here's my second attempt. Web 2.0 is the future, right? Here is Rescue Princess 2.0. There's a single button. You press it and you rescue the princess.

Studies show that users will use 80% of the features in a program once or never. At a certain point, those extra features hurt more than they help. Web 2.0 has a couple of important ideas
- **Remove extra** features
- Focus only on the **task** at hand.
- Use skills that people **already know**. Don't force them to learn anything new!

There was a lovely usability study by Jakob Nielsen (http://www.useit.com/alertbox/20050711.html) about how to make the most usable Flash Scrollbar. The results: Make it look and work like a standard scroll bar. People already know how to use scrollbars so innovation just confuses them.

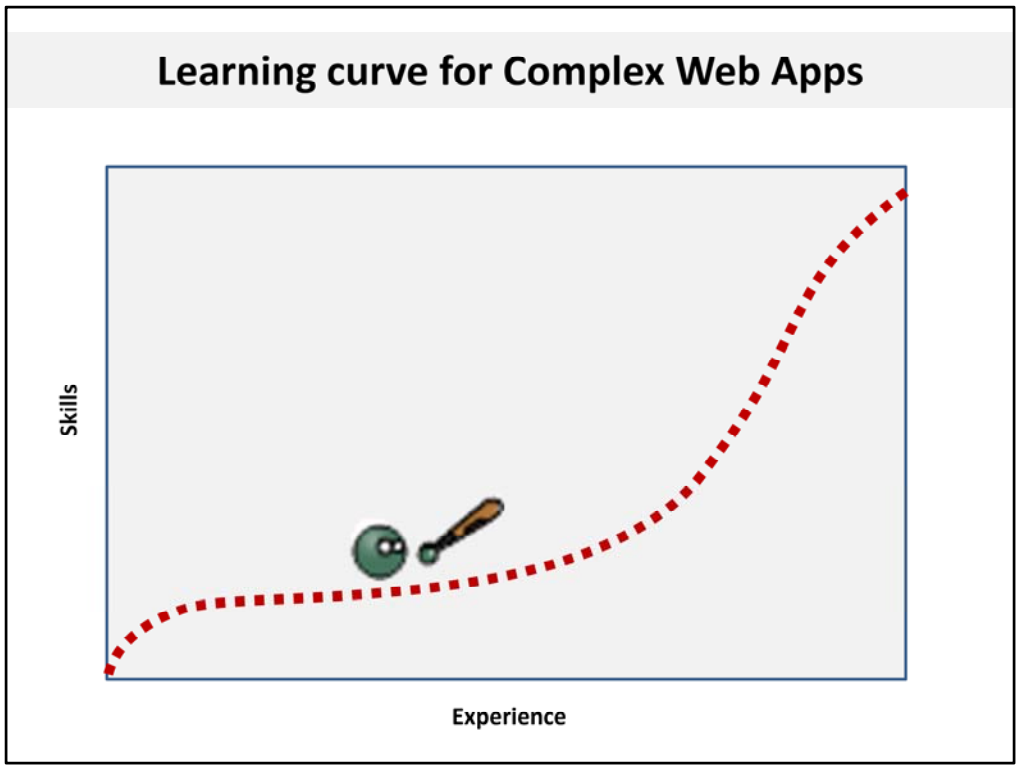


Here is the learning curve for your typical Web 2.0 app. The emphasis on pre-existing skills yields are apps that you can start using quickly, but aren't really all that deep. You can start using Digg in a few minutes. But there isn't a lot of depth beyond reading and recommending news.



What happens when our cute little web apps grow up? This is “PrincessDocs.” It is a web app, but it has been extended with has menus and sharing, spell checking, version control and about a hundred other little features.

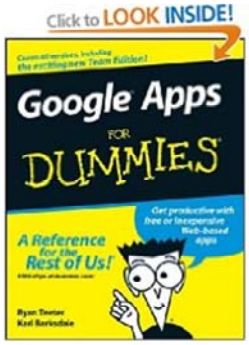
It turns out that Princess Peach likes to write Mario Fanfiction while she is waiting to be rescued. I’m not going to leave this one up on the screen for very long because you might actually read it.



Complex web apps are more useful than simple web apps. And they are easier to use initially. We gain a little bump in skills initially because our app is a clone of Word and most people know how to use Word already. However, it still takes people a large amount of time and experience to learn the more advanced features.

We still have the dip where users beat themselves over their head in frustration as they fail to take full advantage of the full capabilities of the tool.

The Sad Lesson of ~~Google~~ Princess Apps



There are Dummy books for websites

- Going back to the old learning curve.
- Current usability model doesn't scale well to complex skills, only for smaller pieces.
- Hacks exist. They help only a little.

I just saw that a book was released the other day that teaches people how to use GoogleDocs. The more complexity that you add, the closer you get to something like Word. When we add 'features' we hurt learnability and end up turning off users.

Hacks:
- Segmenting features by user skill level,
- Layering less commonly used or expert features so they are out of the way.
- Creating a unifying UI metaphor that lets users understand new tools more easily.
- Elegant information architecture and clean visual design.

The usability dilemma

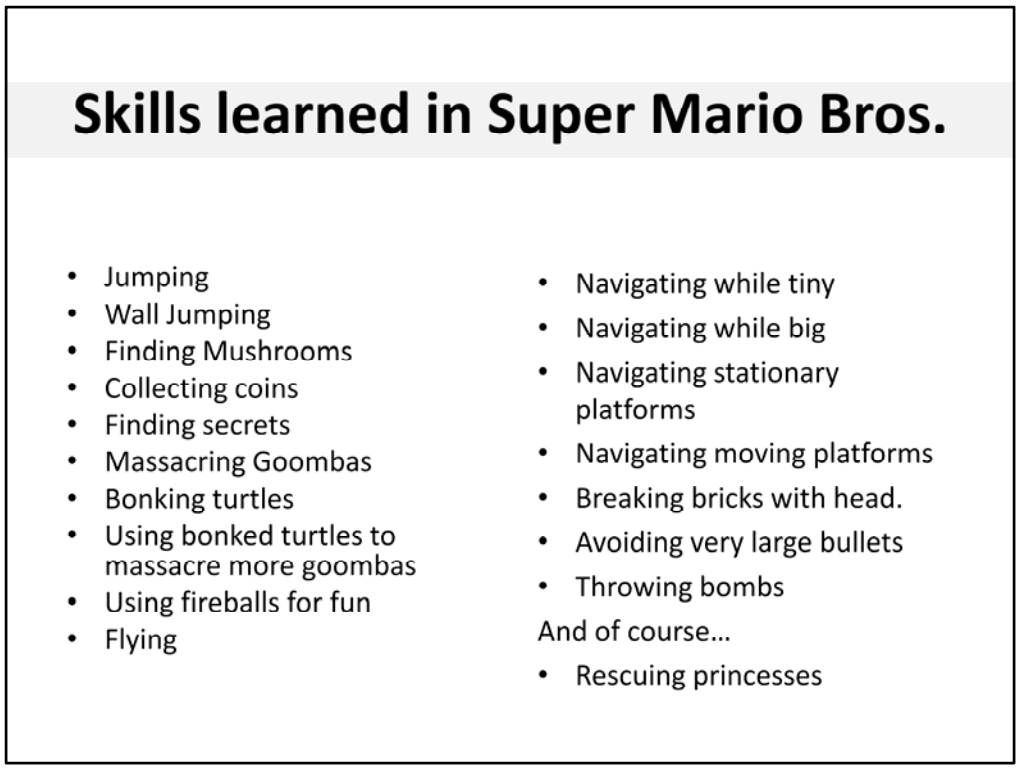
Pick one?

- Simple and easy to learn.
- Complex and painful to learn.

I ran into a bit of a dead end building a Princess rescuing application. Let's turn to the original Princess Rescuing game and see what it can teach us.

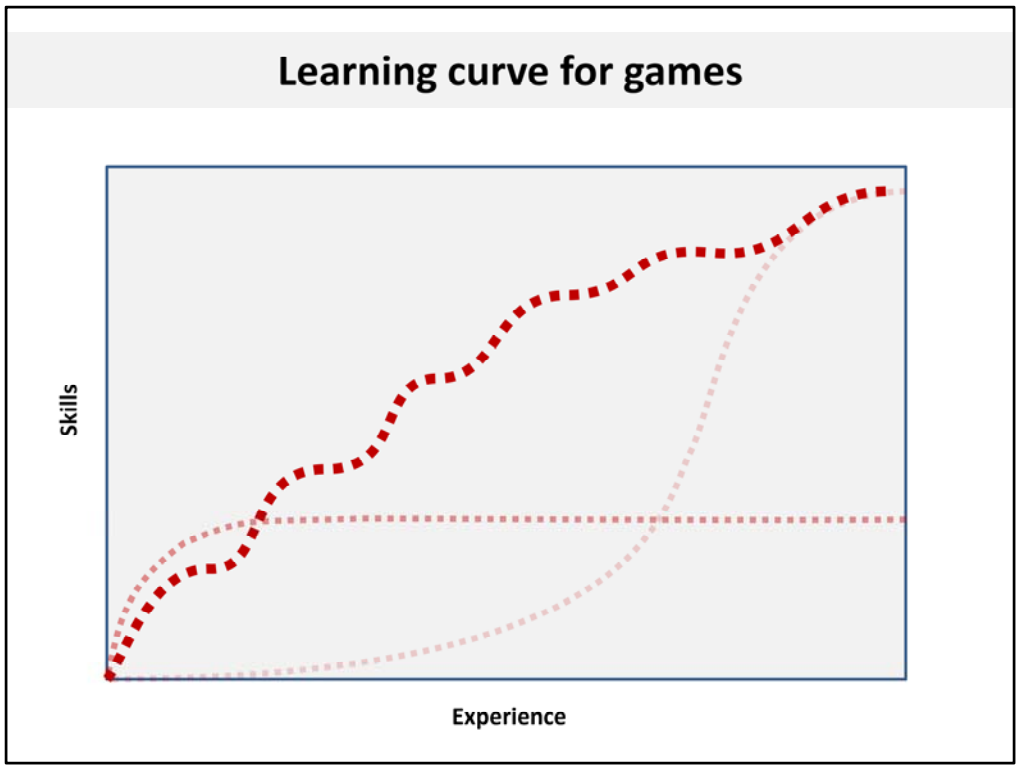
Here's the original. How many of you have played Super Mario Brothers? You are a little plumber and you need to save the Princess. She's trapped behind all sorts of crazy obstacles. It's not so different than the typical workday for many of us.

