Engineering Blog: How a Calculator Enables Gameplay in Demon Crush

Personal Blog Sample
by
Robert Gervais

When my teammates asked me to implement a math system, they asked me if it'd be possible for a designer to write out an expression like $2 * (3 + 4)$ or $-2 + 3 * (4 - 5)$ or $(2 + 3) - (3 - 5)$ – really, all they wanted was any arbitrary human-readable mathematical expressions to yield a final output. And when they described what they were looking for, I immediately thought about the HP 35S calculator.

Reverse Polish Notation

A photo of the HP 35S circa 1972

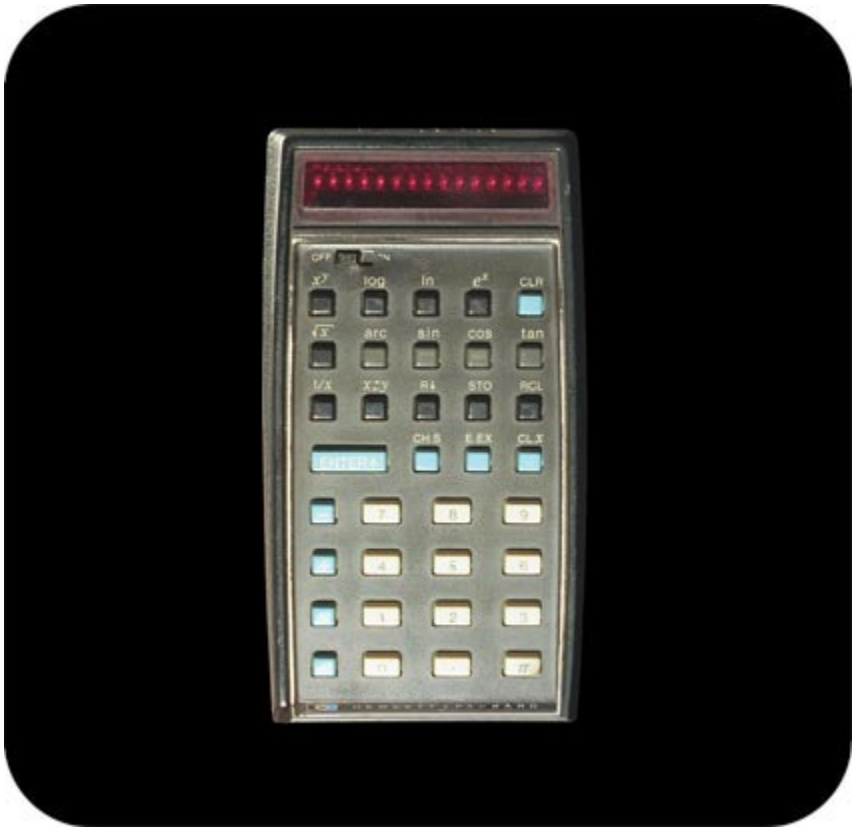


Image credit: [HP Virtual Museum]

Aside from the fact that Hewlett Packard named its scientific calculator the *35S* because it had 35 keys when it debuted, the really interesting note about the calculator is that it was built to compute any arbitrary human readable expression using an operational stack and Reverse Polish Notation.

First, let's talk about what Reverse Polish Notation is, and why it was so useful in solving this problem for our designers. Reverse Polish Notation, or RPN, is the popular name for operator postfix notation in mathematics. Typically, we write mathematical expressions in a way that is easy for humans to read and understand. Take *2*(3+4)* as an example – if we were to write that expression on a whiteboard and ask someone to solve it, it would be a simple task, because a person would know to first solve the parenthetical expression *(3+4)* before multiplying the result by *2* to arrive at the the final value of *14*.

We use the phrase, "human readable" because a computer doesn't know how to semantically evaluate and expression like *a*(b+c)* without some help. And this is where the invention of RPN comes to save the day.

A photo of Jan Łukasiewicz

Image credit: School of Mathematics and Statistics, University of St. Andrews, Scotland

In the 1950's, one of the most important mathematical logicians in modern history, Jan Łukasiewicz, published a book on formal logic, which includes demonstrations on how any arbitrary expression can be represented without its parentheses by placing operators before or after their operands. For example, $a * (b + c)$ can be represented as $* \ a + bc$, which is better known as operator prefix notation because the operator precedes its operands or it can be represented as $a * bc +$, which is (you guessed it) operator postfix notation.

As a way of honoring the Polish-born logician and philosopher, operator prefix notation was popularly named Polish Notation and operator postfix notation was named Reverse Polish Notation (RPN). And if we were to semantically read the $a\ *\ bc\ +$ RPN expression like a computer would, we'd read the expression as "*multiply a by the sum b plus c"* – which is very useful when translating a designer's infix notation (infix means that the operator is placed between its operands) to a data representation that a computer can use to read from and write to.

Which brings me back to the brilliance of the engineering that went into the HP 35S calculator. If you remember what I said about the 35S calculator, I mentioned that it was built to compute any arbitrary human readable expression using an operational stack and RPN. By now, you know all about RPN and if you think about it, RPN expressions like $a\ *\ bc\ +$ lend themselves quite nicely to one of the most widely and reliably used data structures in computer programming, the handy-dandy stack.

Stacks

Stacks are a great data structure – mostly because stacks are used in routines that span a wide range of use cases, including converting infix expressions to the postfix variety. Now, if you've never heard of a stack – you're missing out because it's awesome. Technically, computer scientists label a stack as an abstract data type that has two main operations, pushes and pops, which is super elegant if you think about it in a real-world context.

For example, if you have a whole bunch of plates that need to be washed, you may want to follow the simple method of pushing one plate onto the stack before popping them off to wash them.

Image credit: Wikipedia

This method of pushing and popping elements from a stack is also known as LIFO (last in first out), which is just how it sounds – push an element (like a plate) onto the stack and pop the last element from the top of the stack.

Like I said earlier in the blog, the HP 35S calculator uses an operational stack along with an RPN method of solving mathematical expressions written with infix notation, and those two elements were necessary parts of the solution I implemented for our designers.

So how does one parse infix notation so that the RPN and operational stack can computationally simulate an HP 35S calculator?

