Linear algebra for videogames

MATHEMATICS IMPLEMENTATION IN C++

Vito Domenico Tagliente
[NOME DELLA SOCIETÀ] | [INDIRIZZO DELLA SOCIETÀ]

Linear Algebra for videogames

VITO DOMENICO TAGLIENTE

Prefazione

Questo libro presenta una spiegazione degli elementi fondamentali dell'algebra lineare tramite un approccio orientato alla programmazione. Infatti, i vari concetti teorici saranno spiegati attraverso il formalismo matematico, arricchito di implementazione in codice. L'implementazione dei vari concetti matematici, quali vettori, matrici, consente di definire un approccio allo studio più concreto, permettendo il lettore di capire meglio gli argomenti trattati, permettendogli di svolgere diversi esercizi utilizzando le strutture dati che verranno definite attraverso i vari capitoli.

Lo scopo di questo libro è quello fornire una infarinatura generale sull'algebra lineare, senza entrare troppo nel dettaglio con informazioni che possono essere approfondite in manuali appositi. In particolare, verranno esaminati gli aspetti più importanti, strettamente correlati al contesto dei videogiochi. Le conoscenze acquisite, a temine volume, permetteranno il lettore di avere più consapevolezza sui concetti generali proposti da diversi motori grafici (Unity o Unreal Engine, per citarne alcuni).

Note sull'autore

Il mio nome è Vito Domenico Tagliente. Mi sono laureato con lode alla facoltà di Ingegneria Informatica del politecnico di Bari, Italia. TODO.

Sommario

Vettori

In natura esistono grandezze determinate dal numero che le misura rispetto a una prefissata unità, come per esempio la lunghezza, l'area, il volume, il tempo. Queste grandezze sono dette scalari. Altre grandezze, come per esempio lo spostamento e la velocità, sono rappresentate da un numero, una direzione e un verso. Tali grandezze vengono chiamate grandezze vettoriali e vengono descritte mediante vettori.

Ad esempio, se vogliamo descrivere il movimento di una particella, dire che questa si muove ad una certa velocità non è sufficiente. Affinché la descrizione sia completa, occorre specificare anche dove questa si sta muovendo, in particolare in quale direzione e con quale verso questa si sta spostando.

Cenni storici

La notazione matriciale venne introdotta principalmente per esprimere le relazioni dell'algebra lineare in forma compatta, soprattutto per quanto riguarda la visibilità. A prova di ciò, consideriamo un insieme di relazioni lineari definite tra un insieme $X = \{x_1, x_2, \dots, x_n\}$ di n elementi e un insieme $Y = \{y_1, y_2, \dots, y_m\}$ di m elementi.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = y_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = y_2$$

$$\dots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = y_m$$

Il contenuto informativo della relazione può essere formalmente rappresentato dalla seguente notazione:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{bmatrix}$$

Dalla notazione proposta, due oggetti matematici possono essere individuati:

- Il vettore a due dimensioni **A**, detto anche matrice
- I vettori a singola dimensione, detti vettori colonna o semplicemente vettori, quali **x** e **y**.

Applicando la nuova notazione, si ricava che la nostra relazione può essere espressa in forma matriciale $\boldsymbol{Ax} = \boldsymbol{y}$.

Notazione vettoriale

Definiamo vettore, un insieme di n elementi che può essere espresso come:

$$\begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix}$$

Questo oggetto è chiamato vettore colonna. In seguito, sarà chiaro il perché un vettore possa essere considerato come un caso speciale di matrice. Una matrice avente n righe ed una sola colonna.

Un vettore può anche essere espresso secondo la notazione $\boldsymbol{x} = [x_1\ x_2 \dots x_n]$, tale rappresentazione prende il nome di vettore riga.

In generale si parla semplicemente di vettori. Se non viene specificato alcun qualificatore, si sottintende si tratti, per convenzione, di un vettore colonna.

I vettori sono rappresentati nei testi di algebra con la notazione $\vec{v}$, in realtà per questioni di comodità relative alla videoscrittura su computer, i vettori vengono anche spesso identificati con una lettera minuscola in grassetto **x**.

Gli elementi che costituiscono un vettore sono chiamati componenti, dove il simbolo n viene adoperato per indicare la quantità di componenti, l'ordine del vettore. Per esempio, definito il vettore $\boldsymbol{v} = [2 \quad 0]$, la prima componente è 2, la seconda è 0. L'ordine di **v** è 2, in quanto $n = 2$.

