Math API

Mathematical utilities

Overview

Origami Engine uses JavaScript's built-in `Math` object for mathematical operations. Additional game-specific math functions are also available.

Random Functions

`random()`

Returns random float between 0 and n.

Syntax: `random(n)`

Arguments:

- `n` (number) - Maximum value (exclusive)

Returns: `number` - Random value [0, n)

Example:

```
create(): void {
  // Random speed (0.0 to 5.0)
  this.speed = random(5);

  // Random position
  this.x = random(room_width);
  this.y = random(room_height);

  // Random chance (30%)
  if (random(1) < 0.3) {
    this.dropItem = true;
  }

  // Random color component
  const r = Math.floor(random(256));
  const g = Math.floor(random(256));
  const b = Math.floor(random(256));
}
```

`irandom()`

Returns random integer between 0 and n (inclusive).

Syntax: `irandom(n)`

Arguments:

- `n` (number) - Maximum value (inclusive)

Returns: `number` - Random integer [0, n]

Example:

```
create(): void {
  // Random dice roll (0-5, then add 1 for 1-6)
  const roll = irandom(5) + 1;

  // Random health (50-100)
  this.health = 50 + irandom(50);

  // Random choice from array
  const colors = ['red', 'green', 'blue', 'yellow'];
  const randomColor = colors[irandom(colors.length - 1)];

  // Random spawn point
  const spawnPoints = [
    {x: 100, y: 100},
    {x: 200, y: 150},
    {x: 300, y: 100}
  ];
  const spawn = spawnPoints[irandom(spawnPoints.length - 1)];
  this.x = spawn.x;
  this.y = spawn.y;
}
```

`random_range()`

Returns random float between min and max.

Syntax: `random_range(min, max)`

Arguments:

- `min` (number) - Minimum value
- `max` (number) - Maximum value

Returns: `number` - Random value [min, max]

Example:

```
create(): void {
  // Enemy speed (2.0 to 5.0)
  this.speed = random_range(2, 5);

  // Spawn in specific area
  this.x = random_range(100, 500);
  this.y = random_range(100, 300);

  // Random timer
  this.attackDelay = random_range(60, 180); // 1-3 seconds

  // Random angle
  this.direction = random_range(0, 360);

  // Random scale
  this.image_xscale = random_range(0.8, 1.2);
  this.image_yscale = random_range(0.8, 1.2);
}
```

JavaScript Math Functions

Use native `Math` object for calculations:

Basic Operations

```
// Absolute value
const distance = Math.abs(this.x - target.x);

// Round down
const tileX = Math.floor(this.x / 32);

// Round up
const tileY = Math.ceil(this.y / 32);

// Round to nearest
const rounded = Math.round(this.health);

// Min/Max
const finalDamage = Math.max(0, damage - armor); // Can't go below 0
const cappedHealth = Math.min(this.health, this.maxHealth);
```

Powers and Roots

```
// Square root
const dist = Math.sqrt(dx * dx + dy * dy);

// Power
const damage = Math.pow(this.level, 2);

// Exponential
const curve = Math.exp(this.time * 0.1);
```

Trigonometry

Note: JavaScript uses **radians**, not degrees.

Convert:

```
// Degrees to radians
const radians = degrees * Math.PI / 180;

// Radians to degrees
const degrees = radians * 180 / Math.PI;
```

Functions:

```
// Sine/Cosine
const waveY = Math.sin(this.time * 0.1) * 50;
const circleX = Math.cos(this.angle * Math.PI / 180) * 100;

// Tangent
const slope = Math.tan(this.angle * Math.PI / 180);

// Inverse functions
const angle = Math.atan2(dy, dx) * 180 / Math.PI;
const arcsin = Math.asin(value);
const arccos = Math.acos(value);
```

Clamping and Limits

```
// Clamp value between min and max
function clamp(value: number, min: number, max: number): number {
  return Math.max(min, Math.min(max, value));
}

// Usage
this.speed = clamp(this.speed, 0, 10);
this.health = clamp(this.health, 0, this.maxHealth);
```

Common Math Patterns

Distance Formula

```
// Manual distance calculation
const dx = this.x - target.x;
const dy = this.y - target.y;
const dist = Math.sqrt(dx * dx + dy * dy);

// Or use built-in
const dist = point_distance(this.x, this.y, target.x, target.y);
```

Normalize Vector

```
// Make vector length 1
const length = Math.sqrt(this.hspeed * this.hspeed + this.vspeed * this.vspeed)
if (length > 0) {
  this.hspeed /= length;
  this.vspeed /= length;

  // Scale to desired speed
  this.hspeed *= this.desiredSpeed;
  this.vspeed *= this.desiredSpeed;
}
```

Linear Interpolation (Lerp)

```
// Smooth transition from a to b
function lerp(a: number, b: number, t: number): number {
  return a + (b - a) * t;
}

// Usage
step(): void {
  // Smooth camera follow (20% per frame)
  this.cameraX = lerp(this.cameraX, this.targetX, 0.2);
  this.cameraY = lerp(this.cameraY, this.targetY, 0.2);
}
```