Shunting Yard

This is where the brilliance of Edsger Dijkstra's shunting yard algorithm enters the story. Shunting yards (also known as marshaling yards) are railway systems that enable operators to separate and reassemble freight trains. So if you think about a mathematical expression written with infix notation as a train, you can imagine every character of the expression being a single car. So a shunting yard would separate every element (or train car) from the infix notation train into separate stacks only to later reassemble the elements into a postfix (or RPN) train for computational purposes.

And for Demon Crush's math system, shunting yard lays at its heart because all a designer wants to type is something like $a * (b + c)$ to get a single output that the game loop can use

for gameplay events, like opening locked doors (to learn more about how doors play a big role in our level design, read our blog about keys).

A screenshot from Demon Crush during alpha testing of Kenzo approaching a locked door.

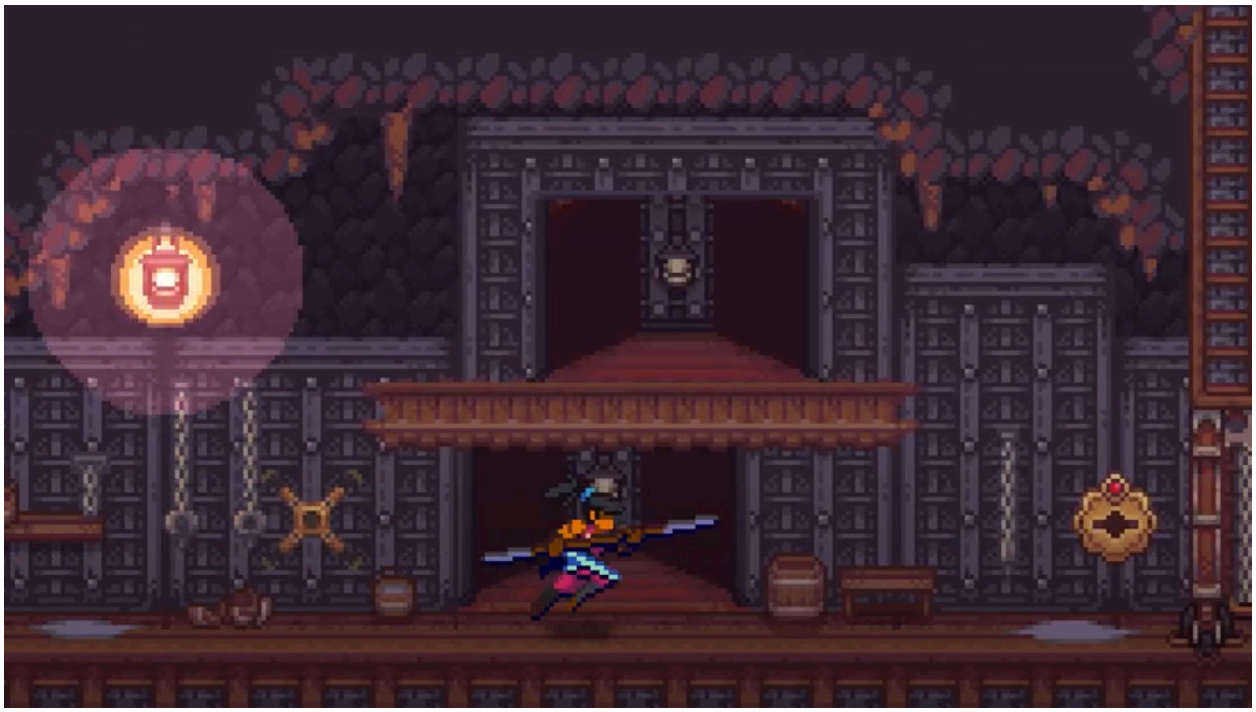
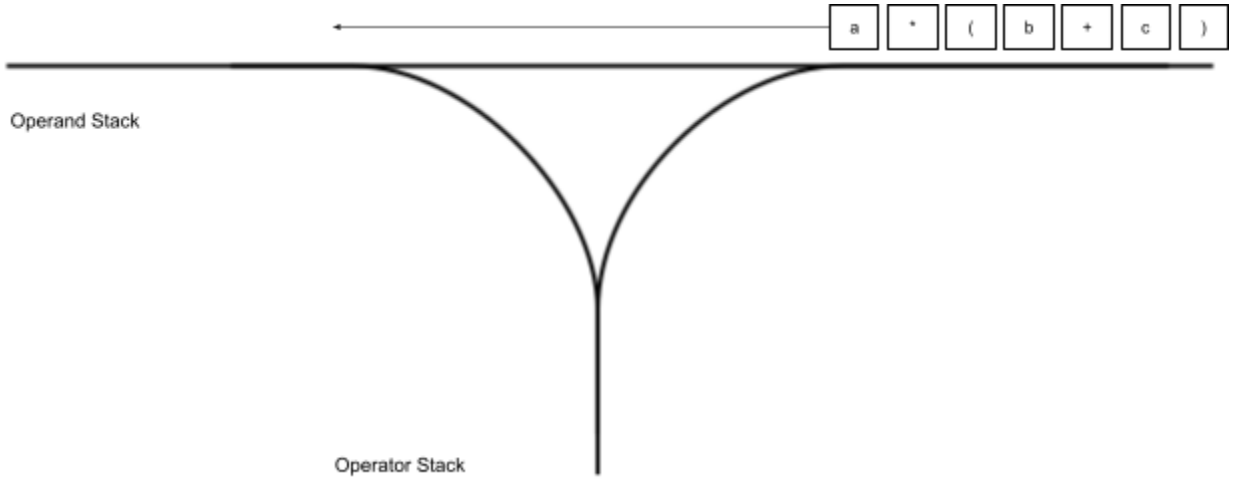