This is a game that teaches you all the skills you need to rescue a princess.

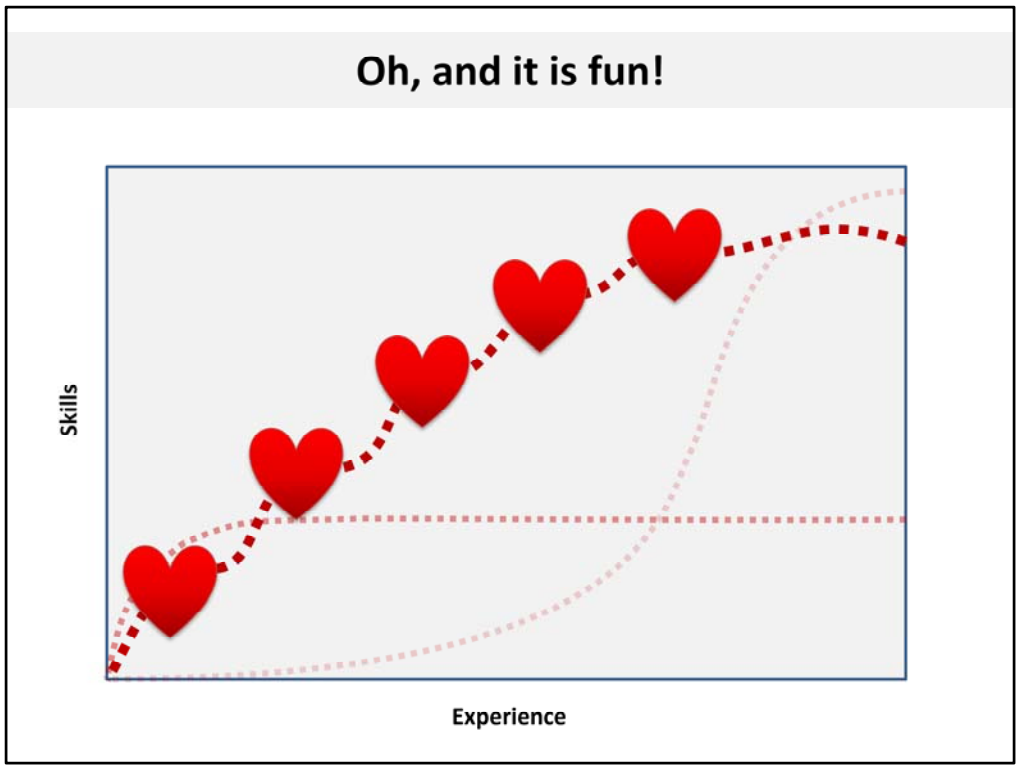


Here are a handful. There are many more you end up learning throughout the game.

You'd think that Goombas would be an endangered species after all Mario's flagrant massacring.



Here's the learning curve for game. You can pick up a well made game and start enjoying it in a few minutes. The better games have weeks of learning.
They steadily crank up your skills at a much faster pace than most applications. There is no dreaded dip where users are struggling to acquire the next level of expertise.



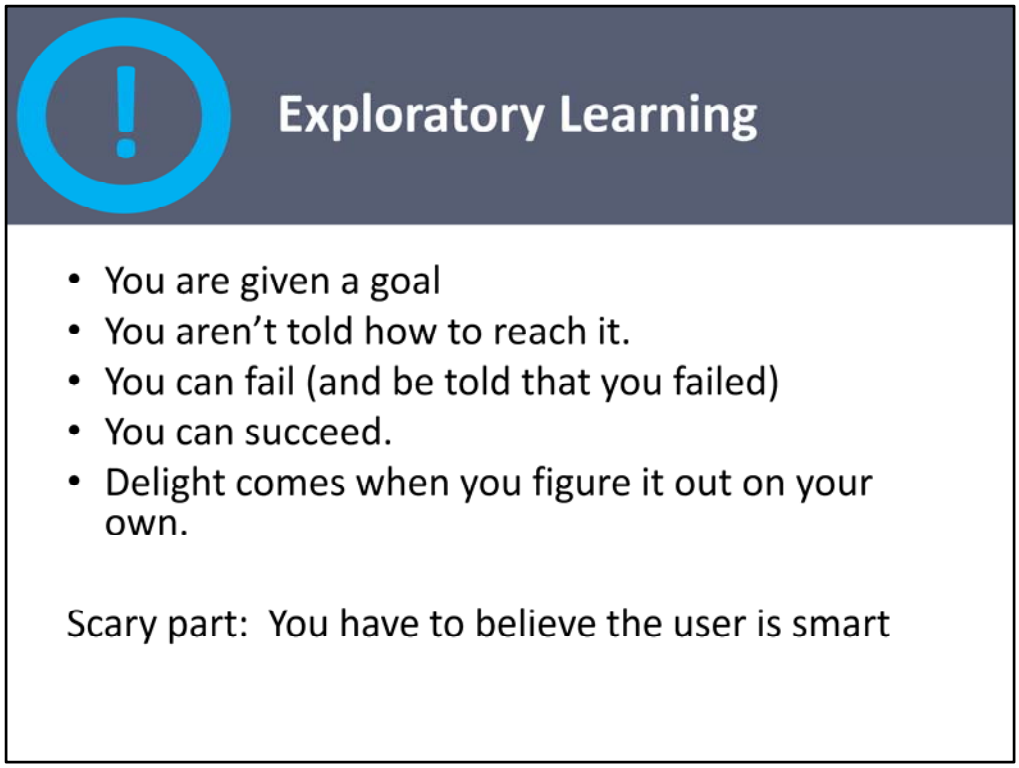
There is one key difference: Fun!

Unlike Word or other apps, users are having so much fun learning to rescue a princess, they will **pay** us for the honor of learning our application. Not only is the game delivering more practical value in a steady fashion, it is doing in such a way that makes the process of learning inherently valuable to the customer. That one-two combo is worth cold hard cash.

Why do games have such a radically different learning curve than advanced applications? It turns out that games are carefully tuned machines that hack into human being's most fundamental learning processes. Games are exercises in applied psychology at a level far more nuanced than your typical application.

Most apps say "Here are some tools"
Games say "Here is how a human being is going to turn a set of tools into useful repeatable skills"

To do this games rely on a simple technique called exploratory learning.

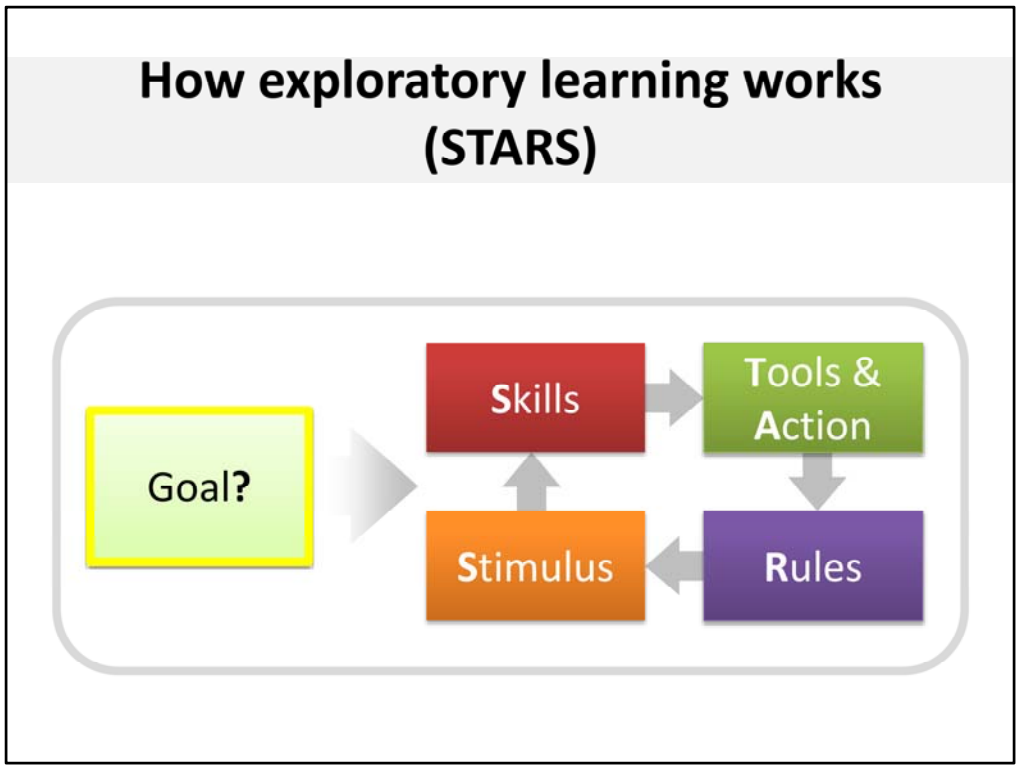


I was talking to one usability expert and he was describing how they measure task completion. Did the user press the buttons in the right order? Their ideal app resulted in new users completing tasks 100% of the time. This isn't exploratory learning. You need to be able to fail and explore the possibility space of a particular tool. Through repeated failure and success, users build up robust skills that can be applied successfully in a wide variety of situations.

Users are not dimwitted monkeys. They are intelligent human beings. Sure, there are limitations, but for the most part we are wired to be problem solvers. Our brains are flooded with lovely chemicals like dopamine and endorphins when we figure something out on our own. That's fun at it's heart: Our brains telling us that we grokked something interesting.

When you build applications that let users be smart, they love you for it. The secret to good game design is simple. Set up situations where there is a problem that must be solved and let the user solve it. Give them subtle clue, but don't take away that 'aha' moment.

(Applications also have exploratory learning, but they lack the mediation of the process that happens in games.)



Here is a basic model for applying exploratory learning to your application. You have a goal, skills in the user's head, tools and actions they take in the application, rules that the application executes and and stimulus that the application feeds back to the user.

