

6 ноября 2011 в 15:29

Линейная алгебра для разработчиков игр перевод [tutorial](#)

Game Development*

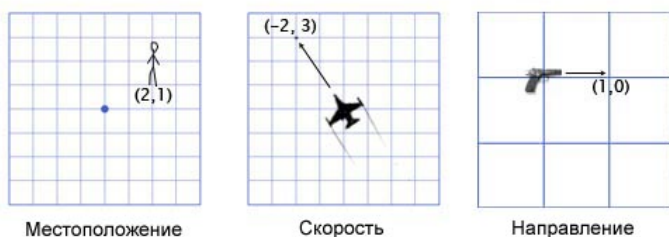
Эта статья является переводом цикла из четырёх статей «Linear algebra for game developers», написанных David Rosen и посвящённых линейной алгебре и её применению в разработке игр. С оригинальными статьями можно ознакомиться тут: [часть 1](#), [часть 2](#), [часть 3](#) и [часть 4](#). Я не стал публиковать переводы отдельными топиками, а объединил все статьи в одну. Думаю, что так будет удобнее воспринимать материал и работать с ним. Итак приступим.

Зачем нам линейная алгебра?

Одним из направлений в линейной алгебре является изучение векторов. Если в вашей игре применяется позиционирование экранных кнопок, работа с камерой и её направлением, скоростями объектов, то вам придётся иметь дело с векторами. Чем лучше вы понимаете линейную алгебру, тем больший контроль вы получаете над поведением векторов и, следовательно, над вашей игрой.

Что такое вектор?

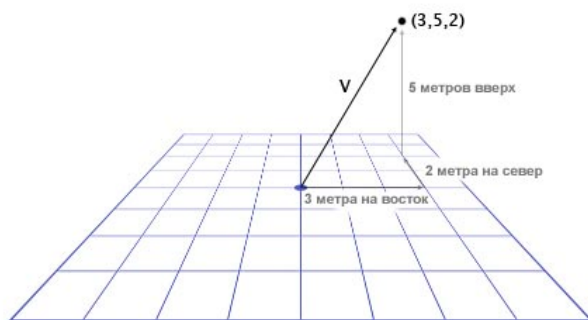
В играх вектора используются для хранения местоположений, направлений и скоростей. Ниже приведён пример двумерного вектора:



Вектор местоположения (также называемый «радиус-вектором») показывает, что человек стоит в двух метрах восточнее и в одном метре к северу от исходной точки. Вектор скорости показывает, что за единицу времени самолёт перемещается на три километра вверх и на два — влево. Вектор направления говорит нам о том, что пистолет направлен вправо.

Как вы можете заметить, вектор сам по себе всего лишь набор цифр, который обретает тот или иной смысл в зависимости от контекста. К примеру, вектор $(1, 0)$ может быть как направлением для оружия, как показано на картинке, так и координатами строения в одну милю к востоку от вашей текущей позиции. Или скоростью улитки, которая движется вправо со скоростью в 1 милю в час (*прим. переводчика: довольно быстро для улитки, 44 сантиметра в секунду*).

Важно отслеживать единицы измерения. Допустим у нас есть вектор $V(3,5,2)$. Это мало что говорит нам. Три чего, пять чего? В нашей игре *Overgrowth* расстояния указываются в метрах, а скорости в метрах в секунду. Первое число в этом векторе — это направление на восток, второе — направление вверх, третье — направление на север. Отрицательные числа обозначают противоположные направления, на запад, вниз и на юг. Местоположение, определяемое вектором $V(3,5,2)$, находится в трёх метрах к востоку, в пяти метрах вверх и в двух метрах к северу, как показано на картинке ниже.



Итак, мы изучили основы работы с векторами. Теперь узнаем как вектора использовать.