Image copyright: Reverting Castles

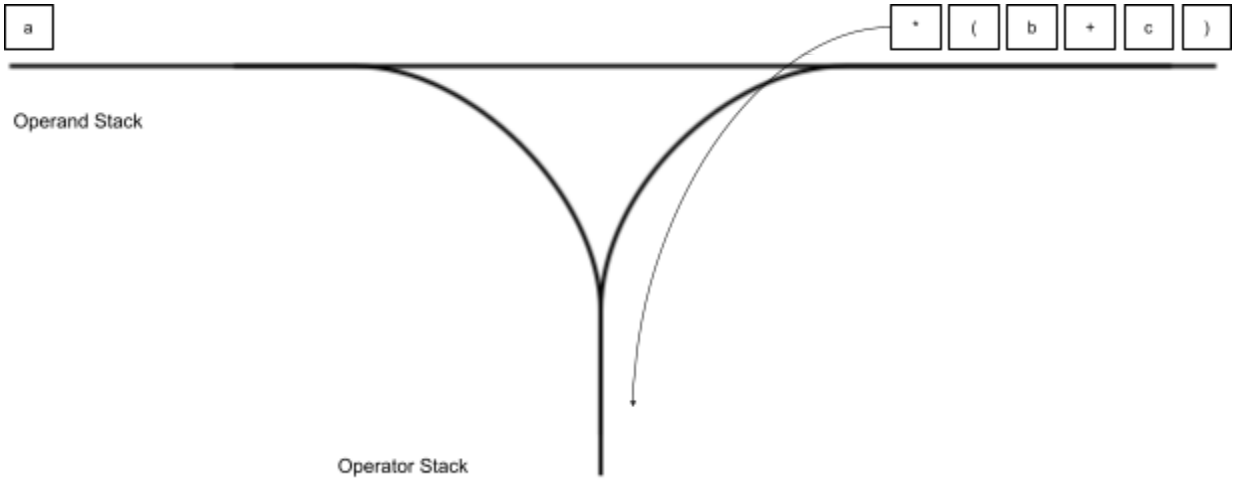
Bringing it all together, after the gameplay system inputs a string into the math system, the system separates an input string like $a * (b + c)$ and parses it into a tokenized character array like the following image.

And after processing the elements of the array with logic that accounts for operator precedence and for cases involving negative values, the system then separates the elements of the character array into two arrays, an operand stack and an operator stack using a three-way railroad junction.