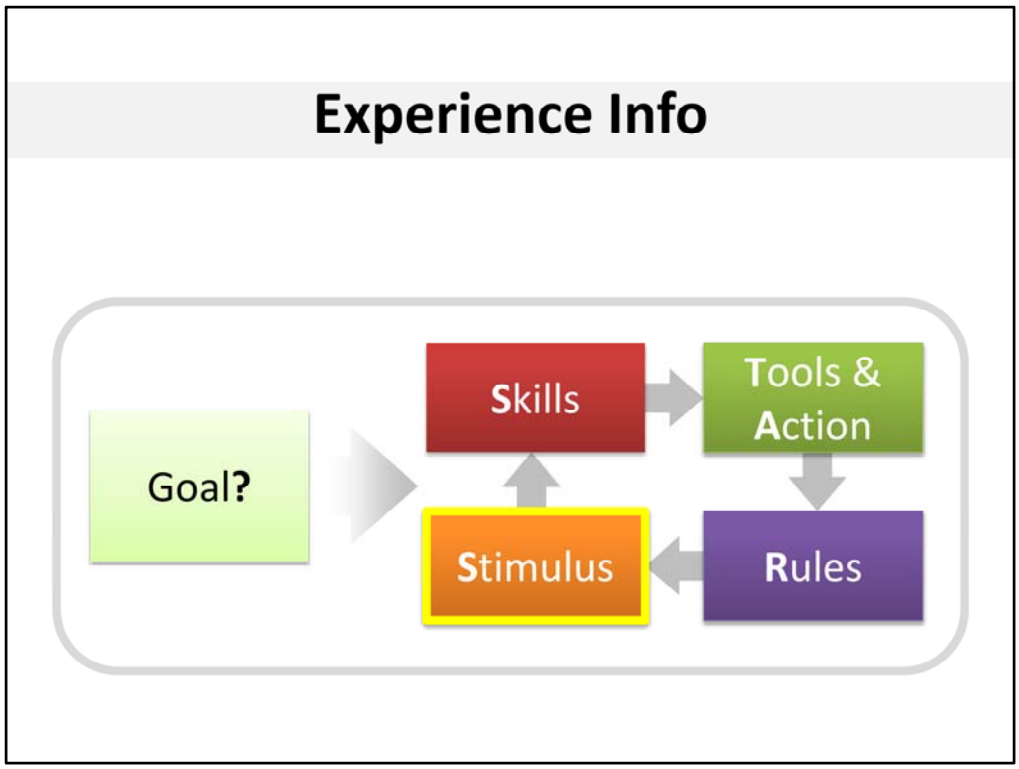
Let's step through these one by one.

How many people here have played a game called Super Metroid?

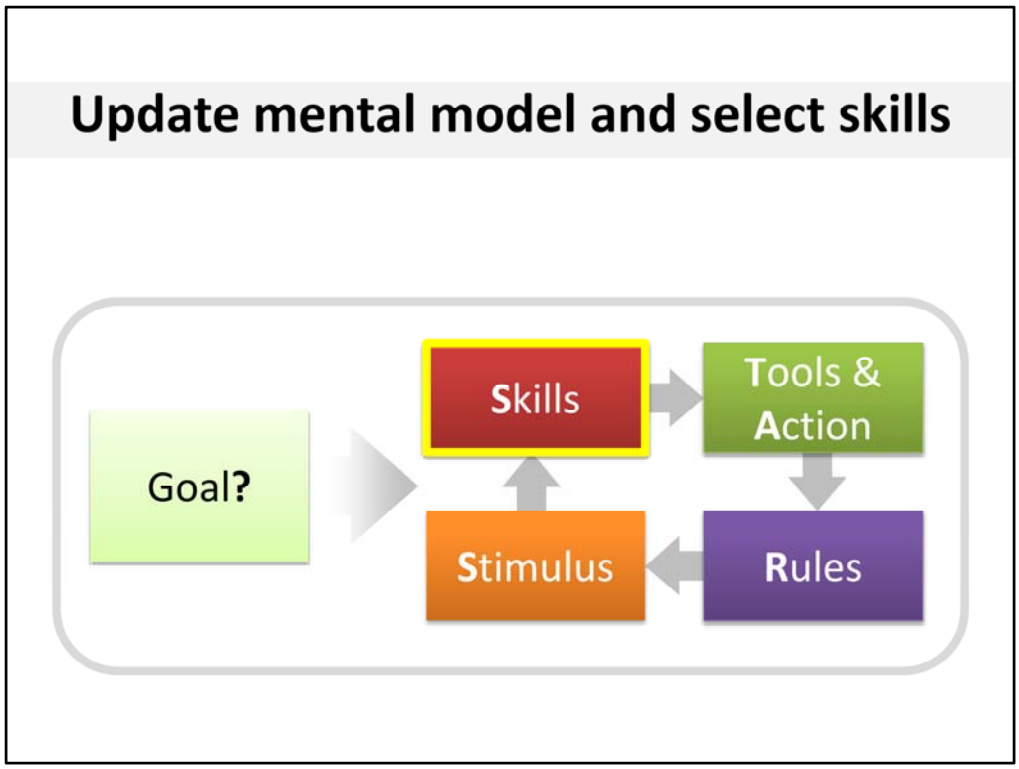
This is a section where you where you learn to perform wall jump. You bounced of the walls, boing, boing, boing until you make it up to the top.

The way that the game teaches you this rather advanced skill is to throw you into a pit. There is no way out except for a deep shaft that seems impossible to ascend. Three little gremlin-like creatures dash by you and bounce up the shaft. Nothing tells you how to get out of the pit. But the gremlins give you a goal and a clue.

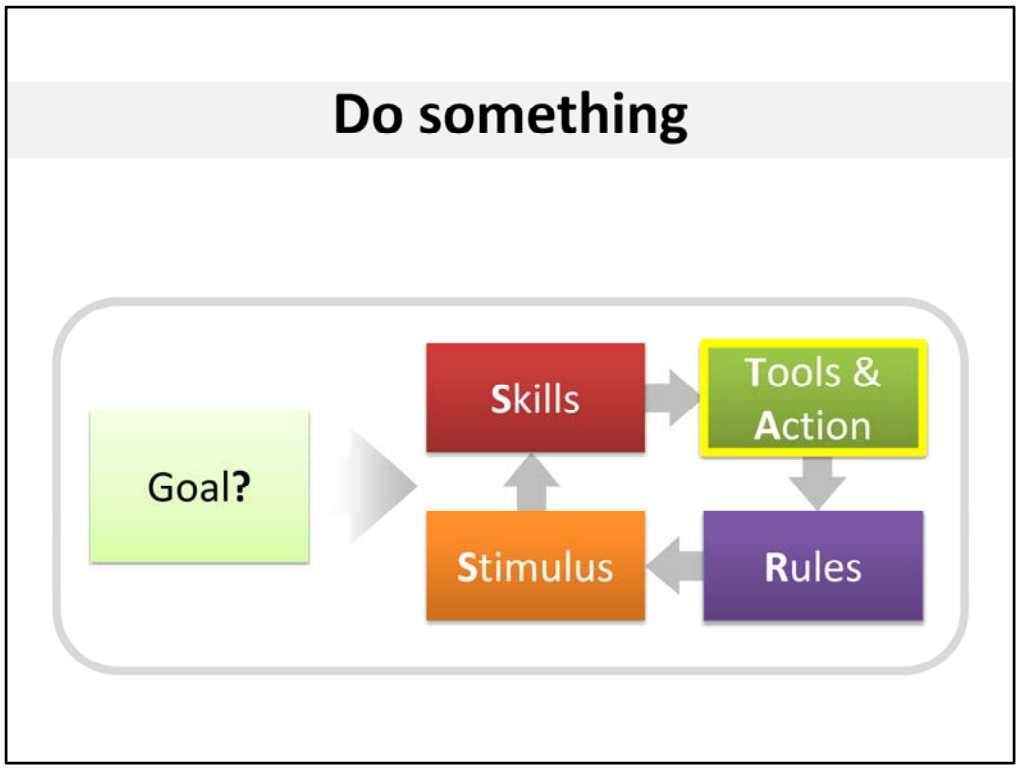
We've put the users in a simplified environments that let them safely explore how to use the tool. In order to succeed, they need to play around with jumping.



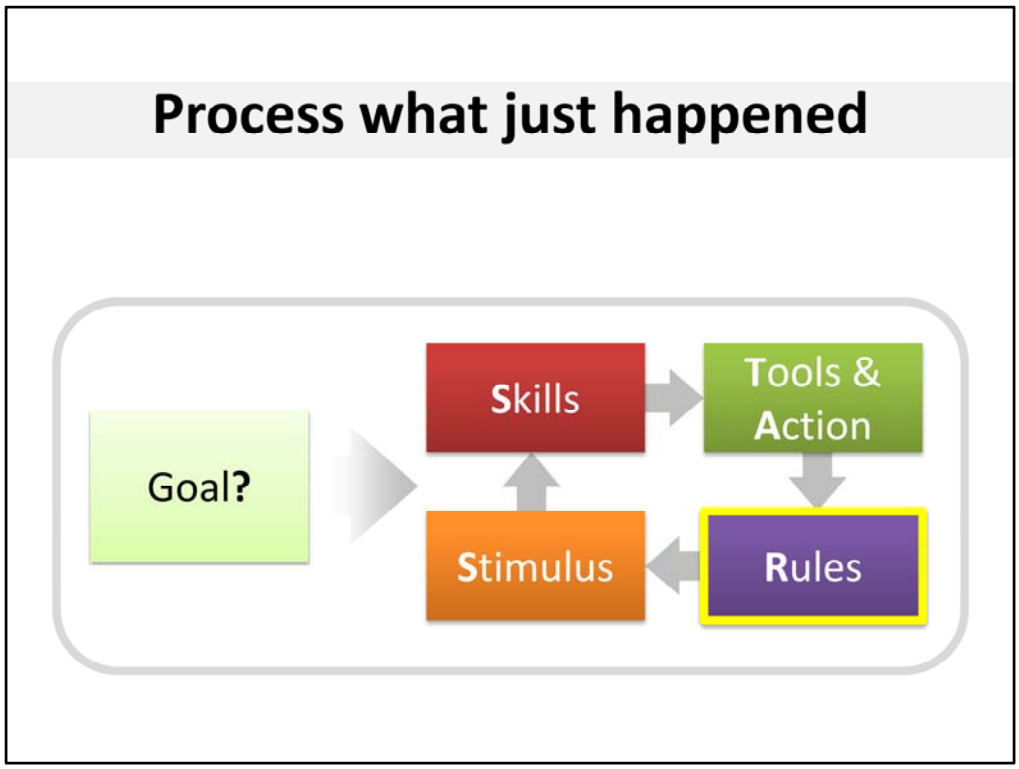
The first thing that happens is you see the gremlins jumping and escaping. The game is telling you something important.