Сложение векторов

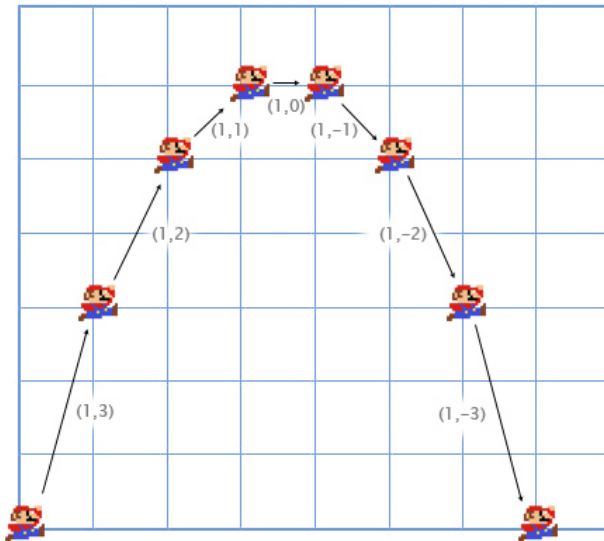
Чтобы сложить вектора, нам надо просто сложить каждую их составляющую друг с другом. Например:

$$(0, 1, 4) + (3, -2, 5) = (0+3, 1-2, 4+5) = (3, -1, 9)$$

Зачем нам нужно складывать вектора? Наиболее часто сложение векторов в играх применяется для физического интегрирования. Любой физический объект будет иметь вектора для местоположения, скорости и ускорения. Для каждого кадра (обычно это одна шестидесятая часть секунды), мы должны интегрировать два вектора: добавить скорость к

местоположению и ускорение к скорости.

Давайте рассмотрим пример с прыжками Марио. Он начинает с позиции $(0, 0)$. В момент начала прыжка его скорость $(1, 3)$, он быстро движется вверх и вправо. Его ускорение равно $(0, -1)$, так как гравитация тянет его вниз. На картинке показано, как выглядит его прыжок, разбитый на семь кадров. Чёрным текстом показана его скорость в каждом фрейме.



Давайте рассмотрим первые кадры поподробнее, чтобы понять как всё происходит.

Для первого кадра, мы добавляем скорость Марио $(1, 3)$ к его местоположению $(0, 0)$ и получаем его новые координаты $(1, 3)$. Затем мы складываем ускорение $(0, -1)$ с его скоростью $(1, 3)$ и получаем новое значение скорости Марио $(1, 2)$.

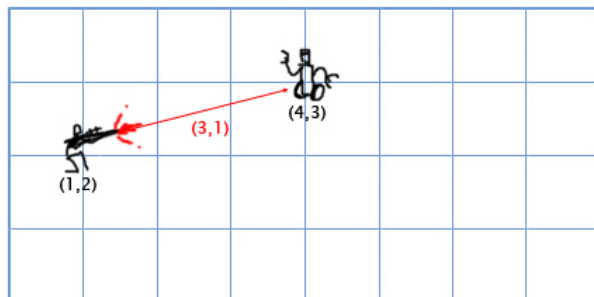
Делаем то-же самое для второго кадра. Добавляем скорость $(1, 2)$ к местоположению $(1, 3)$ и получаем координаты $(2, 5)$. Затем добавляем ускорение $(0, -1)$ к его скорости $(1, 2)$ и получаем новую скорость $(1, 1)$.

Обычно игрок контролирует ускорение игрового персонажа с помощью клавиатуры или геймпада, а игра, в свою очередь, рассчитывает новые значения для скоростей и местоположения, используя физическое сложение (через сложение векторов). Это та-же задача, которая решается в интегральном исчислении, просто мы его сильно упрощаем для нашей игры. Я заметил, что мне намного проще внимательно слушать лекции по интегральному исчислению, думая о практическом его применении, которое мы только что описали.

