수학을 활용하기

저희가 예시로 제작했던 닷찌! 게임에서, 적들은 그저 상하좌우에서 나와 위아래 양 옆으로만 움직일 뿐, 별다른 특별한 처리는 되지 않았습니다. 물론 그럼에도 상당히 어려웠지만 좀 더 적 캐릭터들의 인공지능을 향상시켜서 게임의 난이도를 좀 더 어렵게 할 수는 없을까요?

이번 장에서는 간단한 수학을 활용하여 게임에서 적 캐릭터들이 플레이어를 따라가게 해보고자 합니다. 또한 적들이 미사일을 날릴 때 플레이어의 방향으로 유도탄을 날리는 알고리즘에 대해서도 알아보고자 합니다. 또한 이 문제들은 해결하기 위한 수학 내용들에 대해서도 공부해 봅시다.

추적

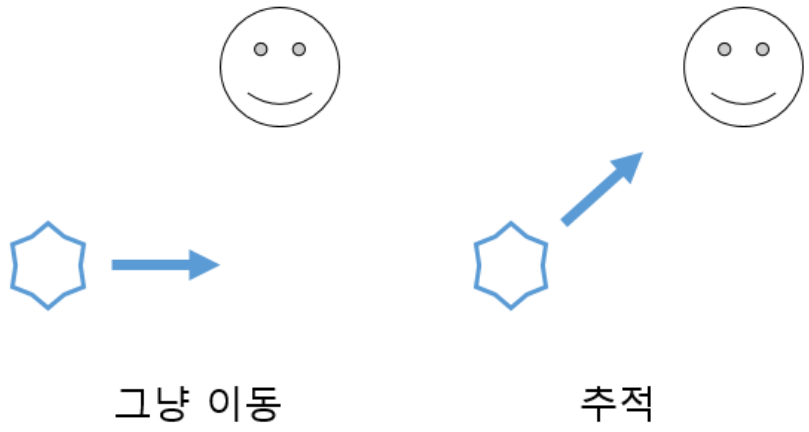
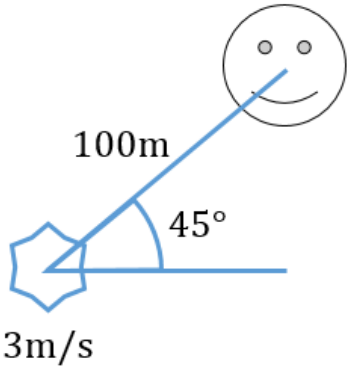


그림 1. 추적

추적은 위 그림과 같이, 목표물을 따라 이동하는 것을 말합니다. 어떻게 하면 목표물을 향해서 다가가는 속도를 가질 수 있을까요? 우선은 삼각함수를 이용하는 방법에 대해서 알려드리겠습니다.

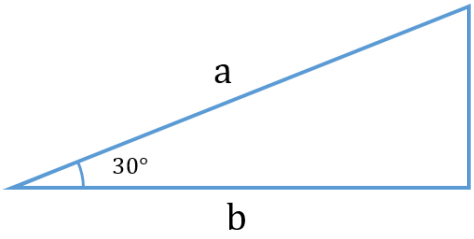
목표물의 위치에서 자신의 위치를 빼면, 목표물과의 변위가 나옵니다. 그러나 문제는, 따라가는 속도가 한정되어 있다는 것이죠, 예를 들어 2차원 상에서 목표와의 거리가 100m이고 각도가 45도이고 내 속력은 초속 3m라면, 1초에 X, Y축으로 얼마나 이동해야 할까요? 고등학교 때 배운 삼각함수를 사용하면, 속도는 X, Y축 모두 $\frac{3\sqrt{2}}{2}$m/s 라는 결과가 나옵니다.



$$\text{속도} = (3\cos 45°, 3\sin 45°)\text{m/s}$$
$$= \left(\frac{3\sqrt{2}}{2}, \frac{3\sqrt{2}}{2}\right)\text{m/s}$$

그림 2. 삼각함수를 이용한 속도 계산

만약 삼각함수에 대해서 잘 모르시는 분들을 위해서 설명을 드리자면, 삼각함수란 직각삼각형의 특정 각에서 세 변의 길이의 비율을 나타낸 함수입니다. 예를 들어 직각삼각형의 한 각이 45도일 때, 세 변의 관계는 아래 그림과 같습니다. 이것을 삼각비라고 합니다.



$$\frac{b}{a} = \frac{\sqrt{3}}{2} = \cos 30°$$
$$\frac{c}{a} = \frac{1}{2} = \sin 30°$$

그림 3. 삼각형의 삼각비

삼각함수란, 이 삼각비를 평면좌표계에서 일반화시킨 것을 의미합니다. 예를 들어 위 그림에서 각이 90도를 넘어간 것과 같은 경우에도 값이 나와 우리가 앞서 말한 상황에도 사용 가능합니다.