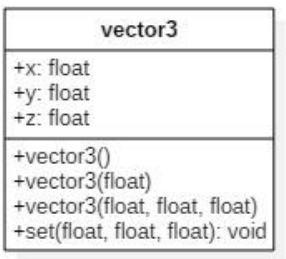
Implementazione in C++

Spesso risulta molto complesso modellare concetti reali in codice, specialmente quando si è alle prime armi con la programmazione. In particolare, i concetti matematici sembrano presentare qualche difficoltà in più, sarà per la complessità degli argomenti o per il grado di astrazione che quest'operazione richiede.

La caratteristica chiave del C++ è il concetto di classe, inteso come tipo di dato definito dall'utente atto a rappresentare un concetto più o meno complesso nel codice di un programma. L'utilizzo di classi permette non solo di modellare meglio le diverse entità che caratterizzano un programma, ma anche di organizzare meglio il codice, migliorando notevolmente la leggibilità dello stesso. Pertanto, possiamo considerare una classe come un contenitore logico di variabili, che forniscono la descrizione dell'oggetto, e di funzioni, che definiscono le modalità con cui quest'ultimo interagisce con il "mondo".

Consideriamo di creare il modello una persona. È possibile individuare diverse caratteristiche importanti ai fini della descrizione dell'entità stessa. Per esempio, si può decidere di caratterizzarla in base al nome, cognome, altezza, peso. Diventa fondamentale il contesto nel quale il modello viene sviluppato, un modello sarà dettagliato tanto quanto richiesto dal caso in esame. Ci possono essere casistiche in cui è necessario specificare il colore degli occhi, altre in cui non lo è, per esempio. Da ciò si ricava che un modello è anche frutto della percezione e del bagaglio di conoscenza personale.

Modelliamo il concetto di vettore. Per semplificare le cose, consideriamo l'implementazione di un vettore di ordine 3, ovvero caratterizzato da tre componenti.



Individuate le caratteristiche descrittive del concetto considerato, dette anche attributi, occorre definire le funzionalità, detti metodi, che permetteranno al contesto (il programma) di interagire con l'oggetto così definito.

È buona norma prendersi del tempo per pensare e modellare bene i concetti. Una volta identificate le caratteristiche che compongono e descrivono un oggetto, è possibile procedere con la stesura del codice.

```
class vector3
{
public:
    float x, y, z;

    vector3() {
        x = y = z = 0.0f;
    }

    vector3(float value) {
        x = y = z = value;
    }

    vector3(float _x, float _y, float _z) {
        x = _x;
        y = _y;
        z = _z;
    }

    vector3(const vector3& other) {
        x = other.x;
        y = other.y;
        z = other.z;
    }

    ~vector3() {}

    vector3& set(float _x, float _y, float _z) {
        x = _x;
        y = _y;
        z = _z;
        return (*this);
    }
}
```

Osservando il codice riportato, è possibile evidenziare diversi aspetti importanti:

- In C++, come in qualsiasi linguaggio di programmazione orientato agli oggetti, è possibile definire il livello di visibilità delle informazioni. Durante la definizione di una classe occorre specificare quali attributi e metodi apparterranno ad un contesto di accessibilità pubblico e quali privato. Le informazioni pubbliche sono visibili all'esterno, ogni entità del programma che ha una relazione stretta con l'oggetto designato è in grado di interagirvi attraverso l'interfaccia pubblica. Le informazioni private, al contrario, hanno ambito di visibilità strettamente correlato al contesto di esistenza dell'oggetto stesso.
- È buona regola utilizzare il costruttore per definire le logiche di inizializzazione delle informazioni di un oggetto.
- Le informazioni di maggior rilievo vengono spesso assegnate ad un ambito di visibilità locale (privato). Questo permette di controllare e vigilare gli accessi e le modifiche di queste ultime, preservando la consistenza del contenuto informativo di tutto l'oggetto.

La classe vector3 possiede tre attributi o componenti chiamate rispettivamente x, y e z. Tali componenti definiscono la posizione dell'oggetto nello spazio, in questo caso in due dimensioni.

Si è deciso di quantificare le componenti come float, ovvero numeri a virgola mobile. Tale scelta si adatta perfettamente all'applicazione di tale implementazione a differenti librerie grafiche di visualizzazione, le quali fanno ampio utilizzo di float.

La classe presenta diversi costruttori che permettono di inizializzare le componenti dell'oggetto. Occorre confrontare il formalismo matematico con quanto implementato.

