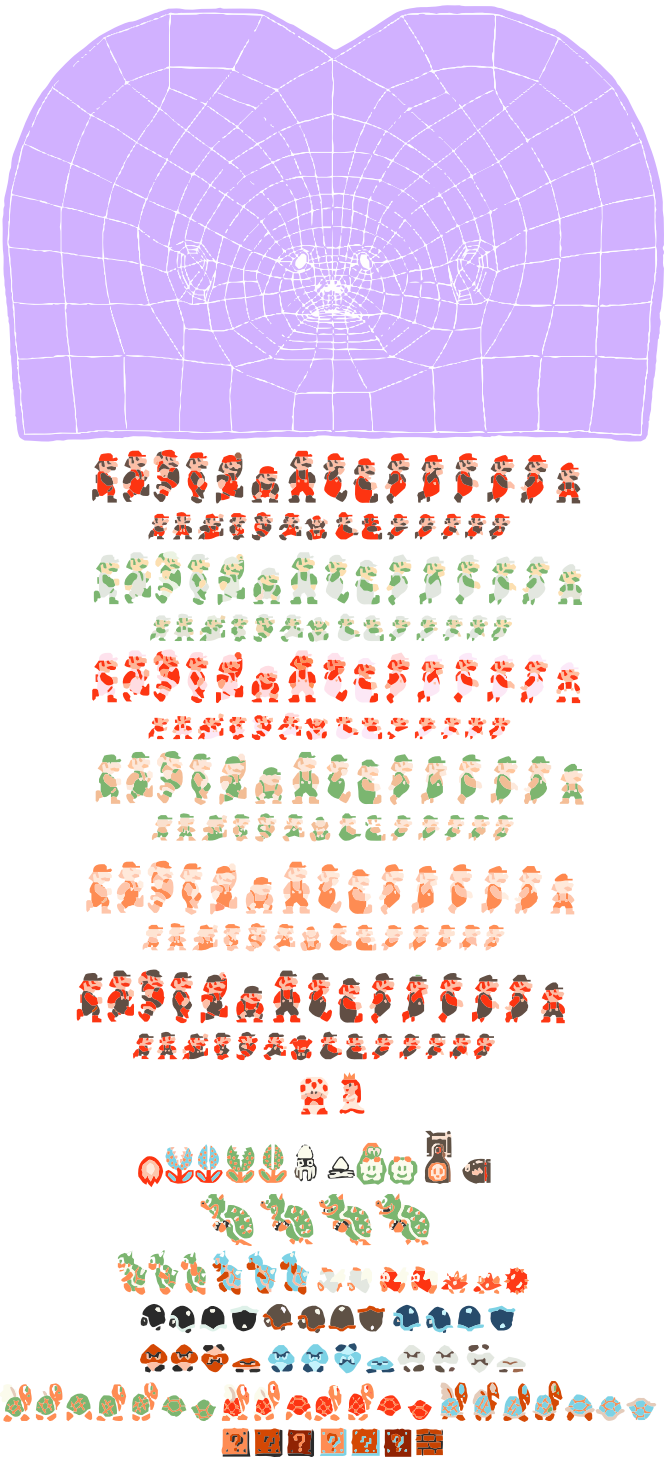
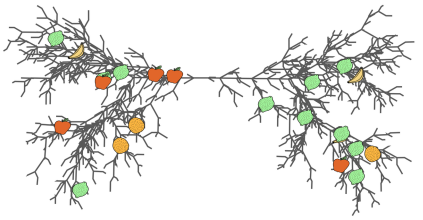
Creeping Bug



Top (3D Mesh with UV Grid): A polygonal model is shown with a superimposed UV grid. The grid flattens a surface into 2D coordinates, assigning each vertex to a location in texture space. This mapping determines how an image will be wrapped onto the surface.
Bottom (Texture Atlas): A sprite sheet provides the 2D image data: character animations, items, and enemies. Each region of the sheet corresponds to a set of UV coordinates. During rendering, the engine looks up the correct portion of the texture based on UV mapping and projects it onto the surface, aligning 2D artwork with 3D geometry.

Creeping Bug

Minecraft's Creeper began as a mistake. Markus Persson entered the pig's dimensions backwards, producing a tall, thin figure that looked nothing like an animal. Instead of deleting it, he added a texture, a frown, and an explosive routine borrowed from the game's block-destruction code. The result was a creature that crept silently, paused, and detonated. A simple modeling error became Minecraft's most iconic enemy.