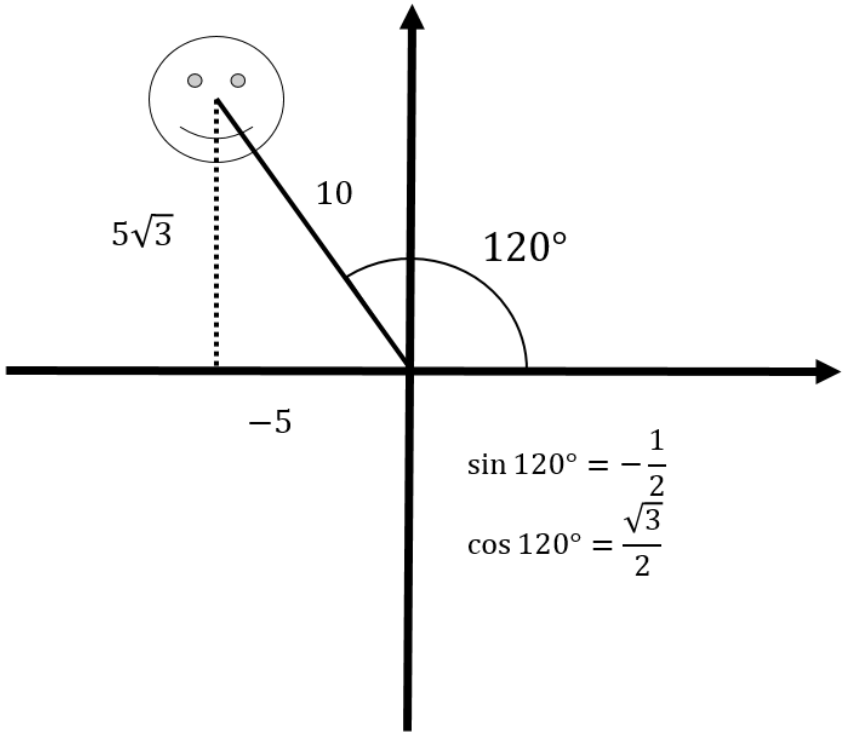


그림 4. 삼각함수

다시 문제로 돌아가서, 일반 데카르트 좌표계인 스크린 좌표계에선 그저 X, Y두 축에 대응하는 위치이기 때문에 단순히 각을 계산할 수는 없습니다. 다행히도, 대부분의 언어들은 특정 점이 원점과 이루는 각을 계산해주는 함수를 제공합니다. 이 함수는 바로 아크탄젠트(arctangent)입니다. 줄여서 atan이라고 표현하는 이 함수는 math.h안에 선언되어 있습니다. 하지만 단순히 위와 같은 목적을 가질 때에는 개량된 atan2함수를 사용해야 합니다.

```
double atan2(double y, double x)
```

이 함수가 반환하는 값은 인자로 온 점 (x, y)가 원점과 이루는 각을 호도로서 (−π, π) 사이의 값을 반환합니다. 이제 속도를 계산하는 함수를 만들 수 있습니다.

```
// (x1, y1)에서 (y1, y2)로 최대속력 speed로
// 가는 속도 (v1, v2)를 계산하는 함수.
void ComputeVelocity (
       float x1, float y1,
       float x2, float y2,
       float speed,
```

```
	float& v1, float& v2
	)
{
	float dx = x2 - x1,
		dy = y2 - y1;
	// 각을 계산하고.
	float angle = atan2f(dy, dx);

	// 거리를 계산해서
	float distance = sqrtf(dx * dx + dy * dy);

	// 거리보다 속력이 크면 속력을 줄입니다.
	if(distance < speed)
		speed = distance;

	// 삼각함수를 통해 속도를 계산합니다.
	v1 = speed * cos(angle);
	v2 = speed * sin(angle);
}
```