Вычитание векторов

Вычитание рассчитывается по тому-же принципу что и сложение — вычитаем соответствующие компоненты векторов. Вычитание векторов удобно для получения вектора, который показывает из одного местоположения на другое. Например, пусть игрок находится по координатам $(1, 2)$ с лазерным ружьём, а вражеский робот находится по координатам $(4, 3)$. Чтобы определить вектор движения лазерного луча, который поразит робота, нам надо вычесть местоположение игрока из местоположения робота. Получаем:

$$(4, 3) - (1, 2) = (4-1, 3-2) = (3, 1).$$



Умножение вектора на скаляр

Когда мы говорим о векторах, мы называем отдельные числа скалярами. Например $(3, 4)$ — вектор, а 5 — это скаляр. В играх, часто бывает нужно умножить вектор на число (скаляр). Например, моделируя простое сопротивление воздуха путём умножения скорости игрока на 0.9 в каждом кадре. Чтобы сделать это, нам надо умножить каждый компонент вектора на скаляр. Если скорость игрока $(10, 20)$, то новая скорость будет:

$$0.9 \cdot (10, 20) = (0.9 \cdot 10, 0.9 \cdot 20) = (9, 18).$$

Длина вектора

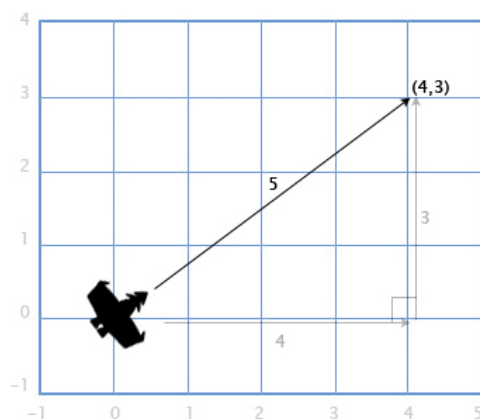
Если у нас есть корабль с вектором скорости $V(4, 3)$, нам также понадобится узнать как быстро он движется, чтобы посчитать потребность в экранном пространстве или сколько потребуется топлива. Чтобы сделать это, нам понадобится найти длину (модуль) вектора V . Длина вектора обозначается вертикальными линиями, в нашем случае длина вектора V будет обозначаться как $|V|$.

Мы можем представить V как прямоугольный треугольник со сторонами 4 и 3 и, применяя теорему Пифагора, получить гипотенузу из выражения: $x^2 + y^2 = h^2$

В нашем случае — длину вектора H с компонентами (x, y) мы получаем из квадратного корня: $\sqrt{x^2 + y^2}$.

Итак, скорость нашего корабля равна:

$$|V| = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$$



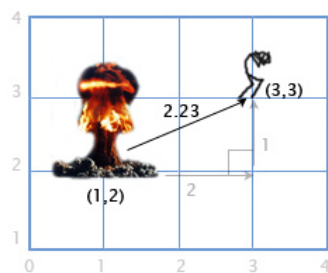
Этот подход используется и для трёхмерных векторов. Длина вектора с компонентами (x, y, z) рассчитывается как $\sqrt{x^2 + y^2 + z^2}$

Расстояние

Если игрок P находится в точке $(3, 3)$, а взрыв произошёл в точке E по координатам $(1, 2)$, нам надо определить расстояние между игроком и взрывом, чтобы рассчитать степень ущерба, нанесённого игроку. Это легко сделать, комбинируя две вышеописанных операции: вычитание векторов и их длину.

Мы вычитаем $P - E$, чтобы получить вектор между ними. А затем определяем длину этого вектора, что и даёт нам искомое расстояние. Порядок следования операндов тут не имеет значения, $|E - P|$ даст тот-же самый результат.

$$\text{Расстояние} = |P - E| = |(3, 3) - (1, 2)| = |(2, 1)| = \sqrt{2^2 + 1^2} = \sqrt{5} = 2.23$$



Нормализация

Когда мы имеем дело с направлениями (в отличие от местоположений и скоростей), важно, чтобы вектор направления имел длину, равную единице. Это сильно упрощает нам жизнь. Например, допустим оружие развёрнуто в направлении $(1, 0)$ и выстреливает снаряд со скоростью 20 метров в секунду. Каков в данном случае вектор скорости для выпущенного снаряда?

Так как вектор направления имеет длину равную единице, мы умножаем направление на скорость снаряда и получаем вектор скорости $(20, 0)$. Если-же вектор направления имеет отличную от единицы длину, мы не сможем сделать этого. Снаряд будет либо слишком быстрым, либо слишком медленным.