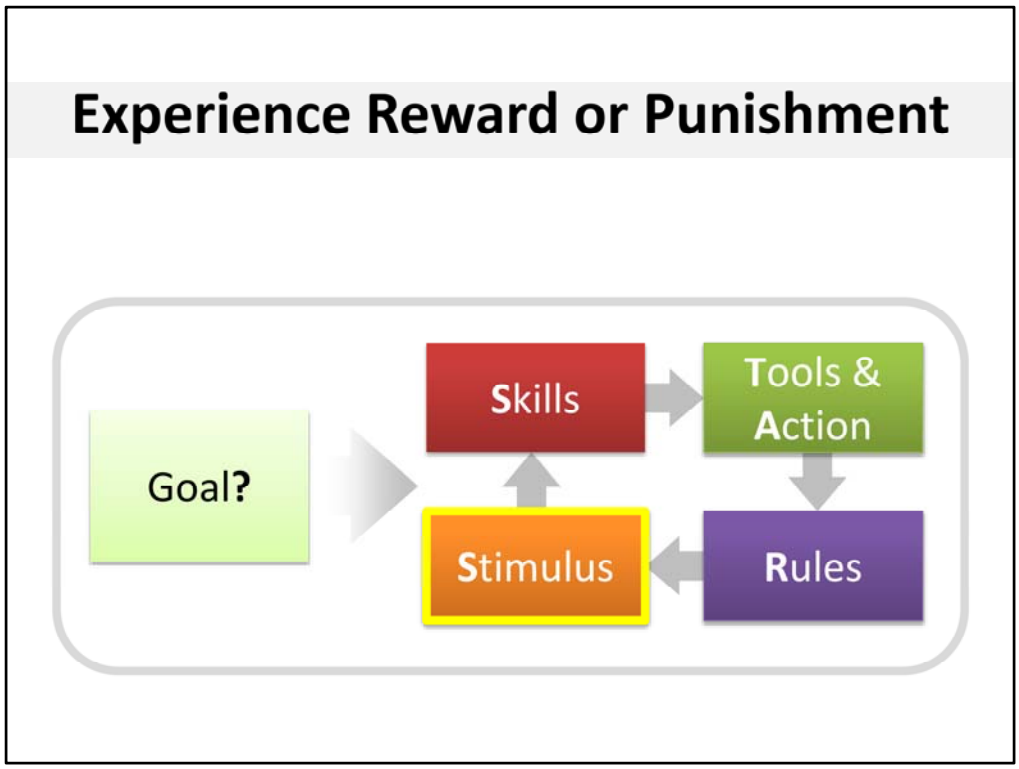
Next, you take a quick inventory of your mental skills. You can run, you can roll…and you can jump. Maybe it is worth trying that out. That fact that there are other jumping creatures in the room subtly primes the player to take the correct action without blatantly telling them “Jump, you idiot!”



Now the user 'does something', often the most likely action they think will move them closer to their goal. In this case, most of them will try to jump.

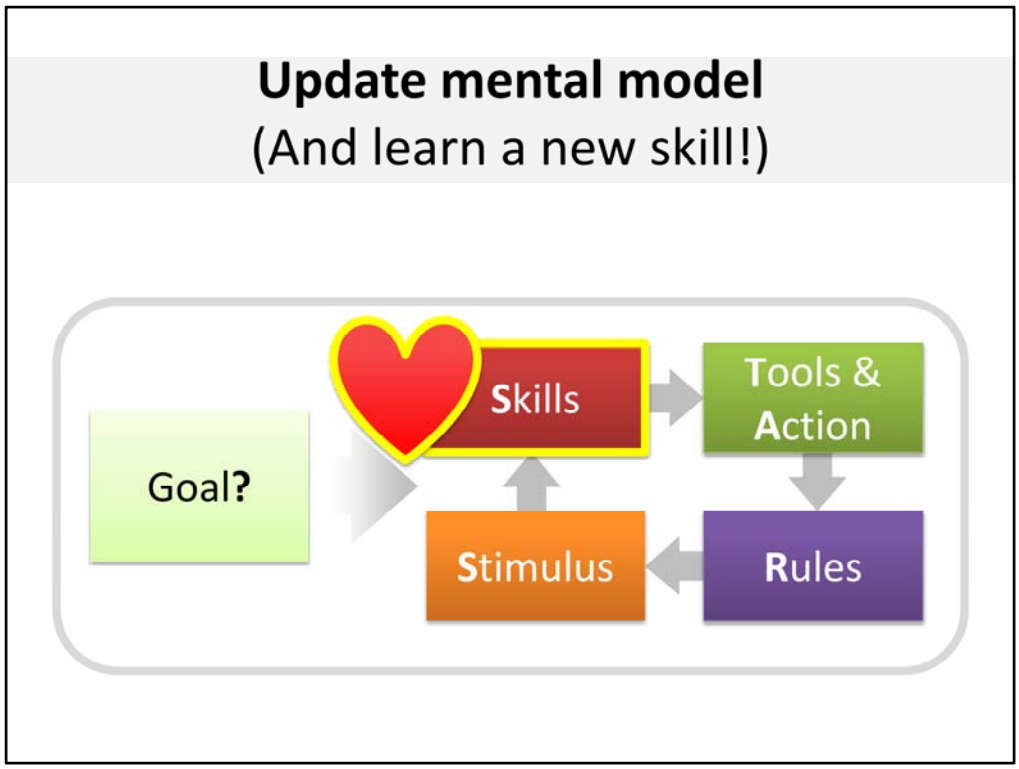


Upon jumping, the complex physics and jumping engine kicks in and calculates exactly what happens. If they are good, they jump a second time as they push off the wall.



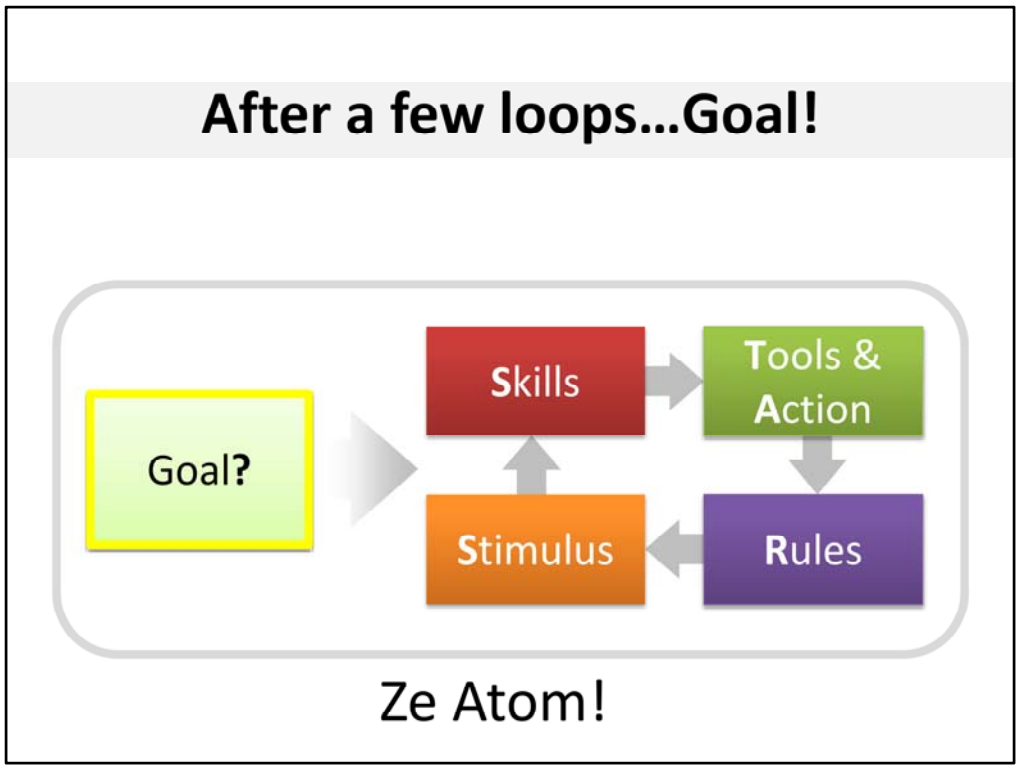
Now two things could have happened.
-They successfully figured out the timing of jumping off walls. In this case, the game shows them making it a bit higher up the shaft than they thought previously possible.
- They failed and fell back down the shaft.

The game's stimulus provides clear feedback on whether or not their action got them closer to the goal.

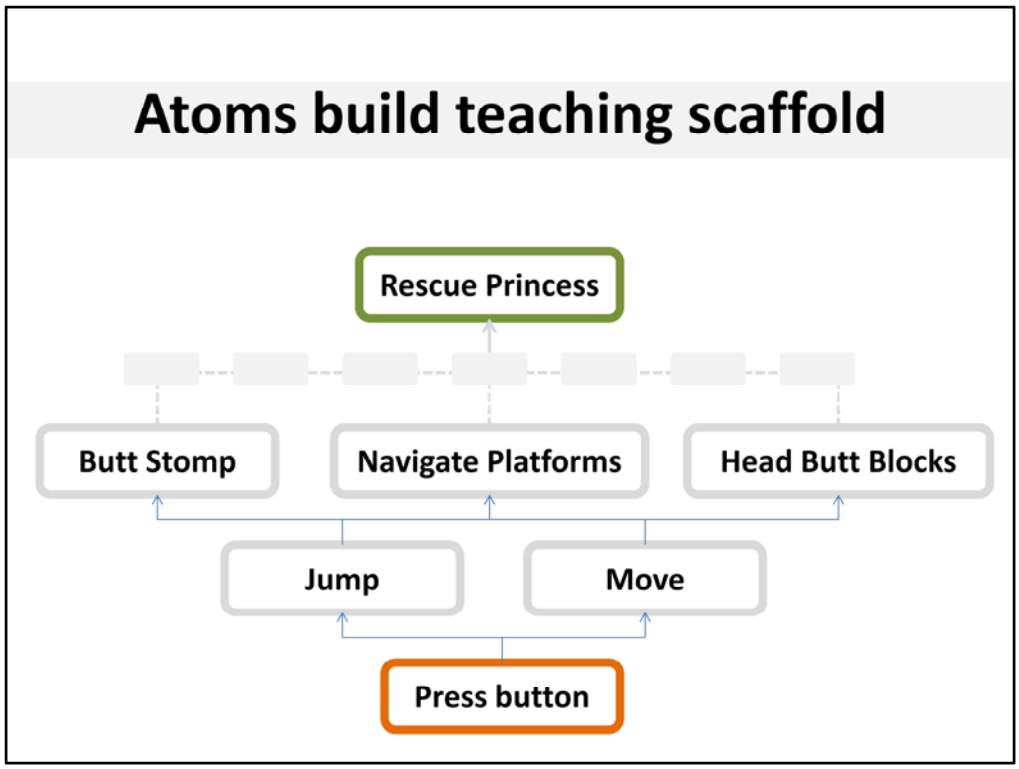


One way or another the stimulus updates the user's mental model. Negative stimulus tells the users "Hey, that didn't work very well…try something different". And their mental model of jumping changes. Positive stimulus tells the user "Hey, that worked! Do it again!"