예제 1. 삼각함수로 속도 계산하기 / ComputeVelocity 함수

앞서 사용된 sin, cos, atan2 등의 삼각함수들은 복잡한 연산 과정을 거치기 때문에 연산에 시간이 많이 소요됩니다. 또 다른 복잡한 연산에도 이렇게 각을 계산하고 삼각함수를 사용한다면 매우 식이 복잡해질 것입니다. 따라서 이번에는 벡터에 대해서 알아보고, 벡터를 통해서 이 문제를 해결해 봅시다. 앞으로 여러분이 해결해야 할 수학적 문제들은 이제 벡터가 필수입니다

위 함수로 얻은 속도를 그대로 적 캐릭터에 적용하면, 부자연스럽게 사용자 캐릭터를 졸졸졸 따라다닐 것입니다. 원래 방향을 급선회하면, 이전의 속도로 인해서 천천히 속도가 줄어들었다가 방향을 바꿔가면서 다시 다른 방향으로 이동하는 것이 자연스러운데 말이죠. 이러한 처리 역시 기존의 방식으로는 많이 복잡할 것입니다.

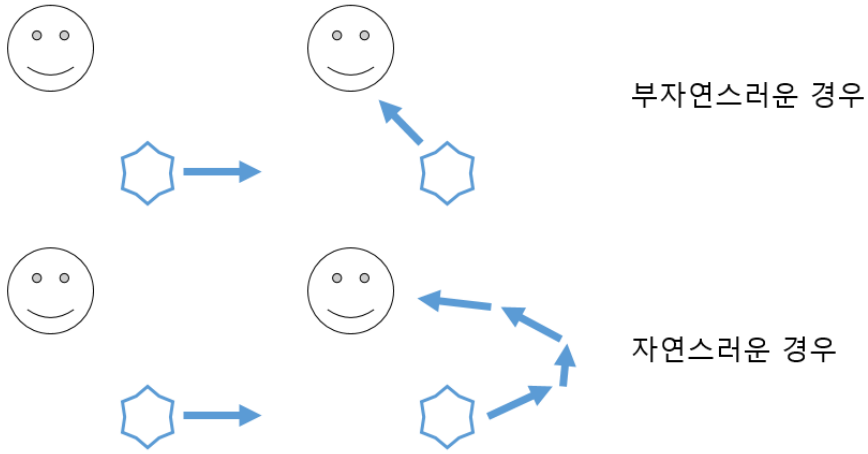


그림 5. 좀 더 자연스러운 추적

벡터(Vector)

그러면 잠시 벡터에 대해서 알아보겠습니다. 벡터(Vector)란, 크기와 방향만을 가진 것을 말합니다. 벡터는 n개의 성분(Component)들로 이루어져 있고, n개의 성분들로 이루어진 벡터를 n차원 벡터라고 합니다. 벡터와는 다르게 크기만을 가진 것을 스칼라(Scalar)라고 합니다. "동북쪽으로 시속 10km" 는 벡터이다. 왜냐하면 동북쪽이라는 방향과 시속 10km라는 크기를 가지고 있습니다. 그러나 100kg은 스칼라입니다. 이는 단순히 100kg이라는 질량의 크기만을 가질 뿐이기 때문이죠. 제가 앞서서 속력과 속도를 구분해서 말했습니다. 속력(Speed)는 빠르기의 양을 나타내는 스칼라이고, 속도(Velocity)는 빠르기와 방향을 가지는 벡터입니다. 또한 거리(Distance)가 두 위치의 먼 정도를 나타내는 반면, 변위(Displacement)는 거리와 방향 역시 가지고 있습니다. 따라서 변위 역시 벡터입니다.

n차원 벡터 v^{n}의 성분은 $\mathrm{v}_{\mathrm{x}}, v_y, \mathrm{v}_{\mathrm{z}}$, ...(혹은 x, y, z, ...)으로 나타냅니다. 그러나 n차원 벡터를 나타낼 때에는 $\mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3, \dots, \mathrm{v}_{\mathrm{n}}$과 같은 방식으로 나타냅니다. 또 아래처럼 행렬로 나타낼 수도 있습니다.

$$\mathrm{v}^{\mathrm{n}} = \begin{bmatrix} \mathrm{x} \\ y \\ z \\ \dots \end{bmatrix} = (x, y, z, \dots) = (v_x, v_y, v_z, \dots) = (v_1, v_2, v_3, \dots, v_n)$$

n차원 벡터의 각 성분을 n차원 직교 좌표계[1]의 한 점으로 향하는 화살표로 나타낼 수 있습니다. 아래의 그림은 2차원 공간에서의 한 예입니다.