Creeper Origin Story ◦ Pig Model Bug ◦ Entity Component Systems ◦ AI Task Composition ◦ Game Engine Evolution ◦ Bugs Becoming Features ◦ Street Fighter Combos ◦ Rocket Jumping ◦ Minecraft Redstone Logic ◦ Emergent Gameplay ◦ Fail-Soft Design

"Once you've got a task to do,
it's better to do it than live with the fear of it."
— Logen Ninefingers, 575 AU

"The first 90 percent of the code accounts for
the first 90 percent of the development time.
The remaining 10 percent of the code accounts for
the other 90 percent of the development time."
— Tom Cargill, 1985

Creeping Bug

Minecraft (2009) began as the independent project of Markus "Notch" Persson, a Swedish programmer inspired by sandbox-building games such as Infiniminer (2009) and Dwarf Fortress (2009). He began development on 10 May 2009, using Java with the Lightweight Java Game Library (LWJGL), releasing the first public version seven days later on 17 May. Early versions focused on creative construction in a procedurally generated block world (Perlin, 1985), appealing to players' innate curiosity and design instincts. Persson incorporated player feedback rapidly, adding survival mechanics, crafting, hostile mobs, and multiplayer support. Development was open and iterative, creating a strong early community on forums such as TIGSource.

In 2010, Persson founded Mojang to support ongoing development. Jens Bergensten (known as Jeb) joined and later took over lead development. The game officially launched on 18 November 2011 at MineCon in Las Vegas, having already sold millions of copies in beta. Mojang maintained a low-friction sales model: a single upfront purchase, no digital rights management (DRM), and support across multiple operating systems. This simplicity contributed to rapid global adoption.

In 2014, Microsoft acquired Mojang and Minecraft for $2.5 billion. At that point, Minecraft had sold over 54 million copies. Since then, the game has expanded to nearly every platform, including consoles, mobile, and VR. As of October 2023, Minecraft has sold over 300 million copies, making it the best-selling video game of all time. In 2021, it reported more than 140 million monthly active users. It has been used in classrooms, cited in academic studies on spatial reasoning and collaboration, and remains a major force in online content creation — especially on YouTube, where Minecraft videos have accumulated over 1 trillion views.

Minecraft's success stems from a blend of simplicity and depth. It offers intuitive core mechanics (block placement, mining, crafting) with nearly unlimited creative potential. Procedural generation ensures novelty, while redstone logic introduces programmable mechanics akin to electrical engineering. The game fosters personal expression, exploration, and emergent storytelling (Gardner, 1970). Its low system requirements and modding support further extended its reach and longevity. Minecraft's development history is a case study in iterative design, community engagement, and the creative payoff of systems-first thinking.

In 1962, game developers wrote code that directly controlled individual pixels. *Spacewar!* on the PDP-1 computed each dot's position through direct arithmetic: $x + dx$, $y + dy$. The electron beam drew vectors where the calculations specified. No abstraction layers existed between the programmer's calculations and the phosphor display. Each spaceship consisted of six numbers in memory — position, velocity, angle, and fuel. The game loop ran sixty times per second — read switch states from the control panel, update positions by adding velocities, subtract fuel for thrust, apply gravity as a constant acceleration toward the center, check if position vectors intersected for collisions, then send the computed coordinates to the vector display. The entire program fit in 4 KB of memory.

©2026 David H. Silver, CC BY-NC 4.0

Early arcade machines hardcoded every behavior into separate routines. *Space Invaders* (1978) implemented each alien type with its own movement function, its own collision detection, its own point calculation. Moving the bottom row required one function that decremented x-coordinates and checked the left boundary. Moving the middle row used a different function with different speed constants. The top row had its own handler. When any alien touched the screen edge, specific code executed to move all aliens down one row and reverse the direction flag. Adding a new enemy type meant writing new functions for movement, new functions for collision detection, new functions for scoring — touching every system in the game. The code grew linearly with content.

Programmers began utilizing these patterns of repetition. Every moving object needed position and velocity, every visible object needed drawing routines, every destructible object needed health values — the same structures repeated across every game. In *Adventure* (1979), Warren Robinett faced the Atari 2600's 128 bytes of RAM. He couldn't afford separate code for each object type. Instead, he consolidated repeated elements into a single object handler. Dragons, bats, keys, and swords were entries in an object table, each storing position, size, color, and a behavior ID. The behavior ID indexed into a jump table of function pointers. Object `0x1A` (the yellow key) had behavior type `0x04` (can be picked up). Object `0x0E` (the red dragon) had behavior type `0x07` (chase player). A single collision detection routine handled all interactions by comparing behavior IDs, and a single drawing routine rendered all objects by reading their size and color. What would have required dozens of specialized handlers fit into 128 bytes of RAM.