Several loops of learning, the user's mental model will crystallize and they will have an 'aha' moment where they know exactly what they need to do and how. This moment of understanding and mastery is at the heart of what we call Fun. We've triggered the primitive reward system at the base of all learning. It is the same feeling of delight that you feel when you 'get' a joke. It is the same feeling of satisfaction that arises from a job well done.



The loop we've just gone through is known as a 'Skill atom' or STARS atom. This atomic interaction loop is found as a fractal pattern throughout every single game ever made. You can use it to break apart and analyze pretty much any game. It also works quite well for analyzing where exploratory learning is failing in traditional apps.



You can link these atomic learning loops together to describe the player's experience with an entire game. Each lower level skill acts as the foundation for more complex skills.

Even at the highest levels for the most complex tasks, each atom contains an identical loop the game presents stimuli, the user updates their model and reacts with new action. It is like a fractal with the same pattern of learning appearing at the lowest and highest levels. Our brain keeps learning the same way, regardless of the task at hand. The atoms capture this reality.

This skill chain is is a practical working model of the player's experience as seen through the lens of exploratory learning. By analyzing games according to this structure we can spot all sorts of failure points and opportunities for improvement.

Implicit in this description of interactivity is the fact that users change. More importantly, the feedback loops we, as designers, build into our games, directly change the user's mind. When you are done playing a game of tetris, you are better at playing tetris. When you finish a game of Nintendo's Brain Training, you are better at basic math problems.

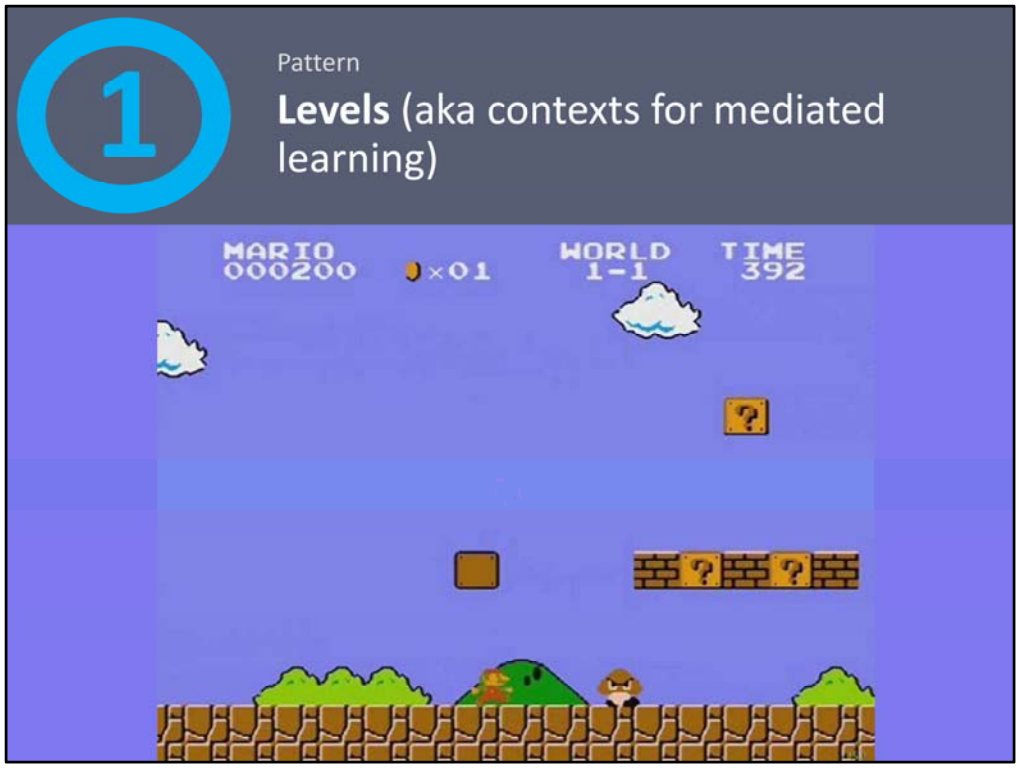
The person that starts using a game is not the same person that finishes the game. Games and the scaffold of skills atoms describes in minute detail how and what change occurs.

This is a pretty big philosophical shift from how application design is usually approached. We tend to imagine that users are static creatures who live an independent and unchanging existence outside of our applications. We merely need to give them a static set of pragmatic tools and all will be good.

Game state that our job is to teach, educate and change our users. We lead them on an explicitly designed journey that leaves them will functioning skills that they could not have imagined before they started using our application. Our games start off simple and slowly add complexity. Our apps must adapt along the user's journey to reflect their changing mental models and advanced skills. Failure to do so results in a mismatch that results in frustration, boredom and burnout.

Picture: http://flickr.com/photos/blueju38/2149767604/

Now that we have a basic model of how games work, we can look at common design patterns found in games and apply them in an intelligent fashion to other applications.



This is the very first screen of Super Mario Bros. It demonstrates the use of a simple constrained environment that lets users build skills around a new tool.

The player is handed a new tool called Mario the first time they see this screen. They don't know how to use him. The screen gives them a playground where they can try different things.
- Blocks that reward jumping by giving out coins.
- Goomba that rewards successfully learning how to attack. It also teach the players to avoid Goombas on pain of death.
- Blocks that teach the player how to collect powerups.

There are a couple interesting points to note:
-The awarding of a new tool is almost always paired with a simple level that lets the player learn the tool in a somewhat safe environment.
- The player cannot pass this section without mastering at least one critical skill, in this case moving and jumping. This sort of gating ensures that the designer can rely upon the user having the jumping skill available at later points in the game.

This is different than most apps. In many apps, you sort through the options and turn on a new feature. There is nothing that is the equivalent of a 'level' or learning context to help you build skills associated with the tool.

Here is an example how learning is treated in many applications (as well as too many games). This is Yahoo! Mail's attempt at creating a simple context for you to acquire new skills.

Let's look at this from the exploratory learning perspective. What is this series of interactions actually teaching?
• **Goal?**: My goal is to use the mail program.
• **Initial stimulus**: In a quick scan, I see a bunch of random text and a button.
• **Skill triggered**: I pick the lowest cost action that will get me closer to my goal. Pressing buttons! I know how to do that.
• **Tools and Action**: I click the next button. I could have read the text, a harder skill. But that wasn't part of my goal and it was harder.
• **Rules and Stimulus**: The tutorial advances and I'm presented with another next button. The text updates to 2 out of 4.
• **Skills**: Ding, Ding, Ding! Progress! Ooh, I'm closer to my goals of using the mail program.
• **Tools and Action**: I immediately press the next button again…I've gotten a hand of this now. At this point, it is basically a game of whack a mole. Next, Next, Next…voila! I can use the application! Success.

Grind: Did I learn a lot? Not really. Most people know how to click the next button so there wasn't much a burst of mastery. In game terms, forcing people to perform actions where they already know the result is known as a 'grind'. You see it many RPG-style games. There is a lovely article in the October issue of game developer magazine where they actually measure the brain activity of people who are grinding through a tutorial. Interest level drops off substantially. Because it doesn't tap into exploratory learning successfully, it is a great way to bore or frustrate your audience.

Lessons

- **Purpose**: Always ask "What am I actually teaching?"
- **No rote learning**: "Do A to advance" = rote learning = boring.
- **Part of the experience**: Make the tutorial seem like a natural part using the application.
- **Simpler scenarios**. Fewer parts, fewer possibilities. The user will be struggling with the new concept and new interface.
- **Gating**: Prevent the user from advancing until they have mastered the basic skill associated with the tool.
- **Look for the smiles**: Test "Is the user delighted?"

If the user isn't smiling, your exploratory learning loop is improperly balanced. People aren't learning at the proper pace or they are skipping the learning that you've built into your STARS atom.

This is the inventory screen from Zelda: A Link to the past.
Remember that a game is a journey that changes the player. In turn, the application must change as the user evolves.

Apps will often have an options dialog that lets you turn on and off features, but for the most part what you see the first time you start the application is what you are using two years later.

Games use the design pattern of 'item', which you can think of as simple modular tools that are introduced piecemeal to the player. In Zelda you start off with nothing. Then you get a sword that allows you to attack enemies. Eventually you get a boomerang that lets you attack enemies from a distance. Each time you get an item, you are trained on how to use it with the level pattern we discussed in the previous slides. Piece by piece, your toolkit evolves until by the end of the game, you have a wide array of tools and the skills to use them.