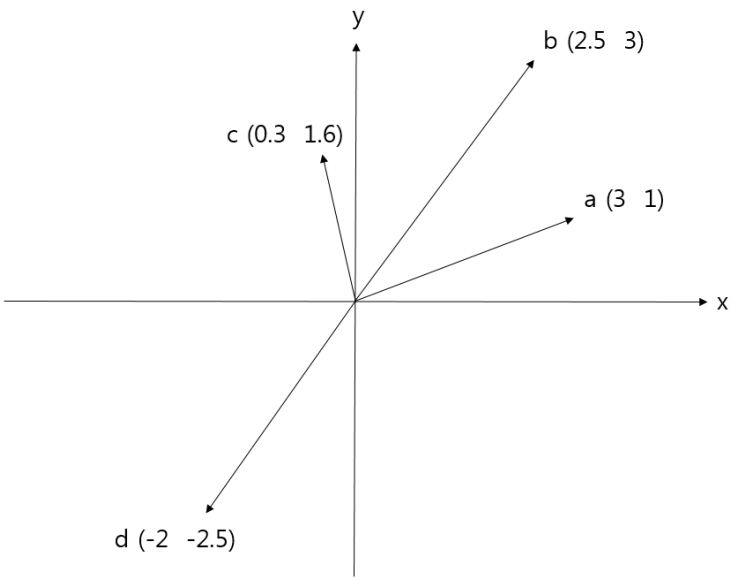


그림 6. 평면좌표계 안에서의 벡터

벡터는 무한한 차원으로 확장할 수 있지만 여기서는 간편하게 2차원 벡터만을 다룰 것입니다. 벡터에는 다양한 연산과 성질들이 존재합니다.

1. 벡터의 덧셈, 뺄셈

메모 포함[HK1]: 벡터의 덧셈, 뺄셈을 그림으로 보여주어야 될 듯

두 벡터 u, v 가 있을 때,

$$u + v = v + u = (u_x + v_x,\ u_y + v_y)$$

$$u - v = (u_x - v_x,\ u_y - v_y)$$

벡터의 뺄셈은 교환법칙이 성립하지 않습니다.

[1] 직교 좌표계란 수직인 축들로 위치를 나타내는 좌표계로서, 데카르트 좌표계도 직교 좌표계의 일부입니다.

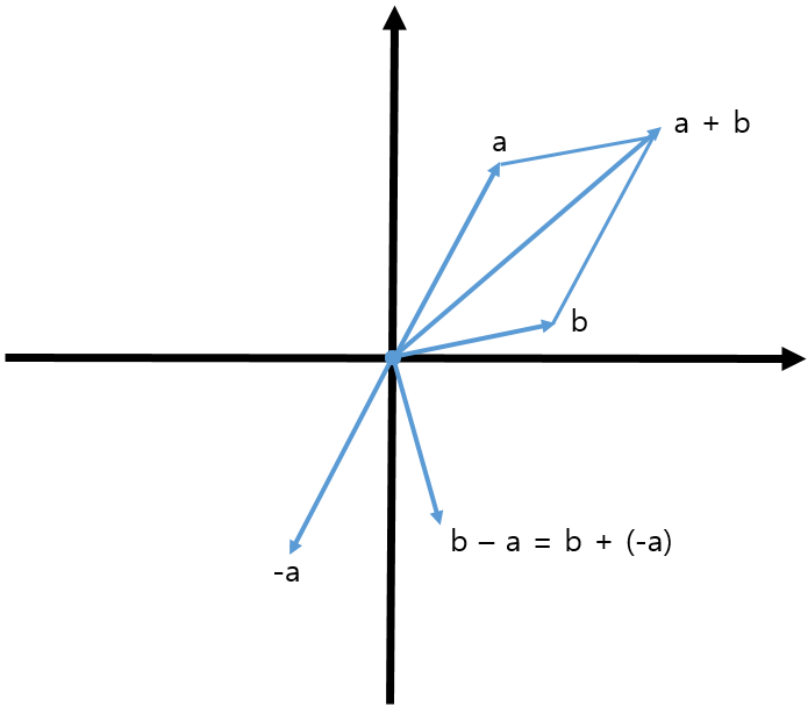


그림 7. 벡터의 덧셈과 뺄셈

위 그림에서 보듯, 벡터의 덧셈의 결과는 원점과 두 벡터가 이루는 평행사변형의 끝 점입니다.