Dal punto di vista del codice, il vettore così modellato non fa distinzione tra la rappresentazione come vettore riga o colonna. L'oggetto si occupa semplicemente di mantenere in memoria le informazioni sulle componenti.

Ricaviamo che la seguente notazione $\boldsymbol{v} = [2 \quad 0 \quad 1]$ si implementa in

```
vector3 v(2, 0, 1);
```

che risulta equivalente alla seguente forma:

$$\boldsymbol{v} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

Vettori Speciali

Esistono diversi vettori "speciali", nel senso che presentano una nomenclatura differente in dipendenza alla specifica valorizzazione delle componenti.

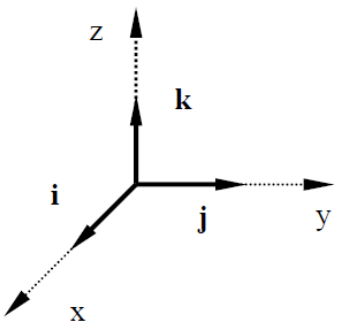
Si definisce vettore nullo, quel vettore le cui componenti sono tutte pari a zero, $\boldsymbol{z} = [0\ 0\ \ldots 0]$. A livello implementativo è possibile definire una funzione apposita allo scopo:

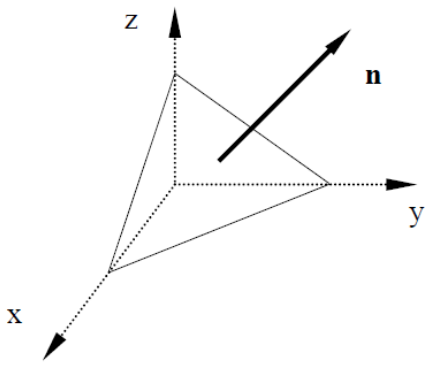
```
static vector3 zero() {
    return vector3(0.0f);
}
```

Pertanto, in questo modo si può definire un vettore nullo utilizzando la funzione statica specificata:

```
vector2 zero = vector2::zero();
```

Si definisce versore e si denota con $\boldsymbol{e}_i$, quel vettore le cui componenti sono tutte zero all'infuori della i-esima, pari a uno. I versori hanno modulo unitario e sono utilizzati per definire l'orientamento di una retta o di un vettore. I versori corrispondenti agli assi cartesiani spesso sono indicati come **i**,**j**,**k**. Infine, si denota con **n** il versore avente direzione ortogonale a un generico piano π.





A livello analitico, in uno spazio a due dimensioni si ricava:

- $\boldsymbol{i} = [1 \quad 0]$
- $\boldsymbol{j} = [0 \quad 1]$

```
vector2 i(1, 0);
vector2 j(0, 1);
```

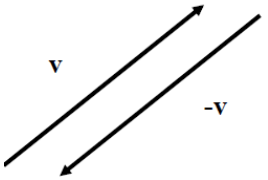
Invece, in uno spazio a tre dimensioni, come riportato in figura in alto

- $\boldsymbol{i} = [1 \quad 0 \quad 0]$
- $\boldsymbol{j} = [0 \quad 1 \quad 0]$
- $\boldsymbol{k} = [0 \quad 0 \quad 1]$

```
vector3 i(1, 0, 0);
vector3 j(0, 1, 0);
vector3 k(0, 0, 1);
```

Definito un vettore **v**, si chiama vettore opposto quel vettore **-v** che possiede stesso modulo, stessa direzione, ma verso contrario a **v**.

Per esempio, dato $\boldsymbol{v} = [2 \quad 1]$, l'opposto è $-\boldsymbol{v} = [-2 \quad -1]$.




```
vector2 operator-() const {
    return vector2(-x, -y);
}
```

È da notare che per raggiungere tale risultato, è stato sovrascritto l'operatore -, che ci permette di avere la stessa notazione matematica anche a livello di codice:

```
vector2 v(2, 1);
vector2 v1 = -v;
```

Operazioni con i vettori

Comportamento predominante del seguente manoscritto consiste nell'utilizzo dell'overloading degli operatori. Per overloading di operatori si intende la capacità di un linguaggio di programmazione di permettere di specificare quale logica comportamentale utilizzare nel momento in cui si applicano gli operatori matematici a tipi semplici e/o specificati dall'utente.