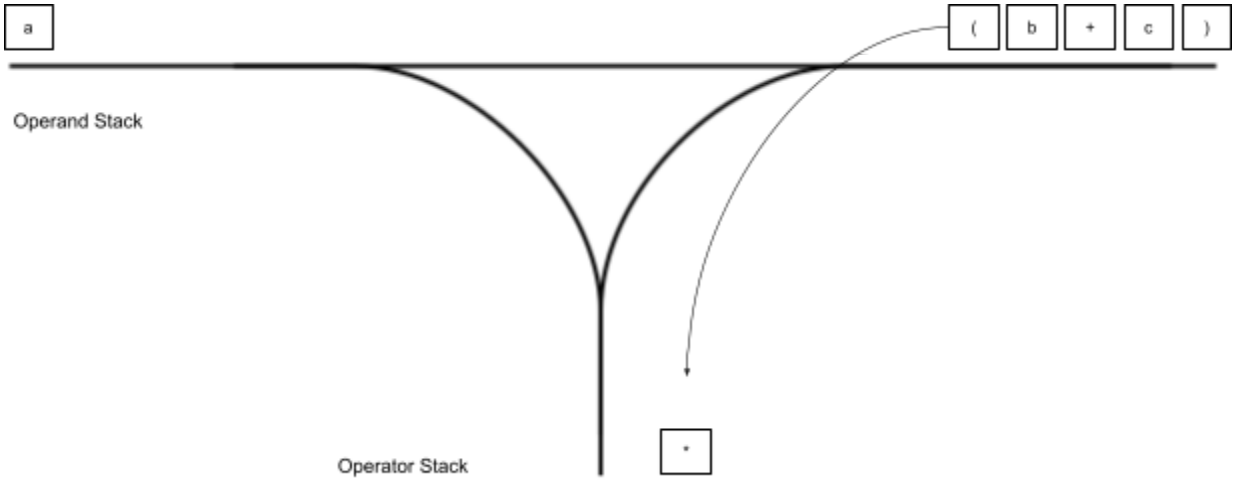
a gets pushed to the operand stack



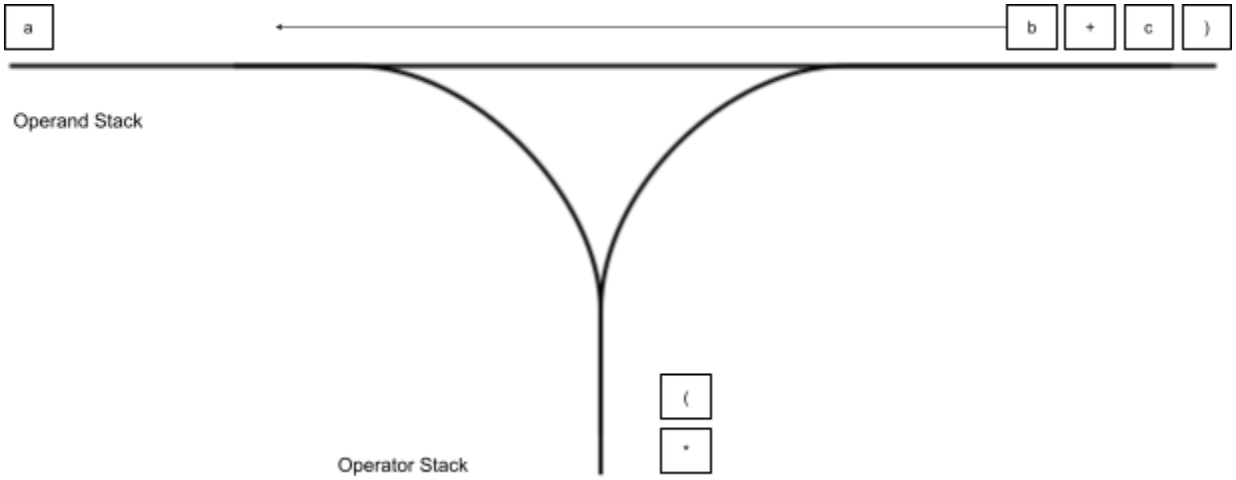
* gets pushed to the operator stack



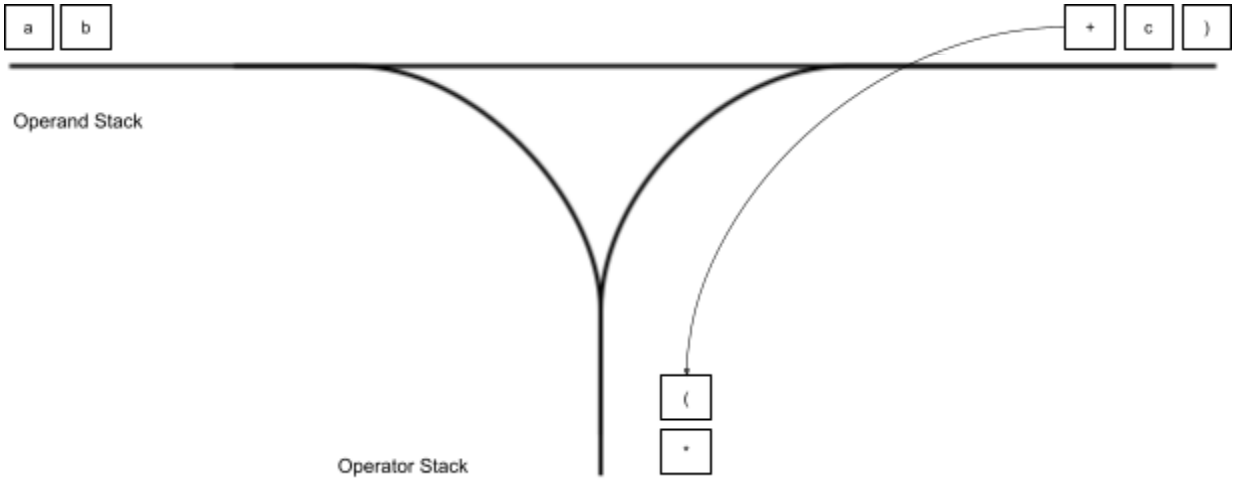
(gets pushed to the operator stack



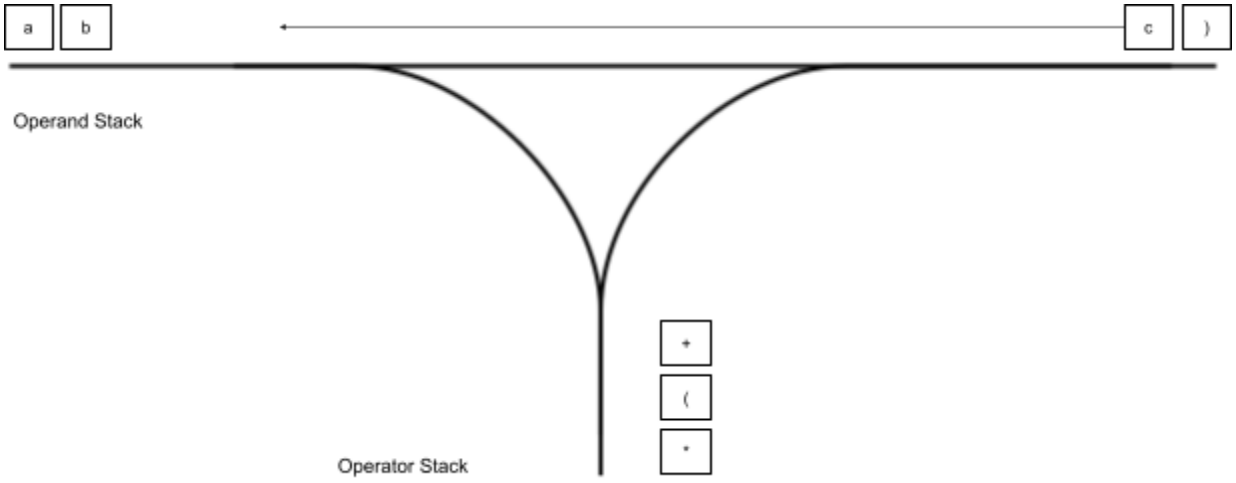
b gets pushed to the operand stack



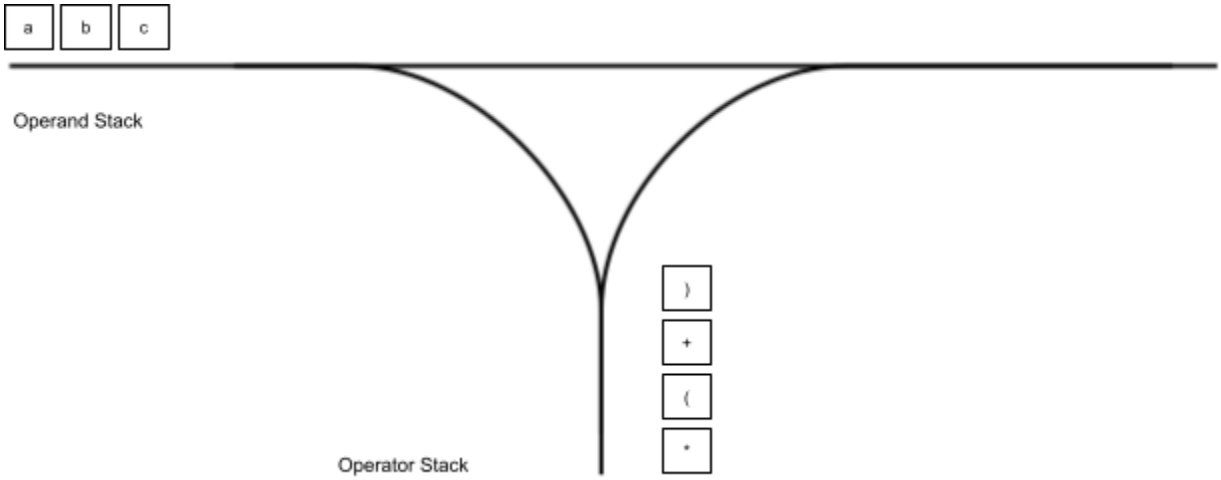
+ gets pushed to the operator stack



c gets pushed to the operand stack



) gets pushed to the operand stack to give us the following two stacks.



To account for special cases, like the input of logical expressions such as $x > y$, our implementation of the shunting yard method includes some general operator precedence routines that require our RPN calculator to operate on a single stack that is assembled during the computation of a single result.

At a very high-level, the) character informs logic in our RPN calculator that it needs to perform a computation before finding a (character, which means that $bc +$ gets computed before getting multiplies with a, such that the final output looks like the following $x = a * bc +$.

What's cool about this math system is that it can take any mathematical expression input that is represented with infix notation and output a single result to gameplay logic that makes use of RPN output in real-time.