2. 벡터 u의 덧셈에 대한 역수는

$$-u = (-u_x, -u_y)$$

3. 벡터와 스칼라의 곱셈

벡터 v, 스칼라 a가 있을 때,

$$av = va = (av_x, av_y)$$

4. 벡터의 크기(길이, norm 혹은 length)

벡터 v가 있을 때, 직교 좌표계에서 벡터의 크기는 벡터가 가리킨 위치와 원점의

길이를 말합니다. 이는 피타고라스의 법칙을 이용해서 쉽게 길이를 구할 수 있습니다. 벡터의 v의 크기는 $\|v\|$로 표시합니다.

$$\|v\| = \sqrt{v_x^2 + v_y^2}$$

길이가 1인 벡터를 단위 벡터(Unit Vector)라고 합니다. 벡터를 단위 벡터로 만드는 방법을 정규화(Normalize)라고 합니다. 벡터 v에 스칼라 a를 곱해서 그 길이가 1이 되면 됩니다. 그렇다면 벡터에 곱할 스칼라 a를 한번 계산해 봅시다.

$$\begin{aligned}
1 &= \|av\| \\
&= \sqrt{(av_x)^2 + \left(av_y\right)^2} \\
1^2 &= a^2v_x^2 + a^2v_y^2 \\
1 &= a^2\left(v_x^2 + v_y^2\right) \\
a^2 &= \frac{1}{v_x^2 + v_y^2} \\
a &= \frac{1}{\sqrt{v_x^2 + v_y^2}} = \frac{1}{\|v\|}
\end{aligned}$$

즉 a는 벡터의 길이의 역수입니다. 이렇게 구한 a를 벡터 v에 곱함으로써 단위 벡터 $\hat{v}$을 만들 수 있습니다.

$$\begin{aligned}
\hat{v} &= av \\
&= \frac{1}{\sqrt{v_x^2 + v_y^2}} \cdot v
\end{aligned}$$

$$= \frac{v}{\|v\|}$$

이제 벡터에 대해서 알았으니, 속도와 변위에 대해서 벡터로써 생각할 수 있을 것입니다. 이제 위에서 각을 써서 해결했던 추적 문제를 손쉽게 해결할 수 있습니다. 목표물의 방향으로 가는 길이가 3인 속도 벡터를 계산하면 되는 것입니다. 먼저 뺄셈으로 변위 벡터를 계산하고 그것을 정규화하면 길이가 1이고 목표물을 향한 벡터가 나오기 때문에 이 벡터에 속력을 곱해주기만 하면 됩니다. 즉 아래 그림과 같습니다.