L'utilizzo di una notazione concisa per le operazioni di uso comune è di importanza fondamentale. In tutti i linguaggi, di programmazione e non, gli operatori sono dei simboli convenzionali che rendono più agevole la presentazione e lo sviluppo di concetti di uso frequente. Per esempio, la notazione $a + b * c$ risulta molto più agevole della frase "moltiplica b per c e aggiungi il risultato ad a".

Il C++ supporta un insieme di operazioni per i suoi tipi nativi. Tuttavia, la maggior parte dei concetti utilizzati comunemente non sono facilmente rappresentabili per mezzo di tipi nativi, e bisogna spesso ricorrere a tipi complessi. Per esempio, i numeri complessi, i vettori, le matrici, sono tutte entità che meglio si prestano a essere modellate e rappresentate mediante le classi. In questo caso risulta molto più agevole e coinciso ridefinire il significato delle operazioni nell'ambito di questi tipi definiti dall'utente, in alternativa alla chiamata di funzioni. Si ricava $a + b * c$ rispetto a $add(a, multiply(b, c))$

Per ottenere l'overload di un operatore in C++, occorre creare una funzione il cui nome deve essere costituito dalla parola chiave operator, seguita dal simbolo dell'operatore che si vuole ridefinire. Gli argomenti dell'operatore devono corrispondere agli operandi dello

stesso. Ne consegue che per gli operatori unari è necessario specificare un solo argomento, per quelli binari ne occorrono due.

Occorre sottolineare che non è possibile "inventare" nuovi simboli, ma si possono utilizzare solo quelli predefiniti. Inoltre, le regole di precedenza e associatività restano legate al simbolo e non al suo significato. Per esempio, un operatore in overload associato al simbolo *, non può mai essere definito unario e ha sempre la precedenza sull'operatore associato al simbolo +, qualunque sia il significato di entrambi.

Tornando all'esempio precedente in cui abbiamo definito la classe point, vediamo come è possibile definire l'operatore di somma fra due punti.

```
Point operator+(const Point& p1, const Point& p2){
    Point p(p1.GetX() + p2.GetX(), p1.GetY() + p2.GetY());
    return p;
}
```

Con la seguente definizione, diventa possibile definire l'espressione `Point p = p1 + p2;` anziché richiamare un ipotetico metodo `Point p = add(p1, p2);`

Notare:

- La funzione ha un valore di ritorno di tipo Point
- Gli argomenti sono passati per riferimento e dichiarati costanti, questa pratica è comune in C++ quando si vuole passare delle informazioni che non devono essere modificati dal corpo della funzione chiamata.

Finora abbiamo visto come ridefinire il comportamento di un operatore, ma non ci siamo occupati di specificare dove tali funzioni devono essere scritte. In particolare, quando queste devono accedere a informazioni private della classe, è opportuno ridefinire gli operatori tramite metodi pubblici della classe stessa, oppure attraverso delle funzioni friend.

In generale, quando il primo operando è oggetto della classe o la funzione lo restituisce come lvalue, La miglior progettazione degli operatori di una classe consiste nell'individuare un insieme ben definito di metodi per le operazioni che si applicano su un unico oggetto o che modificano il loro primo operando. Usare funzioni esterne (o friend) per le altre.

Nel primo caso, risulta importante utilizzare l'operatore this. Il this punta allo stesso oggetto della classe definita, in cui il metodo è incapsulato e viene automaticamente inserito dal C++. Si ricava che nel momento in cui una ridefinizione di operatore viene descritta come metodo della classe stessa, deve essere ridotto di una unità il numero di argomenti rispetto al numero di operandi, in quanto il primo operando è l'oggetto stesso. Si ricava che se l'operatore è binario, la funzione non deve avere argomenti, se l'operatore è binario, la funzione è caratterizzata da u solo argomento.

Se il risultato dell'operazione è l'oggetto stesso, l'istruzione di ritorno sarà caratterizzata dall'espressione `return *this`.

Vediamo come implementare l'overloading dell'operatore += come metodo della classe stessa:

```
class Point
{
    // ...
public:
```

```
    // ....

    Point& operator+=(const Point& p){
        x += p.GetX();
        y += p.GetY();
        return *this;
    }
}
```

Quanto visto finora si presenta come breve ripasso alla sintassi del C++. In seguito, partendo dall'implementazione dei primi concetti algebrici, risulterà più chiaro come classi e operatori possono essere utilizzati per implementare una notazione chiara e conforme con quella matematica anche a livello di linguaggi di programmazione.