Gameplay Features

We use the system to enable the following gameplay features.

- Key management

- Nezumo fight sequence

- Doors and levers

As you can see, theoretical concepts from mathematics and even from real world phenomena like railroads enable all sorts of gameplay features that make real-time interaction fun and interesting.

Engineering Blog: How Player Input Generates a Game Intensity Response

Personal Blog Sample
By
Robert Gervais

Hi, Demon Crush fans! Elias here to drop another engineering blog post (previously, I wrote about How a Calculator Enables Gameplay). Today, I'll write about how I compute game intensity to enhance Demon Crush's game feel.

Game feel is difficult to define, and even more difficult to compute.

Why?

Because game feel is as much a computer science endeavor as it is an artistic undertaking. That said, I leveraged computer science concepts to enhance the game feel of Demon Crush by building an input-response fuzzy controls system. Specifically, I leveraged player input to scientifically compute game intensity, which Demon Crush uses to generate an "artistic" response.

At a basic level, input is how the player controls what happens in the game and response is how Demon Crush interprets and reacts to input. For example, the most basic interaction between input and response is Kenzo's fluid movement when the player presses a specific set of inputs with their game controller.

Figure 1: Kenzo throws a kusarigama in response to player input.

If there is ever going to be an official definition for game feel, it will most probably lean on the basic control systems we design with input-response in mind. And when building the logic that drives the controls system responsible for implementing game feel, we often start by defining input predicates using first order logic.

Predicate Logic

First, a quick definition of predicate logic – technically, a predicate is a functional procedure that returns a truth value, and we typically write them as the following.

$$\text{Predicate}(P) \triangleq \text{FunctionalProcedure}(P) \wedge \text{Codomain}(P) = \text{bool}$$

Where you can have n-ary predicates and n-ary operations, which is very useful when designing input-response control systems that deliver game feel. In the case of Kenzo's kusarigama input-response control system, we can assume that it is a unary predicate, which is a predicate that takes one parameter (in this case, the "B" button input on the controller) to execute an operation, such that the definition looks like the following.

$$\text{UnaryPredicate}(P) \triangleq \text{FunctionalProcedure}(P) \wedge \text{UnaryFunction}(P)$$

And the pseudocode for Kenzo's kusarigama unary operation could look something like the following.

```
int32 IsBButtonPressed(int32 key)
{
    if(compareChar('B', key.toChar())) return 1; else return 0;
}
```

Now, let's say you want the kusarigama to respond to a combination of inputs such that a binary predicate enables Kenzo to execute a binary operation, consisting of throwing the kusarigama before pulling an enemy combatant towards Kenzo, like in the following figure.

Figure 2: Kenzo uses the kusarigama to pull the enemy towards him to throw a strike.

In this case, the player inputs the “B” button press and holds the button for a specific amount of time, which could look like the following kusarigama binary operation.

```
std::function<int (int32)> kusarigamaCombo() {return [](int32 key,
int32 time) { return key * time; };}
```

The great thing is that one can express a predicate in n-ary ways, which is what makes them so valuable when designing a control system that implements n-ary operations. And because combat is central to Demon Crush’s game feel, our Director (Richard, who also wrote a combat design article on Buffering and Canceling) spent a lot of engineering time to implement predicates for various operations – just take a moment to look at our technique scrolls to see how many incredible combinations you can execute when chaining attacks against an enemy force!

While developing the combat system, Richard reached out to me and asked if we could design a system that computes game intensity input so that we could use that intensity value to enhance the core game feel of Demon Crush – combat.

Fuzzy Logic

One of the use cases for computing game intensity is to dip the audio when the game is super intense, which enables players to focus on the combat that’s taking place on screen. Also, when we dip audio during intense moments of gameplay, we simulate what warriors go through in combat.

Because game feel is so difficult to define and because it's subjective and full of nuance, I had to think of a different way to represent the inputs to generate a single intensity value that enables the game to produce a response that's appropriate to the context of the game.

While predicate logic has its strengths when using classical set theory, such that $Xa(x) = 1\, if\, x \in A \vee 0\, if\, x \notin A\, where\, A \subseteq X$. And indeed, if you have to define a system where you have a series of discrete input-response controls, predicate logic is great at helping you define your controls, as with the combat input-response system that was built for Demon Crush. However, if you have a series of continuous input-response controls that need to be mapped to an intensity system, fuzzy logic presents a great framework to help define that control system because a fuzzy set A is a defined mapping where $A: X \rightarrow [0, 1]$ where $A(x)$ is the membership degree of x to the fuzzy set.

What does this all mean?

Basically, it means that for the most part, predicate logic effectively models linguistic certainty whereas fuzzy logic effectively models linguistic uncertainty. For example, if you look at the earlier example with Kenzo's kusarigama, we are able to build n-ary predicates to model classical sets of the input-response controls we need to implement for the game's combat control system. We can visualize predicate logic with the following picture.