This only works because each tool is:
- Modular: Tool can be added or taken away without affecting the usability of the rest of the app.
- Paired with learning levels

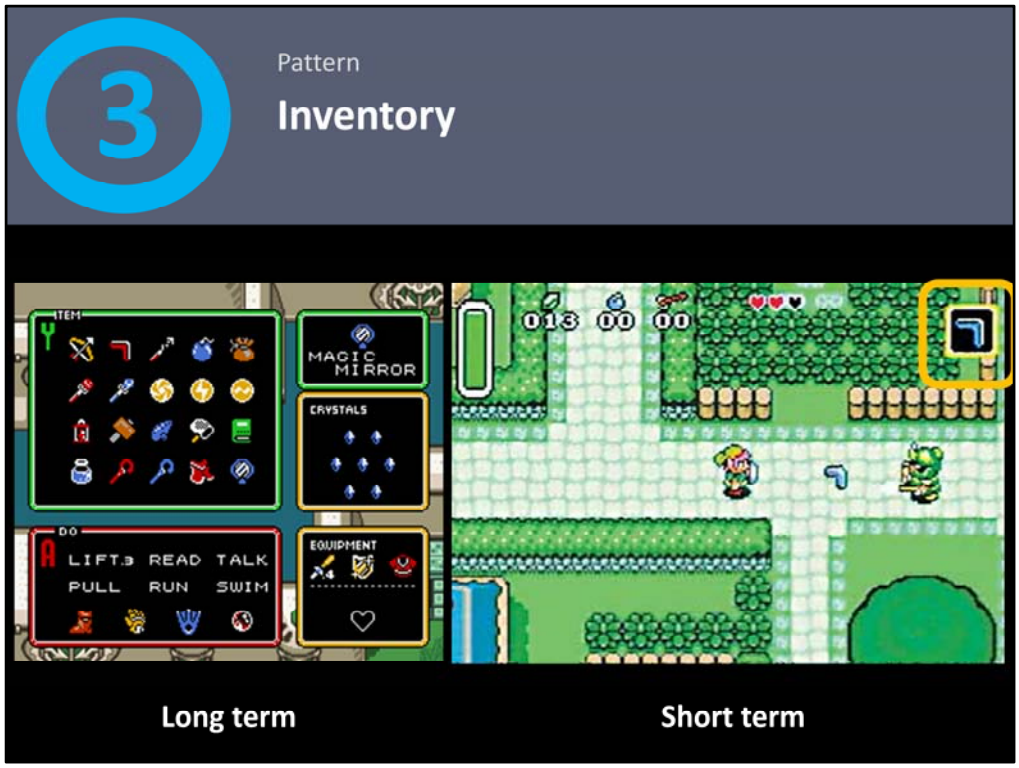
The Item pattern has also made it's way into software in the form of widgets and gadgets. They are modular, but they usually aren't paired with level-like contexts that help users explore them in a safe fashion. Still, this is a huge step up from the massive sets of fixed functionality that you find in applications like Office or GoogleApps.

Lessons

- **Limit fixed tools to simple core activities**. “Blog”.
- **Modular**: Make adding and replacing activities like Legos.
- **Create items according to skills**. Let the user learn and master one tool before you give them the next.
- **Evolve the toolset over time**: Introduce advanced skills through item acquisition.

You want the core fixed toolset of your application to be as simple as possible. Everything else should be a modular addition. A fun exercise is to imagine a product like Word. Now strip out everything that isn’t in the core 10% that you use 90% of the time. Now make everything else a modular widget that can be added or removed as the player learns.

Such exercises really make you think about how you structure complex applications in exciting new ways.



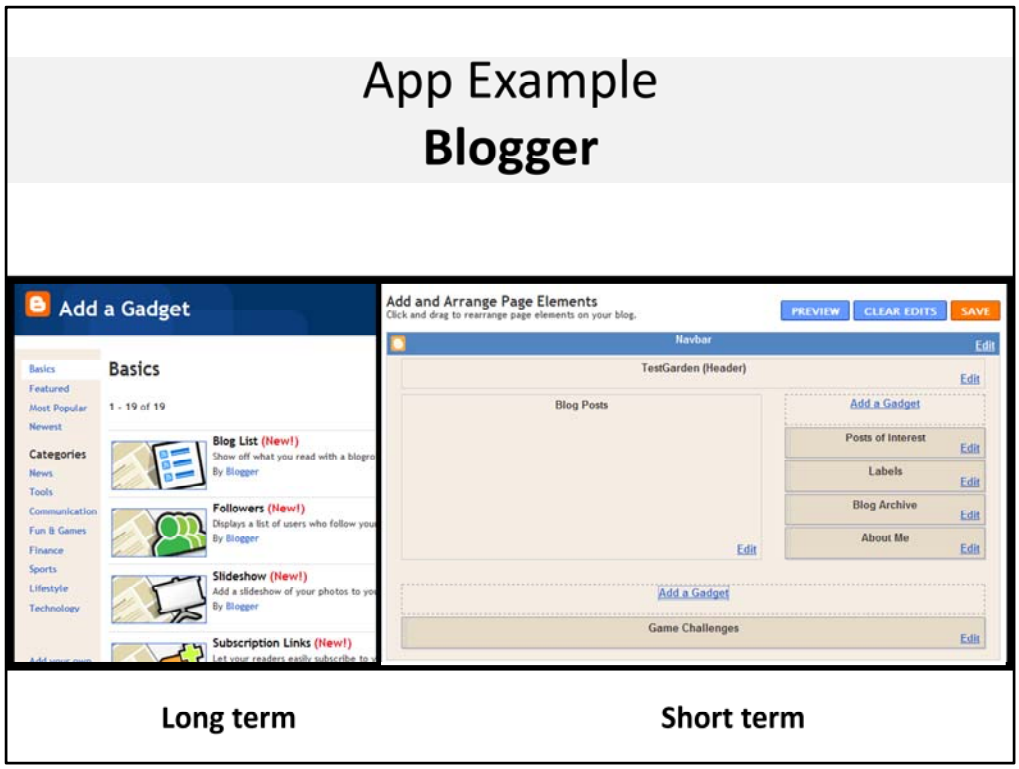
Here's Zelda again. When you start turning your tools into modular swappable elements, you need an easy way to manage them. So you create a new basic tool that you teach players to use immediately. It is called an inventory system. This is one of the earliest tools you introduce the player to and you make sure that they know how to use it. By doing so, you unlock all the other modular tools out there.

The basic inventory pattern is simple:

• **Long term storage**: Any items or tools that you collect over the course of learning the game.

• **Short term active usage**: The current items or tools that you are using right now. There are often a fixed number of slots based off 'best practices'

• **Item acquisition**: A system for slowly introducing tools into your long term storage.



Some modern apps have inventory systems as well. They tend to have a few problems. Here's blogger's widget system. The palette system in Adobe products is also an inventory system.

• **Optional**: They are treated as optional. Many users don't know how to use them and the inventory systems are often quite baroque.

• **Set up for expandability, not best practices**: Product customization is typically seen as the domain of experts, not everyday users. As such the systems focus on power user's need for massive flexibility. They don't focus on the 80-90% of scenarios that a moderately skilled user will encounter. Contrast the power of the Adobe palette system with the ability to switch out a single weapon slot in Zelda. In the latter, quick inventory switches are easy and common so that you have the minimal set of tools that you need at any one moment. Most apps err on the side of having too many features up on the screen at once since the cost of switching is high.

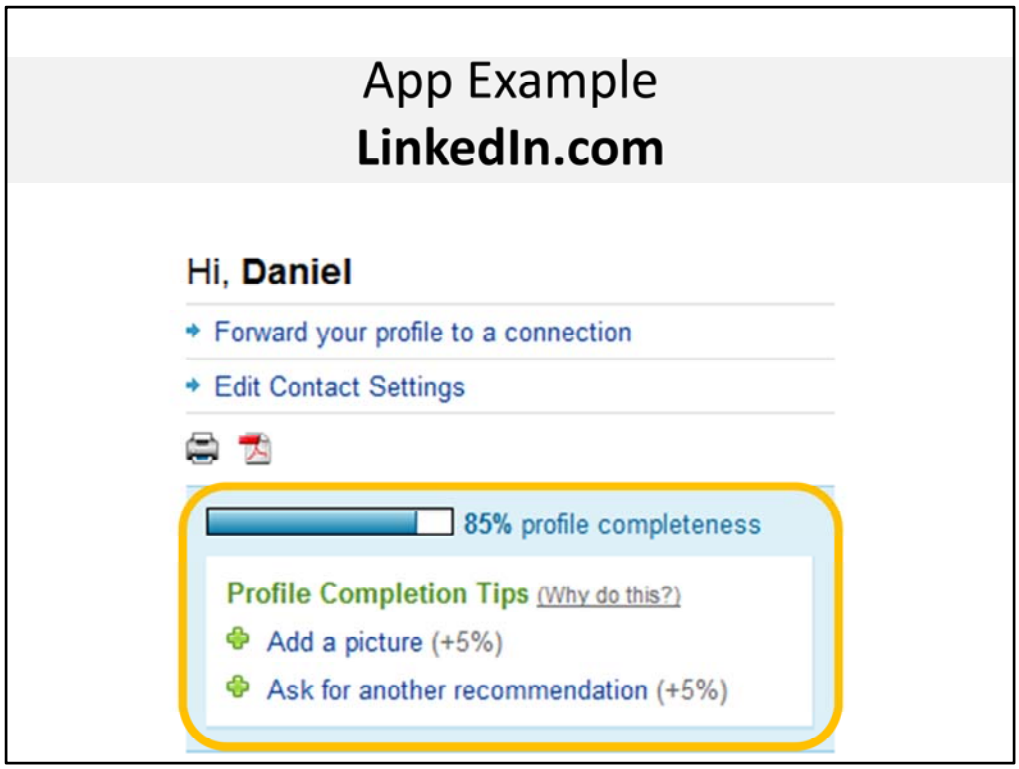
Lessons

- Active list is short and contains only the tools necessary for the tasks at hand.
- Active list is pre-organized: Armor, helmet, runes > Blog, Secondary links.
- Storage list is easy access but usually modal.
- Ability to discard excess items

This is World of Warcraft and one of the many quests that you encounter on your adventure through the land.

Games are well known for making the user aware of explicit goals. Super Mario Bros tells you up front that you need to rescue the princess. By letting the player know the 'what' they are attempting to achieve, you dangle a carrot in front of them. This is very important for exploratory learning to take place.
- By explicitly offering the goal, you alert the player that something is possible. Often, without this prompt, they won't even realize the outcome was feasible.
- The goal gives them something to measure their incremental progress towards. Even if they fail, they are able to say "I made it closer to my goal". This results in updating their mental models and continuing in a positive direction.

When explicit goals do not exist, users often stop doing activities that are actually quite positive. There is nothing to get them over the little bumps of frustration that are a natural part of exploratory learning.