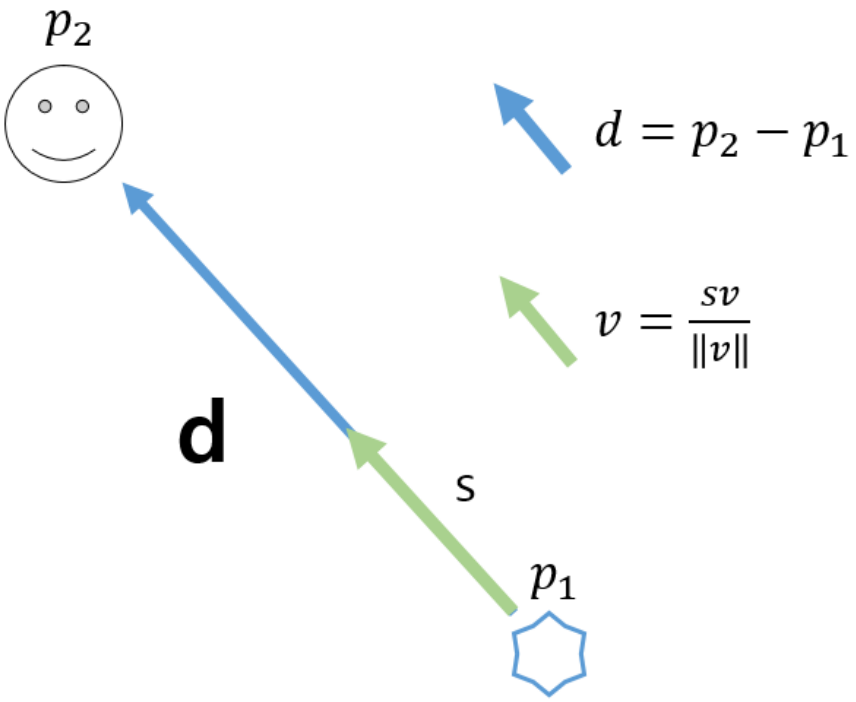


그림 8. 벡터를 통한 추적

이를 구현한 함수는 아래와 같습니다.

```
// 2차원 벡터를 나타내기 위한 Vector2D 구조체
struct Vector2D {
	float x, y;
```

```
	inline Vector2D(float x = 0, float y = 0)
		: x(x), y(y)
	{}

	// 벡터의 크기를 구하는 함수
	inline float length() const {
		return sqrtf(x * x + y * y);
	}
};

// p1에서 p2로 speed속력으로 가는 속도 vel을 계산합니다.
void ComputeVelocity (
	Vector2D p1,
	Vector2D p2,
	float speed,
	Vector2D& vel
	)
{
	// 변위(d) 계산
	Vector2D d (
		p2.x - p1.x,
		p2.y - p1.y
		);

	// 거리를 계산(변위 벡터의 길이 = 거리)
	float distance = d.length();

	// 거리가 속도보다 크지 않게
	if(distance < speed)
		speed = distance;

	// 정규화(원소를 길이로 나눔)하고
	// 속도를 곱하면 끝
	vel.x = d.x / distance * speed;
	vel.y = d.y / distance * speed;
}
```

예제 2 벡터로 속도 계산하기 / ComputeVelocity 함수

예제 1과 예제 2의 두 함수를 통해서 p1 = (100, 100), p2 = (150, -175), speed = 3으로 했을 때 속도는 둘 다 (6.26, -34.44)가 나왔습니다.

벡터의 내적

벡터에는 내적Dot Product이라는 연산도 존재합니다. 이번 장에서는 사용되지 않지만 정말로 쉽고 유용한 기능이므로 알아두시기 바랍니다. 이후 행렬에 대해서 이야기할 때 다시 나옵니다.

벡터 u, v가 있을 때,

$$u \cdot v = v \cdot u = u_x v_x + u_y v_y$$
$$= \|u\|\|v\| \cos\theta$$

두 번째 줄의 내용이 중요한데, 저기서 θ는 두 벡터의 사잇각을 의미합니다.

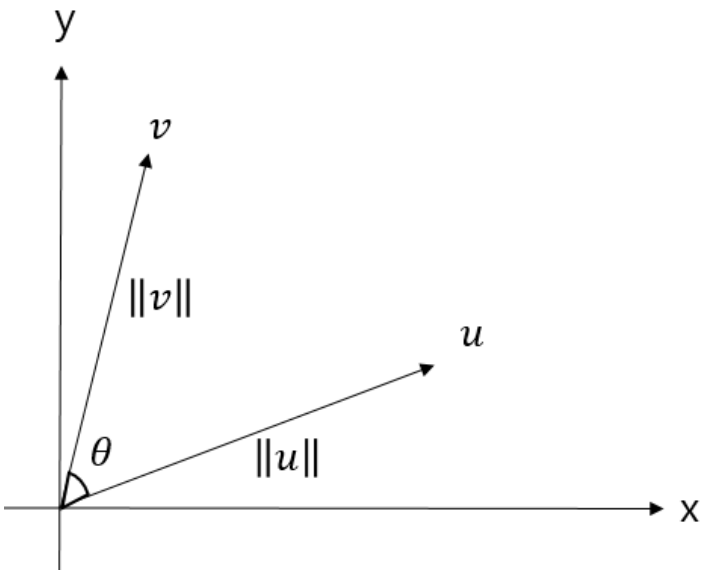


그림 9. 두 벡터와 그 사잇각

위 식을 조금 변형하면, 두 벡터의 사잇각의 코사인을 얻을 수 있습니다.