Вектор с длиной равной единице называется «нормализованным». Как сделать вектор нормализованным? Довольно просто. Мы делим каждый компонент вектора на его длину. Если, к примеру, мы хотим нормализовать вектор V с компонентами $(3, 4)$, мы просто делим каждый компонент на его длину, то есть на 5, и получаем $(3/5, 4/5)$. Теперь, с помощью теоремы Пифагора, мы убедимся в том, что его длина равна единице:

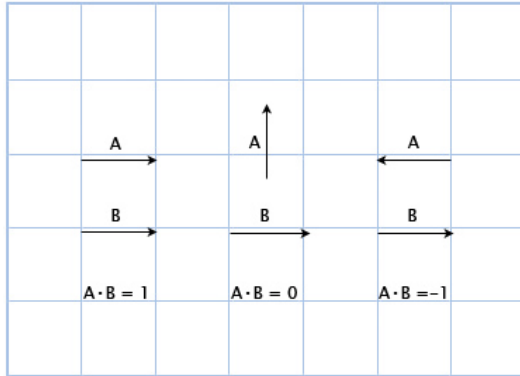
$$(3/5)^2 + (4/5)^2 = 9/25 + 16/25 = 25/25 = 1$$

Скалярное произведение векторов

Что такое скалярное произведение (записывается как \bullet)? Чтобы рассчитать скалярное произведение двух векторов, мы должны умножить их компоненты, а затем сложить полученные результаты вместе

$$(a_1, a_2) \bullet (b_1, b_2) = a_1b_1 + a_2b_2$$

Например: $(3, 2) \bullet (1, 4) = 3 \cdot 1 + 2 \cdot 4 = 11$. На первый взгляд это кажется бесполезным, но посмотрим внимательнее на это:



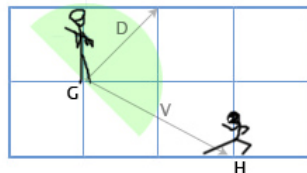
Здесь мы можем увидеть, что если вектора указывают в одном направлении, то их скалярное произведение больше нуля. Когда они перпендикулярны друг другу, то скалярное произведение равно нулю. И когда они указывают в противоположных направлениях, их скалярное произведение меньше нуля. В основном, с помощью скалярного произведения векторов можно рассчитать, сколько их указывает в одном направлении. И хоть это лишь малая часть возможностей скалярного произведения, но уже очень для нас полезная.

Допустим у нас есть стражник, расположенный в $G(1, 3)$ смотрящий в направлении $D(1,1)$, с углом обзора 180 градусов. Главный герой игры подсматривает за ним с позиции $H(3, 2)$. Как определить, находится-ли главный герой в поле зрения стражника или нет? Сделаем это путём скалярного произведения векторов G и V (вектора, направленного от стражника к главному герою). Мы получим следующее:

$$V = H - G = (3, 2) - (1, 3) = (3-1, 2-3) = (2, -1)$$

$$D \bullet V = (1, 1) \bullet (2, -1) = 1 \cdot 2 + 1 \cdot (-1) = 2 - 1 = 1$$

Так как единица больше нуля, то главный герой находится в поле зрения стражника.



Мы уже знаем, что скалярное произведение имеет отношение к определению направления векторов. А каково его более точное определение? Математическое выражение скалярного произведения векторов выглядит так:

$$A \bullet B = |A| |B| \cos \Theta$$

Где Θ (произносится как «theta») — угол между векторами A и B .