When Toru Iwatani designed *Pac-Man* (1980), he gave all four ghosts the same movement code but different target selection algorithms. Blinky (red) targeted Pac-Man's current tile. Pinky (pink) targeted four tiles ahead of Pac-Man in his facing direction. Inky (cyan) computed a complex target: take the position two tiles in front of Pac-Man, draw a vector from Blinky to that position, then double it. Clyde (orange) chased Pac-Man when more than eight tiles away but fled to his home corner when closer. Four ghosts from one movement function with different target coordinates.

The 1990s brought object-oriented programming to game development (Dahl & Nygaard, 1966). *Doom* (1993) designed its actors through inheritance hierarchies. Every monster derived from a common base class containing position, health, and state machine logic. The imp and the baron of hell executed identical state machine code. They differed only in their data tables — health points (60 versus 1000), projectile type (fireball versus plasma ball), pain chance (200/256 versus 50/256). State machines were data. The imp's fireball attack was state S_TROO_ATK3: display sprite TROOF, duration 8 tics, action function A_TroopAttack, next state S_TROO_1. New monsters were created by mixing existing action functions with new sprites and parameters. The Revenant combined A_SkelMissile (fire homing missile) with A_SkelFist (punch if close). The Archvile combined A_VileChase (resurrect dead monsters) with A_VileAttack (immolating flame attack). No new code required — just new data tables.

By 2000, inheritance hierarchies had limitations. A FlyingEnemy class couldn't share code with a FlyingProjectile without multiple inheritance. A FireGolem pulled from both Golem and FireCreature, creating diamond ◇ inheritance — when a class inherits from

two classes that themselves inherit from the same base class. Deep hierarchies were fragile. Changing Animal broke Dog which broke FlyingDog which broke FireBreathingFlyingDog. Entity-component systems replaced them. An entity was just an ID number. Components were bags of data: `Position {x: 5, y: 10, z: 3}`, `Velocity {dx: 1, dy: 0, dz: 0}`, `Sprite {texture: "goblin.png"}`, `Health {current: 30, max: 30}`, `AI {behavior: "aggressive"}`. Systems were functions that processed components. MovementSystem iterated through all entities with Position and Velocity, updating positions. RenderSystem drew all entities with Position and Sprite. DamageSystem processed collisions for entities with Health. Adding flight to a goblin meant adding a Flying component. Making a barrel explode meant adding an Explosive component. A flying, exploding, invisible barrel needed no new class — just combine Flying, Explosive, and remove the Sprite component.

Game engines were standardized architectures. id Tech (1993), Unreal Engine (1998), and Unity (2005) provided complete frameworks — rendering pipelines with shaders and occlusion culling, physics engines with collision detection and constraint solvers, audio systems with 3D spatialization, networking layers with client prediction and lag compensation. Developers no longer built engines from scratch. They configured existing systems, wrote gameplay scripts, created art assets. A game was configuration data plus custom logic, running on someone else's foundation. Most games using these engines still required teams of specialists to craft specific experiences. Some games took a different approach, providing tools for players to build their own content within the game's systems.

Minecraft used these principles extensively. Markus Persson wrote no hardcoded mining animations, no specific crafting sequences, no predetermined progression. He built basic systems. Blocks had properties — hardness, tool requirements, light emission, update behaviors. Items had functions — dig block, place block, damage entity. Entities had composable AI tasks: place blocks, break blocks, update neighbors. Players built cities, computers, musical instruments, working calculators. Redstone dust carried signals up to 15 blocks. Torches inverted signals — powered input produced unpowered output. Repeaters delayed signals by 1–4 ticks. Pistons pushed blocks when powered. Players built logic gates — NOT gates (Shannon, 1938) from single torches, OR gates from merged dust lines, AND gates from torch arrays, memory cells from piston feedback loops (Eccles & Jordan, 1919). Players even built 8-bit CPUs. The programmer set the stage, the players set the game.