$$\frac{u \cdot v}{\|u\|\|v\|} = \cos\theta$$

코사인 함수의 아래의 성질을 이용하면, 두 벡터 사이의 관계를 알 수 있습니다.

$$\cos 0° = 1$$

$$\cos 90° = 0$$

$$\cos 180° = -1$$

만약 두 벡터의 내적이 0이라면, 두 벡터는 직교(수직)한다.

자연스러운 추적

이제 앞서 말한 자연스러운 추적을 해 봅시다. 속도를 변화시키는 것은 가속도(Acceleration)입니다. 가속도의 기본적인 단위는 m/s^2 입니다. 가속력 $15m/s^2$은 1초동안 속력을 $15m/s$ 올릴 수 있음을 의미합니다. 가속도 역시 벡터입니다. 또한 속도와 속력이 다르듯 가속력 역시 스칼라양이고 가속도의 정도를 나타냅니다. 그런데 가속도는 물체마다 다를 수 있습니다. 예를 들어 코끼리를 멈추는 것 보다는 축구공을 멈추는 것이 쉽습니다. 이것은 힘(Force)과 관련된 것이나, 그것까지 알지 않아도 우리가 미리 가속력을 설정해 둔다면 원하는 방향으로의 가속도를 계산할 수 있을 것입니다.

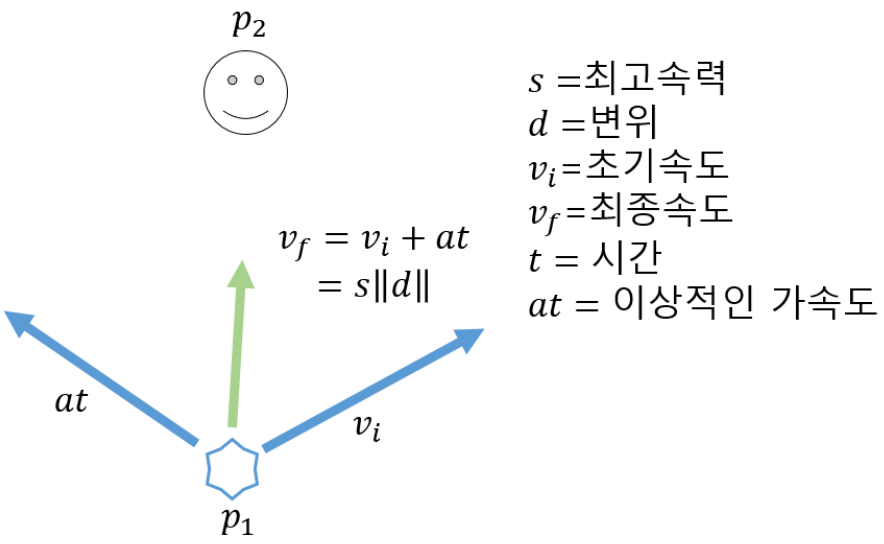


그림 10. 가속도

위 그림에서, v_f 는 이전에 구한 함수로 쉽게 계산할 수 있었습니다. 가속도 at 역시 $v_f - v_i$ 로 계산할 수 있습니다. 하지만 가속도에도 한계가 있습니다. 자전거가 시속 80km로 달리다가 갑자기 방향을 90도 선회할 수는 없습니다. 즉 at 의 크기가 일정 제한을 넘어서지 못하게 해야 합니

다. 즉 최대가속력 a′이 있어야 합니다. 벡터의 길이가 일정 수보다 클 경우, 벡터의 길이를 줄여내야 하는데 이를 잘림(Truncation)이라고 합니다. 이것은 매우 간단합니다. 그냥 길이가 큰 지 검사하고 단위벡터로 만든 뒤 원하는 크기를 곱해주기만 하면 됩니다. 즉 아래 식과 같습니다.