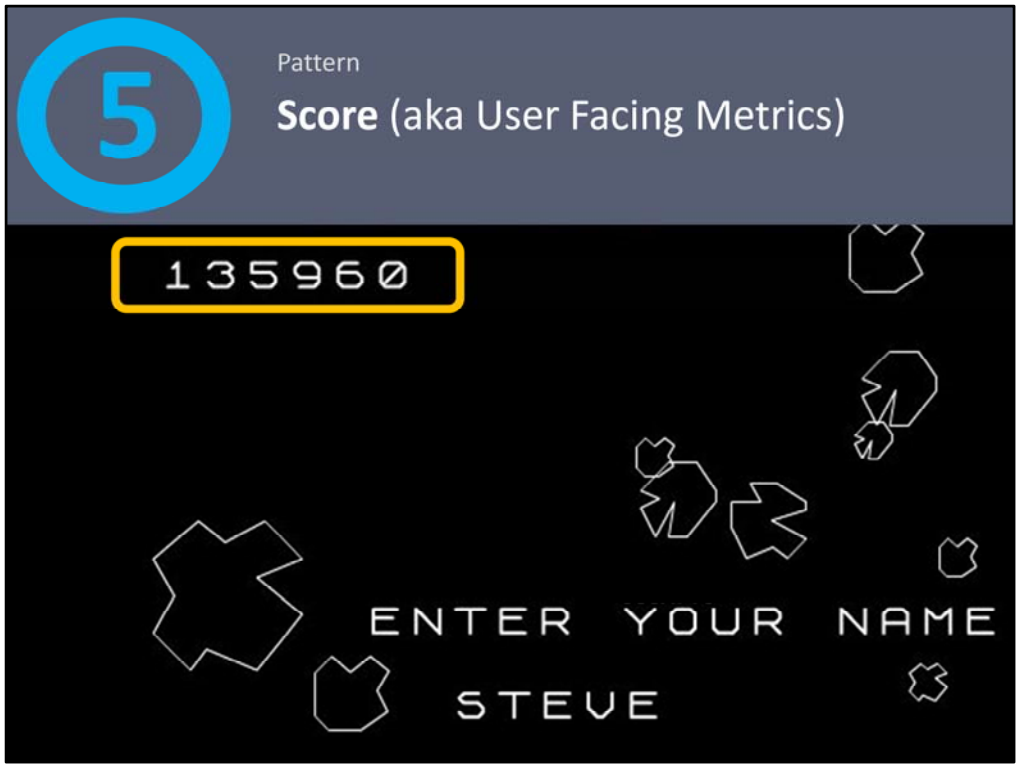
Here's an example of quests in LinkedIn.com, a social networking site. There are actually a couple quests here:
-Profile completeness. Note the explanation 'why do this?' . It doesn't tell you how.
-Add a picture, Ask for another recommendation.

Nesting of goals so that a short term goal help advance a longer term goals is a great technique for leading players with short attention spans towards a more complex outcome.

I'm always impressed by how obsessive folks become about completing their profiles. We are bred to complete activities. If we don't, our brains end up churning away on them, expending excess energy in an attempt to understand what this obvious problem means. Finishing a cognitive loop so that it can be packaged away as a nice neat mental model again gives a distinct feeling of relief and pleasure.

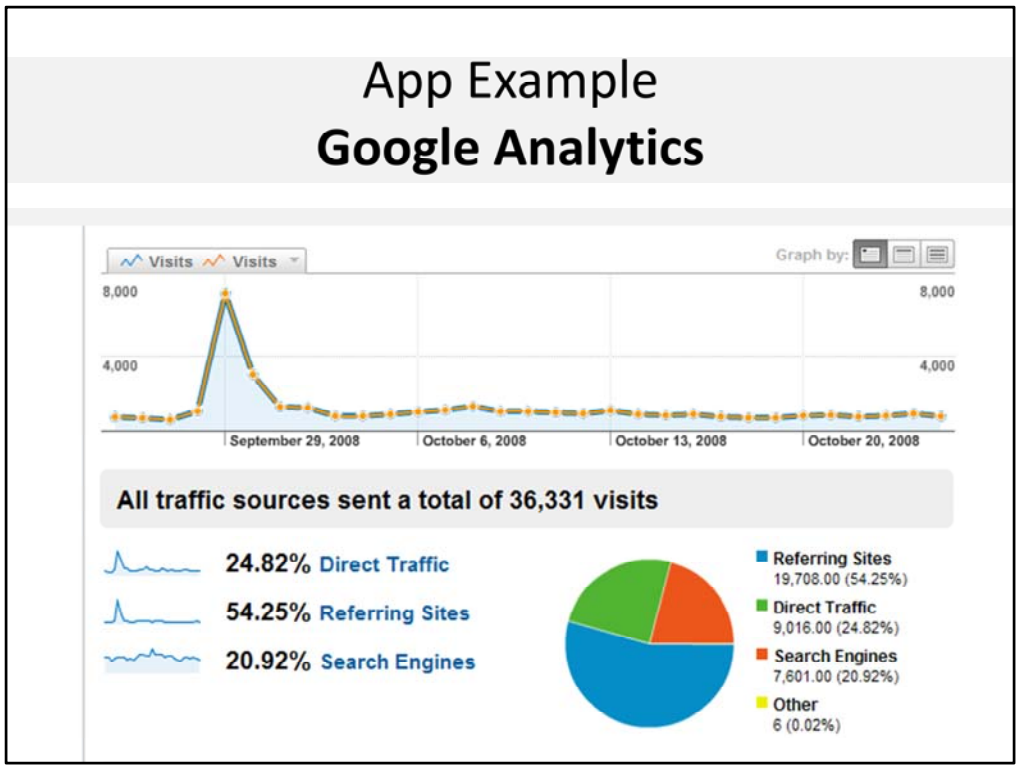
Lessons

- Use quests.
- Be explicit about the reason: “How does this activity help me get toward a bigger goal?”
- Makes the goals easy to understand.
- Lots of variations are possible



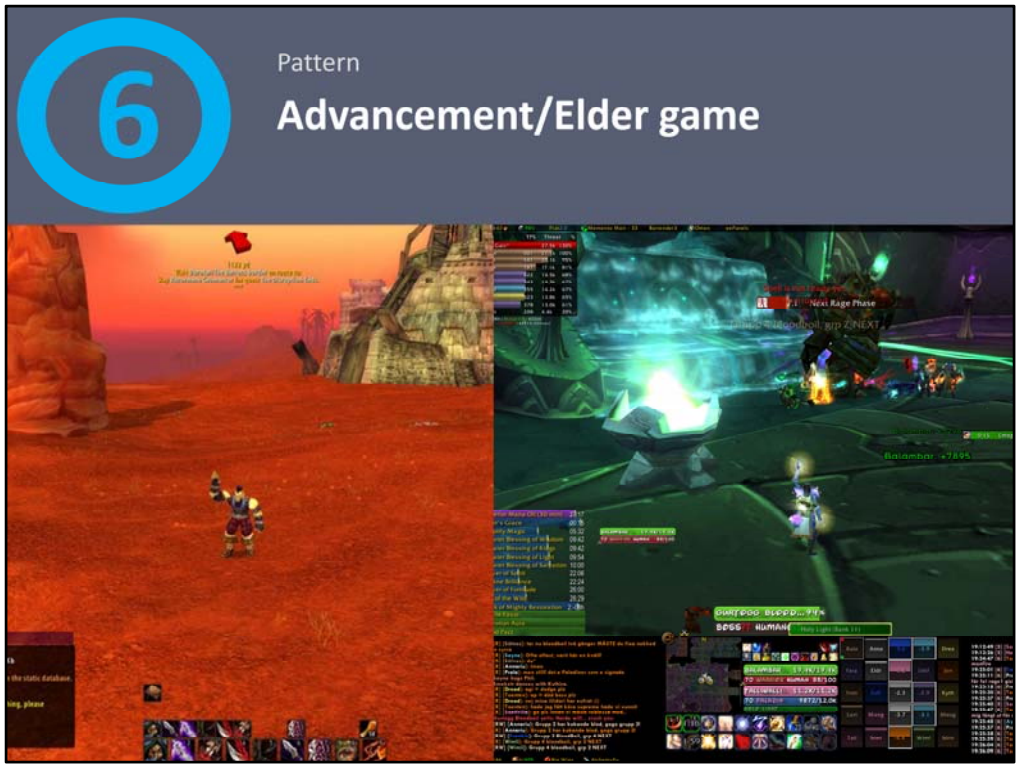
Steve just got one heck of a good score. Go Steve. This is a game called Asteroids and it has a very prominent score.

Points have many uses, but one of the most important in exploratory learning is that they show you progress towards your goal. With a simple number you can tell if you are doing well or if you are doing poorly.



Modern websites use metrics all the time. Here are some states for my site. I get a steady 700 to a 1000 people a day. However, most of the metrics are hidden from the users. We use them to guide the complex systems hidden behind the drapery. Games of course do this as well, but with their scoring systems, they also create friendly user-facing metrics. Applications would stand to do this more often.

Tie metrics to progress through the skill atoms and you'll see learning rates soar.



The most common structure for games. This is World of Warcraft. On the left is an early player, focused on leveling up. On the right is an advanced player out raiding with his crew. Notice that the advanced player's interface looks like a jet fighter cockpit. The app you end up using is not the app you start with. What is interesting about this is the progression that occurs over time.

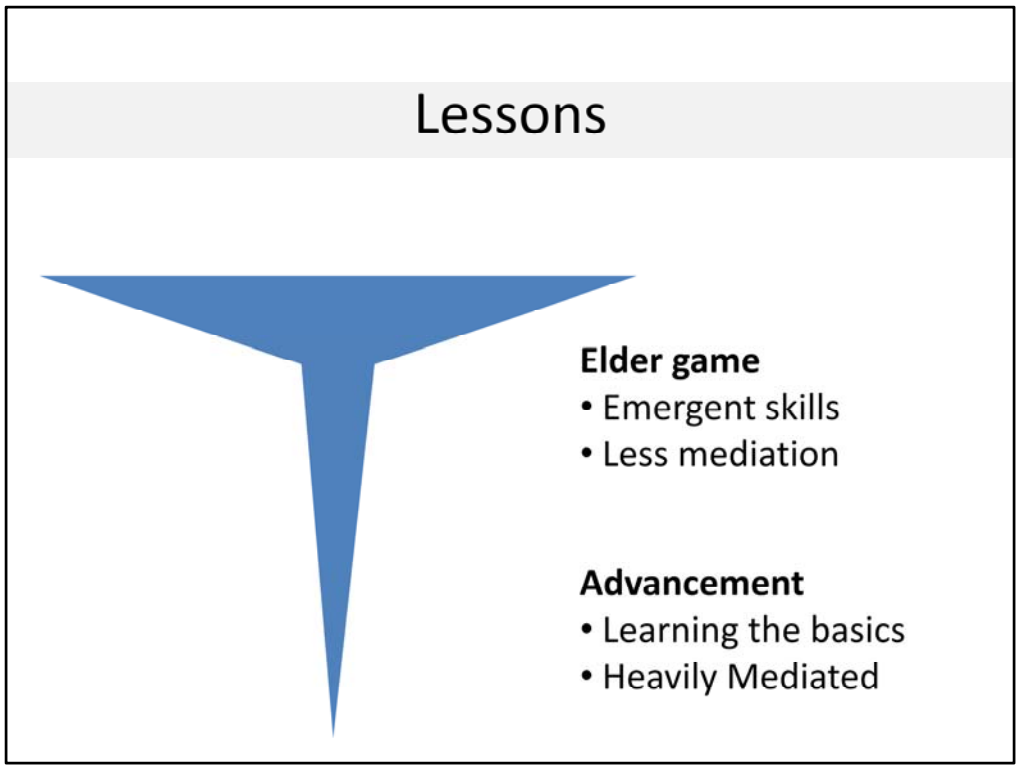
There are three types of skills found in games.