Approach

```
// Move value towards target by max amount
function approach(current: number, target: number, amount: number): number {
  if (current < target) {
    return Math.min(current + amount, target);
  } else {
    return Math.max(current - amount, target);
  }
}

// Usage
this.speed = approach(this.speed, 0, 0.5); // Slow down
this.alpha = approach(this.alpha, 1, 0.1); // Fade in
```

Wrap Around

```
// Wrap value within range
function wrap(value: number, min: number, max: number): number {
  const range = max - min;
  return ((value - min) % range + range) % range + min;
}

// Usage
step(): void {
  // Wrap around screen
  this.x = wrap(this.x, 0, room_width);
  this.y = wrap(this.y, 0, room_height);

  // Wrap angle (0-360)
  this.direction = wrap(this.direction, 0, 360);
}
```

Wave Patterns

```
// Sine wave
step(): void {
  this.waveTime += 0.1;
  this.y = this.baseY + Math.sin(this.waveTime) * 20;
}

// Cosine wave
step(): void {
  this.x = this.centerX + Math.cos(this.time * 0.05) * 100;
}

// Combined waves
step(): void {
  this.x = Math.sin(this.time * 0.1) * 100;
  this.y = Math.cos(this.time * 0.15) * 80;
}
```

Easing Functions

```
// Ease in (slow start)
function easeIn(t: number): number {
  return t * t;
}

// Ease out (slow end)
function easeOut(t: number): number {
  return 1 - (1 - t) * (1 - t);
}

// Ease in-out
function easeInOut(t: number): number {
  return t < 0.5
    ? 2 * t * t
    : 1 - Math.pow(-2 * t + 2, 2) / 2;
}

// Usage
step(): void {
  this.progress += 0.01;
  const easedProgress = easeInOut(this.progress);
  this.x = lerp(this.startX, this.endX, easedProgress);
}
```

Screen Shake

```
step(): void {
  if (this.shakeAmount > 0) {
    // Random offset
    const offsetX = random_range(-this.shakeAmount, this.shakeAmount);
    const offsetY = random_range(-this.shakeAmount, this.shakeAmount);

    // Apply to camera
    view_xview[0] += offsetX;
    view_yview[0] += offsetY;

    // Decrease shake
    this.shakeAmount *= 0.9;
    if (this.shakeAmount < 0.1) {
      this.shakeAmount = 0;
    }
  }
}
```

Percentage Calculations

```
// Health percentage
const healthPercent = (this.health / this.maxHealth) * 100;

// Progress bar (0.0 to 1.0)
const progress = this.currentValue / this.maxValue;

// Scale by percentage
const damageReduction = damage * 0.75; // 75% of damage

// Increase by percentage
this.damage *= 1.25; // +25% damage
```

Grid Snapping

```
// Snap to grid
function snapToGrid(value: number, gridSize: number): number {
  return Math.round(value / gridSize) * gridSize;
}

// Usage
const gridX = snapToGrid(this.x, 32);
const gridY = snapToGrid(this.y, 32);
```

Angle Difference

```
// Get shortest angle difference
function angleDiff(angle1: number, angle2: number): number {
  let diff = angle2 - angle1;
  while (diff > 180) diff -= 360;
  while (diff < -180) diff += 360;
  return diff;
}

// Usage
const targetAngle = point_direction(this.x, this.y, mouse_x, mouse_y);
const diff = angleDiff(this.image_angle, targetAngle);

// Turn gradually
this.image_angle += Math.sign(diff) * Math.min(Math.abs(diff), 5);
```

Map Range

```
// Map value from one range to another
function mapRange(
  value: number,
  inMin: number,
  inMax: number,
  outMin: number,
  outMax: number
): number {
  return (value - inMin) * (outMax - outMin) / (inMax - inMin) + outMin;
}

// Usage
// Map health (0-100) to alpha (0.3-1.0)
this.image_alpha = mapRange(this.health, 0, 100, 0.3, 1.0);

// Map distance to speed
const dist = point_distance(this.x, this.y, target.x, target.y);
this.speed = mapRange(dist, 0, 300, 0, 5);
```

Probability

```
// 30% chance
if (random(1) < 0.3) {
  // Rare drop
}

// Weighted random choice
function weightedChoice<T>(items: T[], weights: number[]): T {
  const total = weights.reduce((sum, w) => sum + w, 0);
  let rand = random(total);

  for (let i = 0; i < items.length; i++) {
    if (rand < weights[i]) {
      return items[i];
    }
    rand -= weights[i];
  }

  return items[items.length - 1];
}

// Usage
const loot = weightedChoice(
  ['common', 'rare', 'epic'],
  [70, 25, 5] // 70%, 25%, 5%
);
```

Constants

```
// JavaScript Math constants
Math.PI      // 3.14159...
Math.E       // 2.71828...
Math.SQRT2   // 1.41421...

// Usage
const circumference = 2 * Math.PI * this.radius;
const area = Math.PI * this.radius * this.radius;
```

Next Steps

- **25-api-motion.md** - Motion functions
- **24-api-input.md** - Input functions
- **40-common-patterns.md** - Math patterns

← Back to Index