$$\text{최종가속도} = \mathrm{a_f} = \mathrm{a'} \cdot \|at\|$$

이를 함수로 만들어보면 아래와 같습니다.

```
void ComputeAcceleration (
       Vector2D p1, // 현 위치
       Vector2D p2, // 목표 위치
       float speed, // 속도
       float maxA,          // 최대 가속력
       Vector2D vel,// 현재 속도
       Vector2D& acc// 결과 가속도
       )
{
       Vector2D velf;

       // 목표까지의 속도 계산
       ComputeVelocity(p1, p2, speed, velf);

       // 이상적인 가속도 계산
       acc.x = velf.x - vel.x;
       acc.y = velf.y - vel.y;

       // 길이가 크면 벡터를 자릅니다.
       float accLen = acc.length();

       if(accLen > maxA)
       {
              acc.x = acc.x / accLen * maxA;
              acc.y = acc.y / accLen * maxA;
       }
}
```

예제 3. 추적가속도계산

아래 그림은 시간의 흐름에 따른 속도와 위치의 변화를 나타낸 그림입니다.

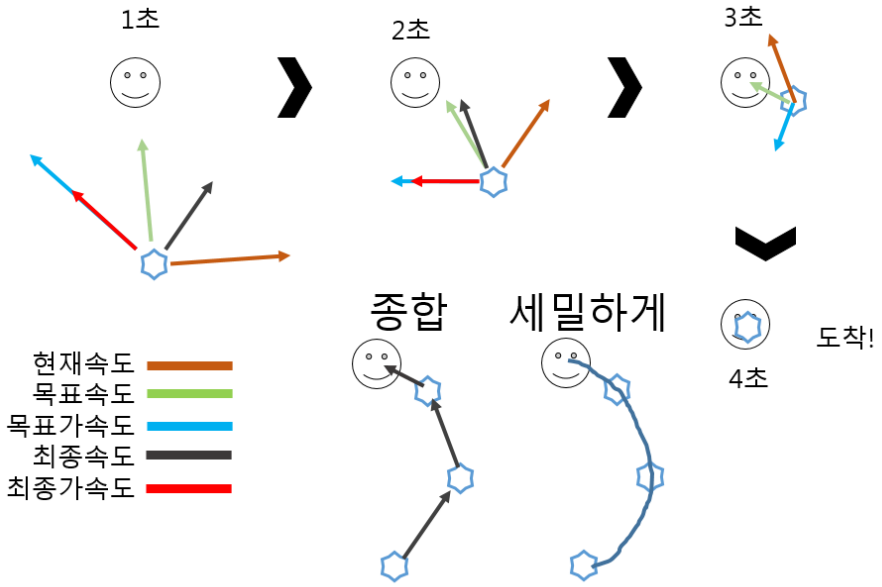


그림 11. 시간의 흐름에 따른 속도와 위치의 변화

그림을 설명하면, 현재 속도(갈색)에서 목표물 ☺ 에 도달하기 위해선 목표 속도(초록색)으로 속도가 바뀌어야 합니다. 따라서 목표 속도에 목표 가속도(하늘색)를 더해주면 목표 속도가 나옵니다. 그러나 가속도에도 한계가 있어 그 한계에 맞춘 가속도가 최종 가속도(빨간색)입니다. 결국 최종 가속도에 시간을 곱해서 현재 속도에 더한 것이 최종 속도(검정색)이 되어 그 방향으로 나아가게 됩니다.

3초대에서는 목표가속도가 한계가속력보다 작았기 때문에 그냥 목표속도로 이동하여 버리면 됩니다. 이 것을 좀 더 세밀하게 볼 경우 부드럽게 이동 방향이 바뀌며 목표물로 돌진! 하게 됩니다. 하지만 대상은 보통 가만히 있지 않습니다. 그래서 대상이 움직일 위치를 예상해서 그 지점으로 이동한다면 좀 더 발전한 AI가 될 수 있습니다. 역으로 이것을 대상이 이용할 수도 있겠지만요.

그러한 알고리즘은 쉽게 또는 간단하게 구현할 수 있습니다. 쉽게 구현한다면 그저 대상이 1초 뒤에 위치할 곳의 방향으로 이동할 수 있겠고, 어렵게 구현한다면 적의 속도와 내 속력을 비교해서 충돌 가능한 지점을 찾아내는 것이겠죠. 예를 들면 미사일을 요격하는 것과 비슷할 것입니다. 관련 구현은 독자 여러분들께서 해 보시기 바랍니다.