Это позволяет нам найти Θ (угол) с помощью выражения:

$$\Theta = \arccos([AB] / [|A| |B|])$$

Как я говорил ранее, нормализация векторов упрощает нашу жизнь. И если A и B нормализованы, то выражение упрощается следующим образом:

$$\Theta = \arccos(AB)$$

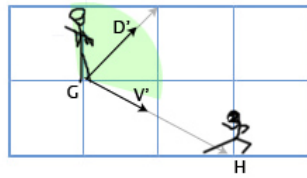
Давайте опять рассмотрим сценарий со стражником. Пусть теперь угол обзора стражника будет равен 120 градусам. Получим нормализованные вектора для направления взгляда стражника (D') и для направления от стражника к главному герою (V'). Затем определим угол между ними. Если угол более 60 градусов (половина от угла обзора), то главный герой находится вне поля зрения стражника.

$$D' = D / |D| = (1, 1) / \sqrt{1^2 + 1^2} = (1, 1) / \sqrt{2} = (0.71, 0.71)$$

$$V' = V / |V| = (2, -1) / \sqrt{2^2 + (-1)^2} = (2, -1) / \sqrt{5} = (0.89, -0.45)$$

$$\Theta = \arccos(D \cdot V') = \arccos(0.71 \cdot 0.89 + 0.71 \cdot (-0.45)) = \arccos(0.31) = 72$$

Угол между центром поля зрения стражника и местоположением главного героя составляет 72 градуса, следовательно стражник его не видит.



Понимаю, что это выглядит довольно сложно, но это потому, что мы всё делаем вручную. В программе это всё довольно просто. Ниже показано как я сделал это в нашей игре [Overgrowth](#) с помощью написанных мной C++ библиотек для работы с векторами:

```
//Инициализируем вектора
vec2 guard_pos = vec2(1,3);
vec2 guard_facing = vec2(1,1);
vec2 hero_pos = vec2(3,2);

//Рассчитываем нормализованные вектора
vec2 guard_facing_n = normalize(guard_facing);
vec2 guard_to_hero = normalize(hero_pos - guard_pos);

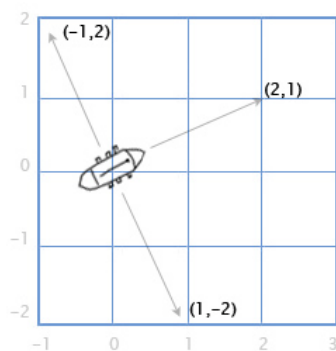
//Рассчитываем угол
float angle = acos(dot(guard_facing_n, guard_to_hero));
```

Векторное произведение

Допустим у нас есть корабль с пушками, которые стреляют в правую и в левую стороны по курсу. Допустим, что лодка расположена вдоль вектора направления (2, 1). В каких направлениях теперь стреляют пушки?

Это довольно просто в двухмерной графике. Чтобы повернуть направление на 90 градусов по часовой стрелке, достаточно поменять местами компоненты вектора, а затем поменять знак второму компоненту.

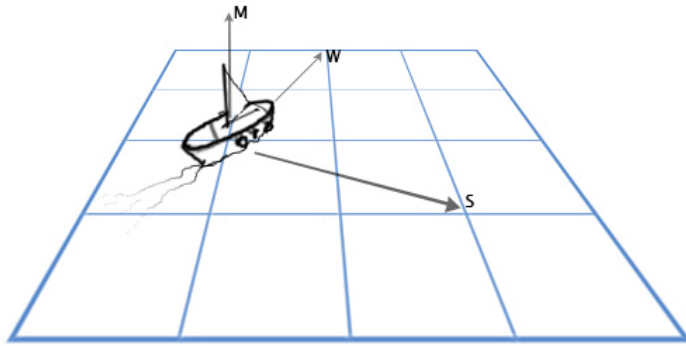
(a, b) превращается в (b, -a). Следовательно у корабля, расположенного вдоль вектора (2, 1), пушки справа по борту будут стрелять в направлении (1, -2), а пушки с левого борта, будут стрелять в противоположном направлении. Меняем знаки у компонент вектора и получаем (-1, 2).



А что если мы хотим рассчитать это всё для трехмерной графики? Рассмотрим пример с кораблём.

У нас есть вектор мачты M, направленной прямо вверх (0, 1, 0) и направление ветра: север-северо-восток W (1, 0, 2). И мы хотим вычислить вектор направления паруса S, чтобы наилучшим образом «поймать ветер».

Для решения этой задачи мы используем векторное произведение: $S = M \times W$.