• **Pre-existing skills**: These are skills that are usually made use of in the first couple minutes of using a piece of software. Identifying and leveraging these skills is what makes an interface seem 'intuitive'.

• **Rewarded skills**: The next set of skills are those that the player doesn't yet know, but are taught through the STARS atoms in the game. The designer adds explicit reward and punishment metrics that guide the player towards understanding these skills. There are a finite number of explicitly rewarded new skills a game supports.

• **Emergent skills**: The last set of skills emerge from clever combinations of existing skills that the designer either did not reward or could not reward due to their complexity. The various moves and insight that an advanced GO player has at their disposal are good examples of emergent skills. There are potentially an infinite number of emergent skills.

Games tend to split themselves into two distinct stages as a way of teaching these different skills.

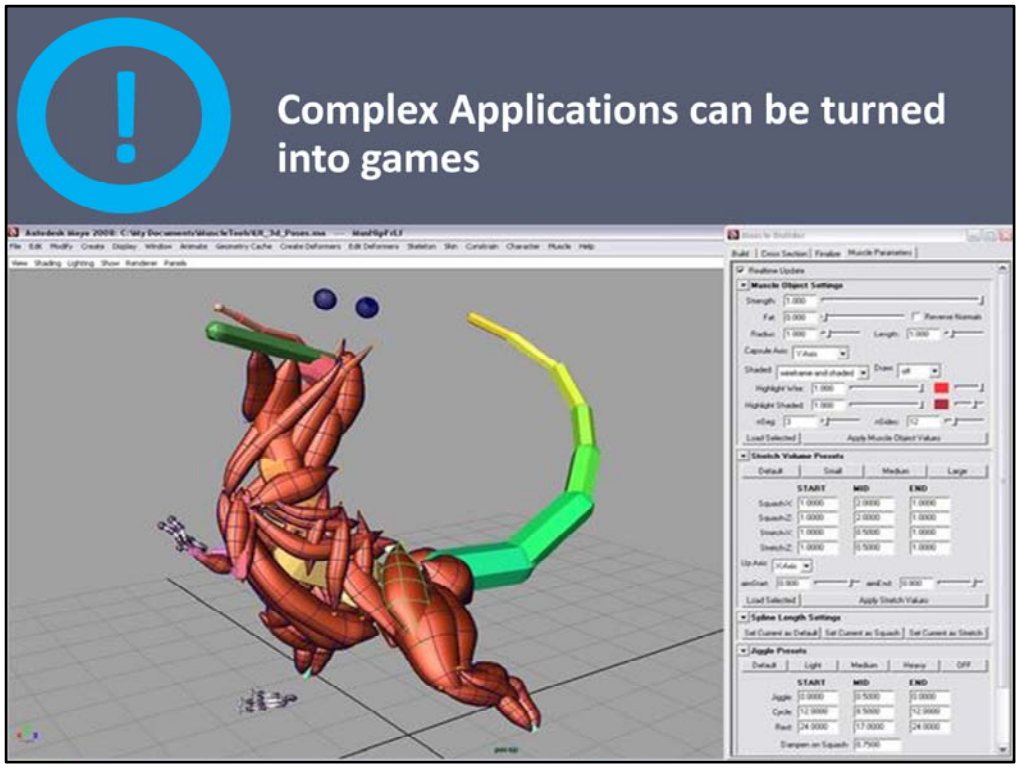


During the initial portion of the game, called the Advancement game, the player is presented with a sequence of tightly controlled levels that teach the player rewarded skills. The goals are often quite short term and the success rate of players is generally high.

After a certain point, the game switches to what is known as the Elder game, where players are expected to learn on their own. The goals become less achievable on a consistent basis and act as prods that encourage players to discover and innovate in the creation of their own emergent skills. In World of Warcraft, the game shifts the focus to guild-based boss battles and Player vs Player combat. The former requires players to master working together as a high performance team, an area that has almost infinite depth. The latter requires players to be more skills than other players, creating an emergent skills arms race that is intrinsically unwinnable.

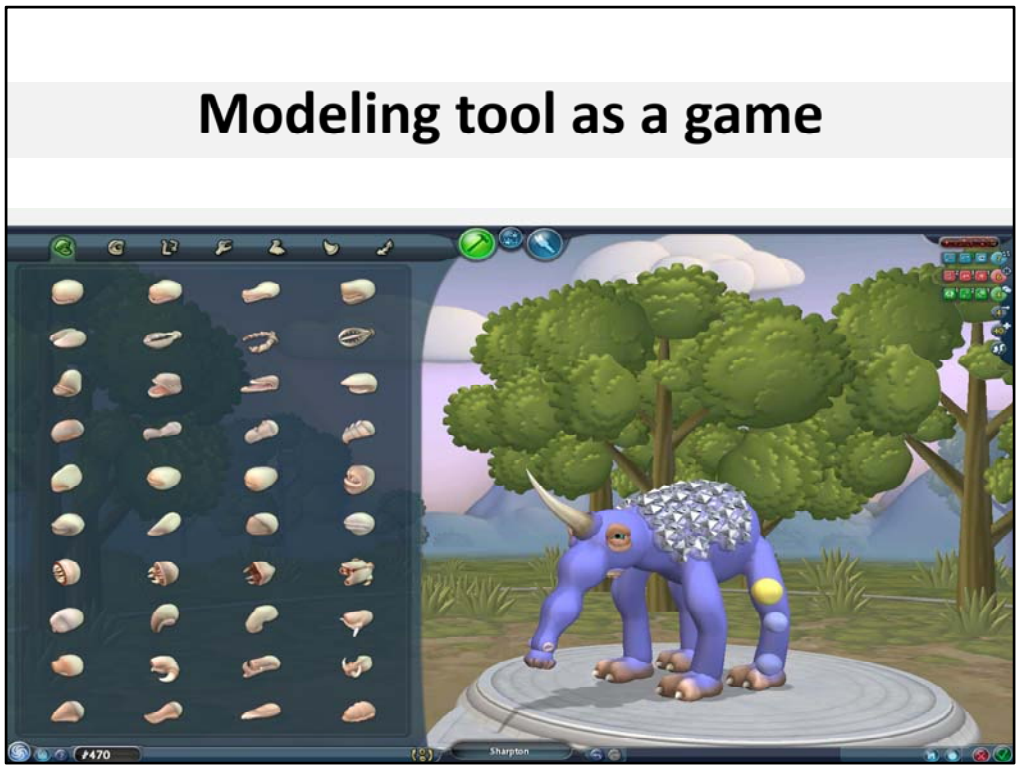
It is quite possible to structure applications along similar lines in which users are given quests to learn basic skills in a relatively controlled manner and then provide broader goals later on.

Adding delight through exploratory learning is the next big revolution in interaction design.



This is a 3D application called Maya.
The folks that make it are very happy to sell a couple hundred thousand copies a year.

It takes years to learn. In fact people go to school and pay thousands of dollars just to gain some small level of competence. This is the sort of elite applications that will never in a million years be replaced by a game.



This is Spore. It is on track to sell 3 million copies. That is about 20 to 30 times more users than will use Maya in a year. In a very short period of time, there will be more models created with Spore than were created in the entire decade long lifespan of Maya. Scarily enough, models created in Spore will be more sophisticated than those created by 95% of Maya users.

It is an unfair comparison. Maya is still considerably deeper than Spore and has a much broader range of applications. And yet, the seed is there. Where will products like Spore evolve into over the next decade? It is based on an inherently more approachable and powerful method of putting advanced skills in the hands of users.

The demand is proven. After iteration upon iteration, the power of the toolset will only get stronger. What happens when you have millions of users capable of making almost any 3D model they can imagine and they are trained in game instead of a school?

Here is where we are heading. This is a chart the shows the evolution of the bathroom scale.

In the upper left corner is a functional scale. It weighs things. In the software world, this is comparable to the typical desktop application.
In the upper right is a usable scale. It makes it easier to weigh things. Notice the indicators on where to put your feet. In the software world, this is comparable to the AJAX web 2.0 applications. They are easy and fast to use for the task at hand.
In the lower left is the other branch, aesthetic design. It makes for an emotionally appealing and attractive scale. In the software world, this is philosophy is comparable to Flash websites.

Finally, in the lower right we have WiiFit. This is a game release by Nintendo this year that helps you do Yoga, track your weight, improve your balance, exercise and do a dozen other interesting and useful activities. It is on track to sell 12 million copies this year. That's a lot for a bathroom scale.

WiiFit is the world's most successful bathroom scale in the history of mankind because it is a game. As a game it combines three incredibly powerful value propositions
• **Delight**: It is fun to use.
• **Ease of use**: You can start using it immediately with very little confusion or frustration.
• **Depth**: Using the same basic hardware as a typical scale, it enables dozens of more activities than your typical scale.

The result is a product that is flying off the shelfs. If game design can do this for bathroom scales, imagine what it can do for other classes of applications.

What we covered

We can build high value applications that are both easy to use and deep. And fun.

- Learning skills is a major and often ignored aspect of interaction design.
- Use the STARS Model to identify what users are actually learning.
- Game design patterns that help unlock exploratory learning in your applications.

We didn't create an application that rescues princesses. However, I hope we did manage to cover some of the steps necessary to bring the joy found in princess rescuing games to the world of application design.

If you'd like to learn more, here are some wonderful links. I particularly recommend the ShuffleBrain presentation if you are looking for practical examples of game design patterns and parallel examples in modern web apps.