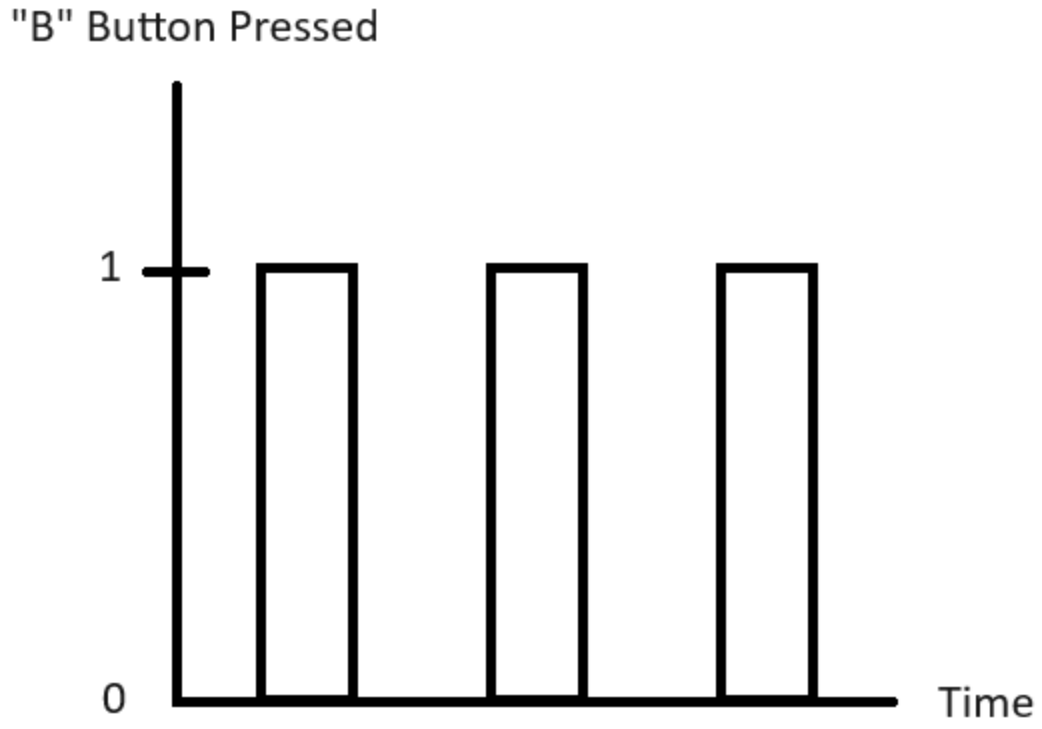


Figure 3: Predicate logic of a player pressing the "B" button over time.

However, when it comes to defining what makes an intense moment, there's a lot more linguistic uncertainty because there are differing degrees of intensity. For example, if we observe the example of the "B" button being pressed over time to deliver some kind of combo damage, we can hypothesize that intensity increases as the frequency of button presses increases over time. But how does someone define high intensity versus medium intensity or low intensity?

We can look at it visually with the following figure.

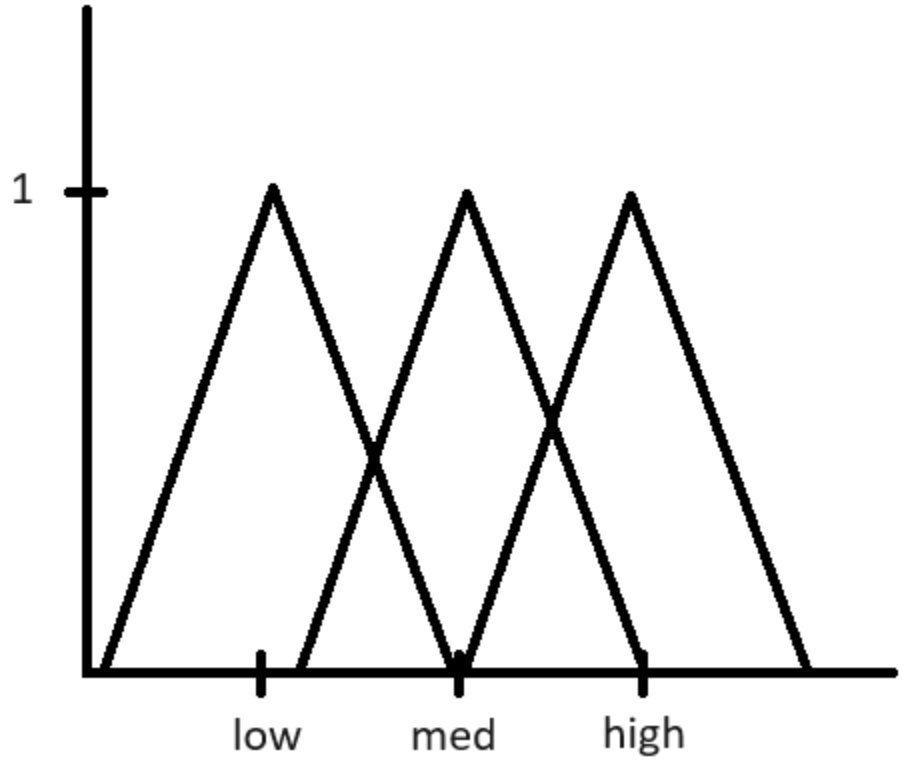


Figure 4: We can define how much the kusarigama combo damage contributes to intensity.

As shown in Figure 4, the mathematical framework of fuzzy logic helps us define a combo damage what equates to an intensity feeling that is linguistically low or medium or high (or somewhere in between) because one of the helpful properties of fuzzy logic is that every classical set is also a fuzzy set, and we can define $A \subseteq X$ as its characteristic function $\mu a(x) = 1\ if\ x \in A \vee 0\ otherwise$. This means that much like predicate logic and classical set theory, you get basic connectives in fuzzy logic and fuzzy set theory, such as complement, union, intersection, and inclusion – and of these connectives, I used intersections and unions a lot when building the intensity control system.

As a reminder, we denote intersection as $C = A \wedge B$ where we let $A,\ B \in \theta(x)$, which is defined as $C(x) = min\{A(x), B(x)\} = A(x) \wedge B(x),\ \forall x \in X$.

And we denote union as $C = A \vee B$ where we let $A,\ B\ \in\ \theta(x)$, which is defined as $C(x) = max\{A(x), B(x)\} = A(x) \vee B(x),\ \forall x \in X$.

Intensity System

To see this in action, here's a recording of one of the first sets of data that was used as input into the intensity system.

Figure 5: This was an early test of ComboDamage over TimeRemaining.

And in those early days, we were mostly using low, medium, and high qualifiers to define the amount of intensity the player would feel because we only defined a couple of gameplay data inputs and a single intensity output. This is a screenshot of some old gameplay input data that fed the first crisp output of the intensity system, which we labeled KID_Intensity (KID stands for Kenzo_Intensity_Data).

24	Old linguistic values below this line												
25		**Linguistic Vals**	**Low**				**Medium**				**High**			
26		*Trapezoid*	*x0*	*x1*	*x2*	*x3*	*x0*	*x1*	*x2*	*x3*	*x0*	*x1*	*x2*	*x3*
27		KID_AttackInputRate	0	5	10	15	10	15	20	25	20	25	50	100
28		KID_Movement	0	25	50	75	50	75	150	200	150	200	250	300
29	*Triple gradient (low, medium, high)*	KID_MovementZ	0	15	35	50	35	50	75	100	75	100	150	200
30		KID_DamageOverTime	0	2	4	6	4	6	10	15	10	15	20	25
31		KID_EnemiesOnScreen	0	1	3	4	3	4	6	7	6	7	9	10
32		KID_EnemiesOnScreenHealth	0	5	10	20	10	20	30	40	30	40	50	60
33		KID_PlayerInverseHealth	0	0.2	0.3	0.4	0.3	0.4	0.5	0.6	0.5	0.6	0.8	1

Figure 6: These linguistic values were derived from thousands of hours of playtests.