Векторное произведение $A(a_1, a_2, a_3)$ и $B(b_1, b_2, b_3)$ будет равно:

$$(a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1)$$

Подставим теперь нужные нам значения:

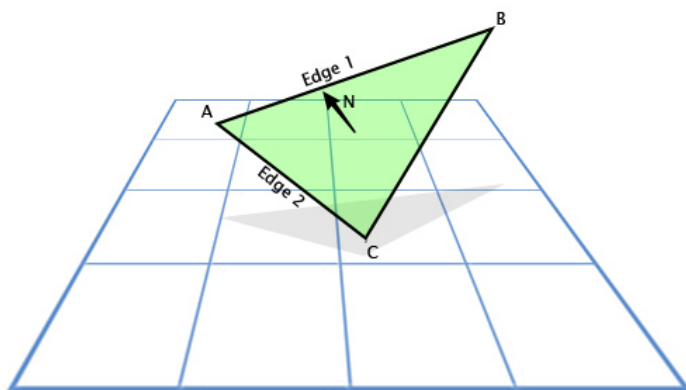
$$S = M \times W = (0, 1, 0) \times (1, 0, 2) = ([1 \cdot 2 - 0 \cdot 0], [0 \cdot 1 - 0 \cdot 2], [0 \cdot 0 - 1 \cdot 1]) = (2, 0, -1)$$

Для расчётов вручную довольно сложно, но для графических и игровых приложений я рекомендую написать функцию, подобную той, что указана ниже и не вдаваться более в детали подобных расчётов.

```
vec3 cross(vec3 a, vec3 b) {
    vec3 result;
    result[0] = a[1] * b[2] - a[2] * b[1];
    result[1] = a[2] * b[0] - a[0] * b[2];
    result[2] = a[0] * b[1] - a[1] * b[0];
    return result;
}
```

Векторное произведение часто используется в играх, чтобы рассчитать нормали к поверхностям. Направления, в которых «смотрит» та или иная поверхность. Например, рассмотрим треугольник с векторами вершин A, B и C. Как мы найдем направление в котором «смотрит» треугольник, то есть направление перпендикулярное его плоскости? Это кажется сложным, но у нас есть инструмент для решения этой задачи.

Используем вычитание, для определения направления из A в C ($C - A$), пусть это будет «грань 1» (Edge 1) и направление из A в B ($B - A$), пусть это будет «грань 2» (Edge 2). А затем применим векторное произведение, чтобы найти вектор, перпендикулярный им обоим, то есть перпендикулярный плоскости треугольника, также называемый «нормалью к плоскости».



Вот так это выглядит в коде:

```
vec3 GetTriangleNormal(vec3 a, vec3 b, vec3 c) {
    vec3 edge1 = b - a;
    vec3 edge2 = c - a;
    vec3 normal = cross(edge1, edge2);
    return normal;
}
```

В играх основное выражение освещённости записывается как $N \cdot L$, где N — это нормаль к освещаемой поверхности, а L — это нормализованный вектор направления света. В результате поверхность выглядит яркой, когда на неё прямо падает

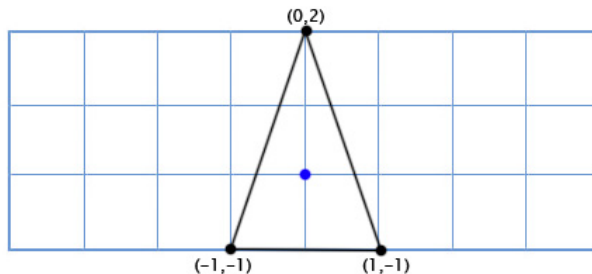
свет, и тёмной, когда этого не происходит.

Теперь перейдем к рассмотрению такого важного для разработчиков игр понятия, как «матрица преобразований» (transformation matrix).

Для начала изучим «строительные блоки» матрицы преобразований.