Entities in *Minecraft* used composition. The base Entity class defined position, velocity, and bounding box. LivingEntity added health and damage handling. Mob added a list of AI tasks, small behavior programs that executed each tick. Tasks were objects with simple interfaces — `shouldExecute()` to check if the task should run, `startExecuting()` to initialize, `updateTask()` to perform the behavior. A zombie's task list contained `AttackPlayerTask` (priority 2), `WanderTask` (priority 5), `LookAtPlayerTask` (priority 8). A sheep had `EatGrassTask`, `FollowParentTask`, `PanicTask`. The cow combined wandering, following players holding wheat, and panicking when hurt. The skeleton combined attacking players, fleeing from wolves, and avoiding sunlight. Every mob was assembled from the same library of parts.

Among these assembled creatures was the Creeper — *Minecraft*'s most recognizable enemy. Silent, green, explosive. It approaches players with little warning and detonates on proximity, destroying carefully built constructions. Unlike zombies that moan or skeletons that rattle, the Creeper moves in complete silence until its final hiss. The Creeper began as a pig. In 2009, Persson was implementing farm animals and creating a pig model. In his 3D modeling program, he entered the creature's dimensions. The pig required length 2.0, height 1.0, width 1.0 — a horizontal rectangle with stubby legs. But when typing the values, Persson accidentally swapped length and height, entering height 2.0, length 1.0. Instead of a quadruped, he got a vertical pillar with four tiny legs at the bottom. The model file loaded without error — *Minecraft*'s model loader accepted any valid vertex data. The renderer displayed exactly what it received — a tall, thin creature standing upright. Persson found the error amusing. He textured it green, added a frowning mouth, adjusted the legs to look more like feet. He kept it as a joke, then made it explode. He copied TNT's explosion code — remove blocks within radius R, damage entities with distance-based falloff, spawn item entities for destroyed blocks. He bound this to proximity detection borrowed from zombie AI — if distance to player less than 3 blocks, start countdown timer. After 1.5 seconds, detonate.

The Creeper was not the only accident that stuck. Bugs have shaped game design as much as deliberate decisions.

Street Fighter II (1991) had combos through a programming oversight. During development, producer Noritaka Funamizu discovered that the recovery time after certain moves was shorter than the hit-stun inflicted on opponents. By timing inputs precisely, players could land a second attack before the opponent recovered from the first. The window was narrow — frame-perfect in some cases — and the developers assumed players would rarely exploit it. But location test players began discovering two-hit and three-hit sequences. Capcom watched players develop increasingly complex chains — jump kick into standing fierce into special move. Instead of patching the timing windows, Capcom kept the oversight and built around it. Every subsequent fighting game implemented deliberate combo systems with cancels, links, and chains — mechanics that originated from a gap between two timing values.

Quake (1996) introduced rocket jumping through physics engine oversight. The game calculated explosion damage and knockback for all entities within the blast radius — including the player who fired the rocket. Players discovered that firing at their feet while jumping added the explosion's upward force to their jump velocity, reaching otherwise inaccessible platforms. The technique required precise timing and cost health, creating a risk-reward trade-off that became a staple of competitive first-person shooters.

Grand Theft Auto (1997) began development as *Race'n'Chase*, a straight racing game with police chases. During testing, a quirk of the police AI made them impossibly aggressive. Instead of trying to box players in, they rammed at full speed. Testers found themselves spending more time fleeing from psychotic police than racing. The chaotic pursuits were more entertaining than the intended gameplay. DMA Design redesigned the entire game around the bug, leaning into the chaos rather than fixing it. The racing game became a crime sandbox.

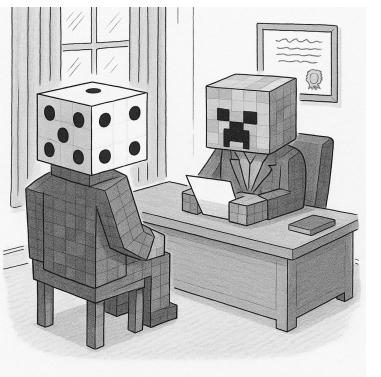
Devil May Cry (2001) originated from a scrapped *Resident Evil 4* prototype. The prototype's combat engine felt too fast and stylish for survival horror — characters launched enemies into the air, chained attacks mid-flight, and juggled opponents with gunfire. Rather than discard the prototype, Capcom built an entire franchise around what didn't fit, rating players on combo variety and rewarding style over caution.

Space Invaders (1978) had increasing difficulty through hardware limitations. Tomohiro Nishikado programmed the aliens to move at constant speed, but the Intel 8080 processor couldn't maintain consistent frame rates. With fifty-five aliens on screen, the game ran slowly. As players destroyed aliens, the processor had fewer sprites to update, causing the remaining aliens to move faster. Nishikado kept the unintended acceleration — what began as a hardware limitation became the game's defining tension, the last alien darting across the screen at terrifying speed.