And as the gameplay inputs got more and more complex, we moved away from triangular linguistic values to trapezoidal linguistic values that eventually gave us more granular data that fed into the intensity system, which is shown in the following footage.

Figure 7: We recorded hundreds of videos showing the KID_* values and intensity output.

We ultimately built the system to integrate lots of gameplay data to compute the intensity of the movement and combat in our game, which helps determine the overall intensity that the player feels in the input-response loop that is core to the experience of Demon Crush, which you can see in this recently recorded footage.

Figure 8: The intensity system responds to player input in real-time.

If you listen closely, you can hear the game play more intense music as you smash through enemies. To do this, we hook intensity data to craft how the game responds to player input.

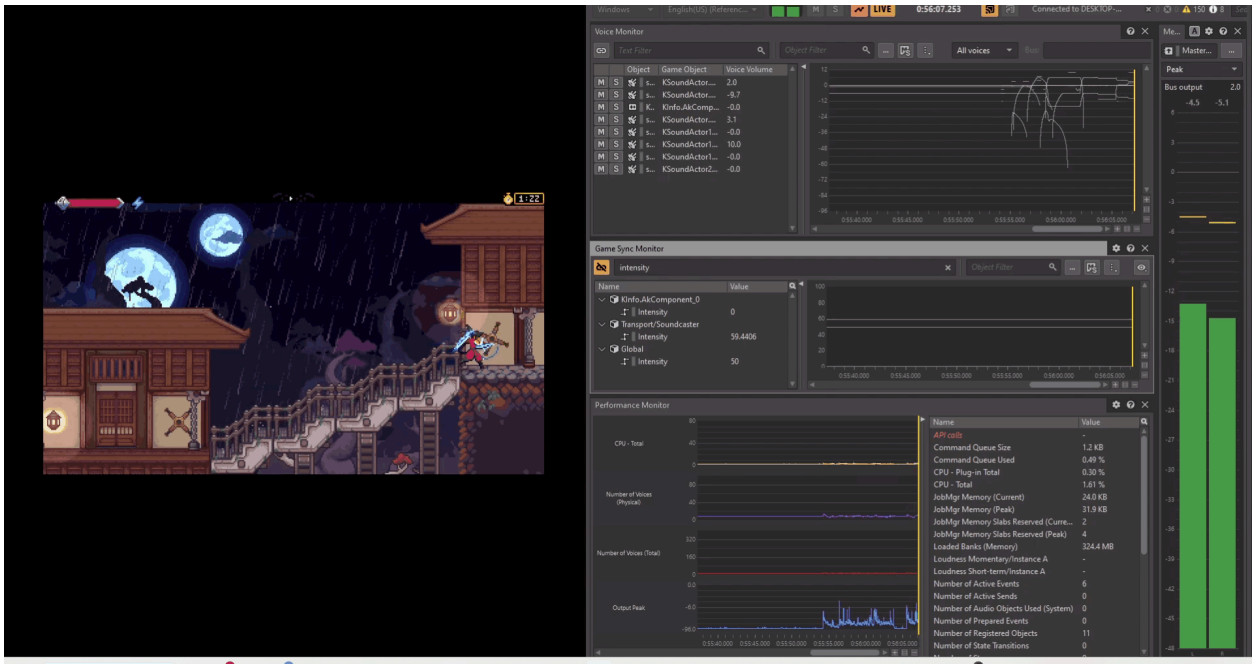


Figure 9: Our Sound Designer (Chris Sabin) plays a test level to observe intensity in real-time.

In addition to the current use of modulating the audio in-game, we plan on continuing to extend the use of the intensity system, so keep that intensity up and see (or hear) how Demon Crush responds to your player input!

Narrative Blog: Wordless Cutscenes, the Ukiyo-e Way

Personal Blog Sample
By
Robert Gervais

Elias here to talk about a really cool thing we've implemented for Demon Crush, Ukiyo-e Cutscenes! If you read one of our earlier blog posts about Speaking Without Speaking, you'll know that during the earliest phases of our game design process, we decided to tell the game's story without words. Conveying the game's narrative without relying on words is certainly a challenge, especially when it comes to building brief, wordless cutscenes for a game with lots of lore.

The Lore

Demon Crush's Game Writer, Kevin McKeon, continues to write a large amount of lore for the team to use as inspiration to build out the aesthetic features you see in-game. Every major character, every NPC, and even the flora and architectural elements have a reason to exist in Kenzo's world.

For example, here's some lore for the opening sequence, "*Plants in Kenzo's realm can consume Niei as well. Plants with a small amount of Niei, however, grow bigger and healthier, and store extra Niei in sap, seeds, leaves, or other parts of the plant, potentially giving them a faint blue tint, or even a slight glow. With greater amounts of Niei, though, the plant will burn itself out, grow into a misshapen version of itself, and ultimately die as a result*."