Базисный вектор

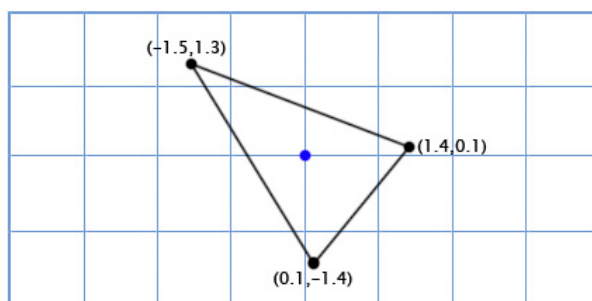
Допустим мы пишем игру *Asteroids* на очень старом «железе» и нам нужен простой двухмерный космический корабль, который может свободно вращаться в своей плоскости. Модель корабля выглядит так:



Как нам рисовать корабль, когда игрок поворачивает его на произвольный градус, скажем 49 градусов против часовой стрелки. Используя тригонометрию, мы можем написать функцию двухмерного поворота, которая принимает координаты точки и угол поворота, и возвращает координаты смещённой точки:

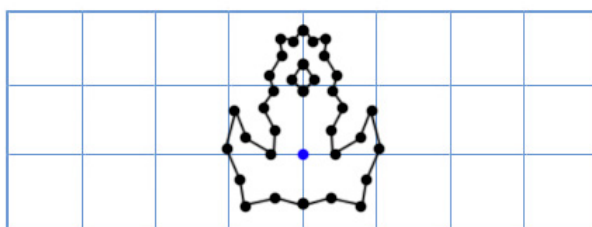
```
vec2 rotate(vec2 point, float angle){
    vec2 rotated_point;
    rotated_point.x = point.x * cos(angle) - point.y * sin(angle);
    rotated_point.y = point.x * sin(angle) + point.y * cos(angle);
    return rotated_point;
}
```

Применяя эту функцию ко всем трём точкам, мы получим следующую картину:

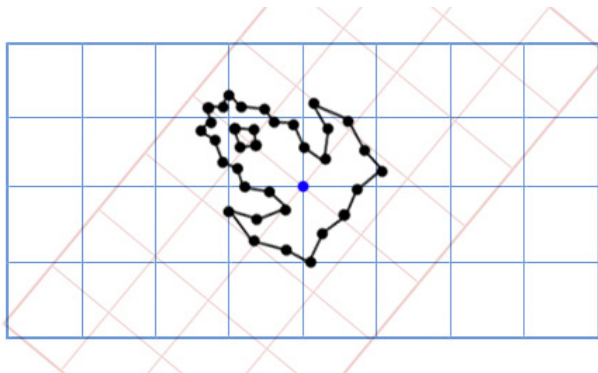


Операции с синусами и косинусами работают довольно медленно, но так как мы делаем расчёты лишь для трёх точек, это будет нормально работать даже на старом «железе» (прим. переводчика: в случаях, когда предполагается интенсивное использование тригонометрических функций, для ускорения вычислений, в памяти организуют таблицы значений для каждой функции и рассчитывают их во время запуска приложения. Затем при вычислении той или иной тригонометрической функции просто производится обращение к таблице).

Пусть теперь наш корабль выглядит вот так:



Теперь старый подход будет слишком медленным, так как надо будет поворачивать довольно большое количество точек. Одно из элегантных решений данной проблемы будет звучать так — «Что если вместо поворота каждой точки модели корабля, мы повернём координатную решётку нашей модели?»



Как это работает? Давайте посмотрим внимательнее, что собой представляют координаты.

Когда мы говорим о точке с координатами (3, 2), мы говорим, что её местоположение находится в трех шагах от точки отсчёта по координатной оси X, и двух шагах от точки отсчёта по координатной оси Y.

По-умолчанию координатные оси расположены так: вектор координатной оси X (1, 0), вектор координатной оси Y (0, 1). И мы получим расположение: $3(1, 0) + 2(0, 1)$. Но координатные оси не обязательно должны быть в таком положении. Если мы повернём координатные оси, в это-же время мы повернём все точки в координатной решётке.