Tribes (1998) had movement physics bugs. Players discovered that rapidly tapping jump while descending slopes prevented the normal friction from applying. Each jump reset the friction calculation before it could slow the player. This "skiing" technique allowed players to traverse maps at tremendous speed, transforming methodical infantry combat into high-speed aerial warfare. Dynamix kept it, designing maps with long slopes, adding routes specifically for skiing, balancing weapons around high-speed combat.

Silent Hill (1999) used fog to hide hardware limitations. The PlayStation couldn't render distant polygons without severe popup and texture warping. Instead of reducing draw distance with traditional fog walls, Team Silent implemented thick, volumetric fog that moved and swirled. The fog became the game's defining atmosphere, hiding threats and turning every few meters of visibility into a source of dread.

Super Smash Bros. Melee (2001) had physics oversights. Wavedashing happened when air dodging diagonally into the ground preserved momentum while landing, causing characters to slide. Players could attack while sliding, opening new approach options.



"Your resume is... interesting. It mentions 'extensive experience in probability determination'?"

The Creeper

Entity models in early Minecraft were defined using bounding boxes in Java. Each model was specified via constructor calls such as:

```
ModelBox(name, x, y, z,
         length, height, width).'
```

For passive mobs, proportions typically followed:

$$\texttt{length} > \texttt{height},$$
$$\texttt{length} \approx 1.2,$$
$$\texttt{height} \approx 0.9.$$

Due to a developer error, `length` and `height` were swapped:

$$(\ell,\ h,\ w) \mapsto (h,\ \ell,\ w).$$

This produced a tall, narrow model. Define the pig's geometry as:

$$G_{\text{pig}} = [0, \ell] \times [0, h] \times [0, w],$$

and the resulting Creeper geometry as:

$$G_{\text{creeper}} = [0, h] \times [0, \ell] \times [0, w].$$

With $\ell \approx 2.0$, $h \approx 1.0$, the resulting model appeared upright and columnar.

Visual Mutation and Face Topology

The model was assigned Minecraft's leaf texture. A simple face overlay was defined on the front-facing surface using pixel coordinates:

$$\texttt{eyes}: \quad (x, y) \in \{(3, 12), (10, 12)\},$$
$$\texttt{mouth}: \quad (x, y) \in \{(5, 6), (4, 4), (10, 4)\}.$$

AI Composition and Swell Behavior

The Creeper's actions derive from prioritized AI goals:

`Goal 1: LookAtPlayer` (range $R = 6$),
`Goal 2: ApproachPlayer` (speed $v = 0.2$),
`Goal 3: StartSwell` (trigger $r < r_s$),
`Goal 4: Explode` (delay $\tau = 1.5\,\text{s}$).

Explosion occurs if the player remains within range $r_s = 2.5$ blocks for the full fuse duration. Let $r(t)$ denote distance to the player at time t. Then:

$$\texttt{if}\ \ r(t) < r_s\ \forall t \in [t_0, t_0 + \tau], \texttt{Explode()}.$$

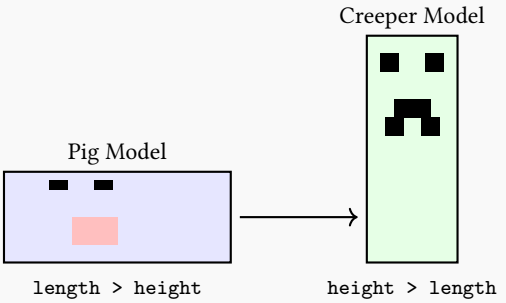
Damage output is modeled by radial falloff:

$$D(x) = \max\left(0, E_{\max}\left(1 - \frac{\|x - x_0\|}{R}\right)\right),$$

where x_0 is the explosion center and $R \approx 7$ blocks in air.

Reconstruction in TikZ

The diagram below illustrates the result of axis misassignment during model construction. On the left, the intended pig model appears with a horizontal body and frontal facial features. On the right, the same parameters are rendered with the `length` and `height` values swapped. This inversion laid the foundation for the Creeper's distinctive form.



References:
Persson, M. (2009). *Initial Geometry and Entity Source*. Mojang.
Minecraft Wiki. (2025). Entity Models and AI Mechanics. `https://minecraft.wiki`