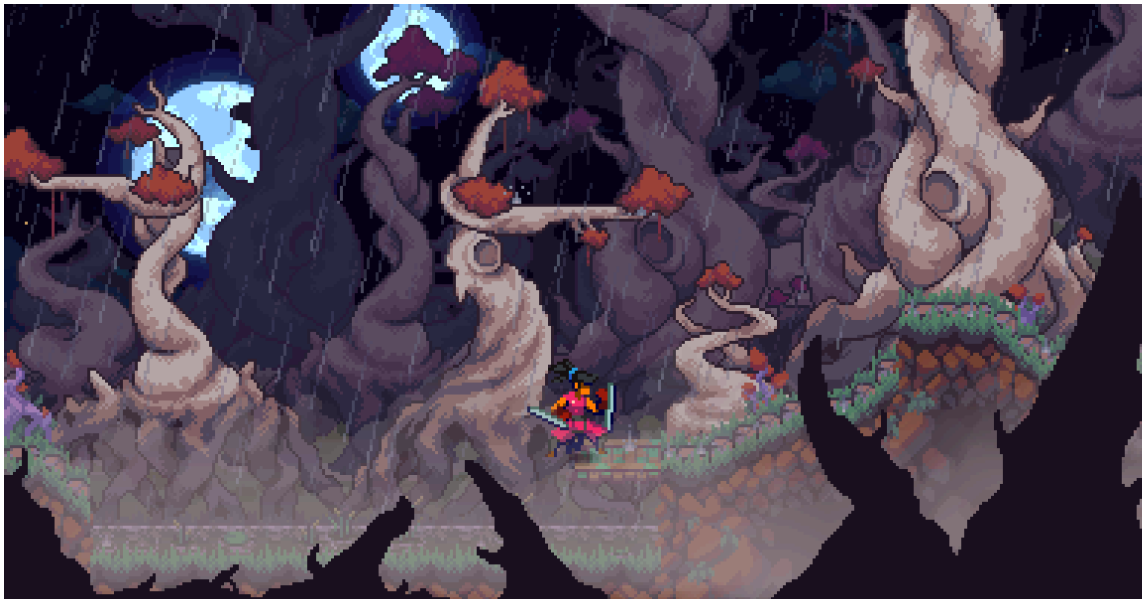


Figure 1: Niei is a primordial substance, which you can find everywhere, even in this forest.

Because the lore is so deep and covers a ton of history with interesting characters that have important roles in Demon Crush, the team made it a priority to reveal lore during a cutscene as players combat their way through Demon Crush's plot in order to achieve their goal of saving the people of Gochima Village.

The Plot

Which brings me to one of the primary reasons that cutscenes exist – they exist to advance the in-game plot in a way that either communicates an objective/goal or they exist to act as a reward after a player achieves the objective/goal.

For example, when the player defeats a boss, it feels great to be rewarded with a cutscene that either teases the next level or shows the outcome of achieving the goal of defeating the boss.

Figure 2: Kenzo prepares to finish Nezumo in this epic fight scene.

Since it is vital to advance the plot when designing player progression in Demon Crush, our cutscenes are an important part of our toolkit when it comes to communicating player progression through the use of standard plot devices, such as:

- **Disrupting Kenzo's equilibrium** in his World by threatening the peace of Gochima
- Giving Kenzo a strong **call to action** to defend the people of Gochima Village
- Having a Niei amulet, which is **an object of great power that must be overcome**
- Characterizing Nezumo as a **powerful antagonist** who must be defeated

- Providing a **plot twist** after defeating Nezumo.
    - *Play the demo to the end to see the twist unfold in the final cutscene after winning over Nezumo! It's worth the effort! I promise!*

These are some of the plot devices we use to move the story along as Kenzo fights to fulfill his destiny in Demon Crush.

The Characters

Which brings us to the importance of revealing characters and their traits (think of characterization) to give meaning to a player's choices being made to achieve Kenzo's objectives and his main goal of defeating Nezumo. And because Demon Crush is designed to be a linear narrative with a defined beginning, middle, and end, there is no need to reveal too much about a character's individual trait within a given cutscene. We do, however, make it a point to show the roles that the protagonists and antagonists play as we introduce the cast of allies, enemies, and NPCs in the game.

For example in the following frame, players will wonder why an Ashigaru warrior is chasing a Villager of Gochima.

Figure 3: Kenzo rushes to rescue this VIllager from the Ashigaru threat.

In fact when players watch this animation sequence, they may consciously or subconsciously wonder who the characters are, why they're in the scene, what they're looking for, and why the set and characters look this way – the effect of all these questions makes for cognitive dissonance in a player's mind, which is never good for a game that's all about player progression through a tightly scoped linear narrative experience.

Wordless Cutscenes

And this is where a well-crafted cutscene rushes in to rescue players from their cognitive dissonance. If you look at the following storyboard from one of our artists, you'll see that we're attempting to answer a few questions by:

- Revealing some of the game's lore, which takes inspiration from medieval Okinawa
- Advancing the plot by giving Kenzop a clear objective, which is to protect the people of Gochima Village from Nezumo and his army
- Revealing key characters, such as the antagonists and the protagonist's allies.

Figure 4: Our artist (Francisco Coda) illustrated this gorgeous storyboard.

After we storyboarded our cutscenes, the art team went to work and animated short films that revealed some lore, advanced the plot, and revealed our characters. What follows is a brief sequence of the opening cutscene.

Figure 5: A clip from an old opening cutscene that introduced the characters of Gochima Village.

And if you pay attention, this is roughly a 10 second clip taken from a cutscene that was about 1 minute long, which is unfortunately too long for a game as fast paced and as action-oriented as Demon Crush. So, after screening the cutscene with the rest of the team, we decided to add a couple of more requirements to cutscenes, specifically, they must be brief and they must be wordless, which goes back to one of the original development principles that went into designing the game.

So, when searching for a cutscene style, we specifically looked for a style that would meet the following requirements.

- Be brief
- Be wordless
- Reveal some lore
- Advance the plot
- Reveal character

And when we thought about it, the tradition of using illustrated panels made sense, because you can fulfill all these requirements without having to act out the performance. And that's the key, if you have to act out the scene, that requires a lot more time and nuance than if you simply animate your panels. And when we searched for a panel-based storytelling device that made sense for Demon Crush, we found it in the tradition of Ukiyo-e.

The Ukiyo-e Way

If you are familiar with the name Hokusai and his famous illustration called “The Great Wave”, you will know about the ancient tradition of Ukiyo-e. And aside from the fact that we can briefly display a Ukiyo-e tryptic during a cutscene event, we use the many layers in a Ukiyo-e illustration to communicate the lore, advance the plot and reveal the characters of Demon Crush.

For example the following cutscene reveals some lore of Demon Crush with Kenzo being charged by Niei, it advances the plot by giving Kenzo a clear call to action to save the Villagers of Gochima, it introduces Nezumo and his warriors as the antagonists, and it introduces the High Priest and the monks as Kenzo’s protagonist allies – all while being brief and wordless.

Figure 6: Our artist (Sergio Suarez) illustrated and animated these Ukiyo-e panels beautifully.

And because Ukiyo-e feels like a natural fit in the world of Demon Crush, the cutscene clearly feels like a call to action for players to begin their journey. Trust me when I say that later cutscenes will bell like a reward as you fight your way to victory over Nezumo.

So, what are you waiting for?

Get in there, beat those levels, and watch Kenzo’s story unfold in our expertly handcrafted and wordless cutscenes, the Ukiyo-e way.