Чтобы получить повернутые оси X и Y мы применим тригонометрические функции, о которых говорили выше. Если мы поворачиваем на 49 градусов, то новая координатная ось X будет получена путём поворота вектора (0, 1) на 49 градусов, а новая координатная ось Y будет получена путём поворота вектора (0, 1) на 49 градусов. Итак вектор новой оси X у нас будет равен (0.66, 0.75), а вектор новой оси Y будет (-0.75, 0.66). Сделаем это вручную для нашей простой модели из трёх точек, чтобы убедиться, что это работает так, как нужно:

Координаты верхней точки (0, 2), что означает, что её новое местоположение находится в 0 на новой (повёрнутой) оси X и 2 на новой оси Y:

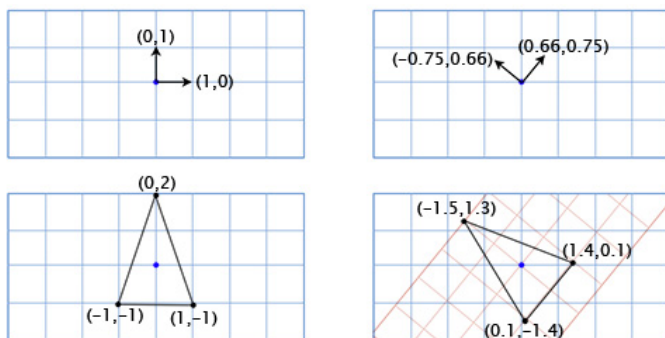
$$0 \cdot (0.66, 0.75) + 2 \cdot (-0.75, 0.66) = (-1.5, 1.3)$$

Нижняя левая точка (-1, -1), что означает, что её новое местоположение находится в -1 на повернутой оси X, и -1 на повернутой оси Y:

$$-1 \cdot (0.66, 0.75) + -1 \cdot (-0.75, 0.66) = (0.1, -1.4)$$

Нижняя правая точка (1, -1), что означает её новое местоположение находится в 1 на повернутой оси X, и -1 на повернутой оси Y

$$1 \cdot (0.66, 0.75) + -1 \cdot (-0.75, 0.66) = (1.4, 0.1)$$



Мы показали, как координаты корабля отображаются в другой координатной сетке с повернутыми осями (или «базисными векторами»). Это удобно в нашем случае, так как избавляет нас от необходимости применять тригонометрические преобразования к каждой из точек модели корабля.

Каждый раз, когда мы изменяем базисные вектора (1, 0) и (0, 1) на (a, b) и (c, d), то новая координата точки (x, y) может быть найдена с помощью выражения:

$$x(a,b) + y(c,d)$$

Обычно базисные вектора равны (1, 0) и (0, 1) и мы просто получаем $x(1, 0) + y(0, 1) = (x, y)$, и нет необходимости заботиться об этом дальше. Однако, важно помнить, что мы можем использовать и другие базисные вектора, когда нам это нужно.

Матрицы

Матрицы похожи на двумерные вектора. Например, типичная 2x2 матрица, может выглядеть так:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

Когда вы умножаете матрицу на вектор, вы суммируете скалярное произведение каждой строки с вектором, на который происходит умножение. Например, если мы умножаем вышеприведённую матрицу на вектор (x, y) , то мы получаем:

$$(a,c) \cdot (x,y) + (b,d) \cdot (x,y)$$

Будучи записанным по-другому, это выражение выглядит так:

$$x(a,b) + y(c,d)$$

Выглядит знакомо, не так-ли? Это в точности такое-же выражение, которые мы использовали для смены базисных векторов. Это означает, что умножая 2x2 матрицу на двухмерный вектор, мы тем самым меняем базисные вектора. Например, если мы вставим стандартные базисные вектора в $(1, 0)$ и $(0, 1)$ в колонки матрицы, то мы получим:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Это единичная матрица, которая не даёт эффекта, который мы можем ожидать от нейтральных базисных векторов, которые мы указали. Если-же мы повернём базисные вектора на 49-градусов, то мы получим:

$$\begin{bmatrix} 0.66 & -0.75 \\ 0.75 & 0.66 \end{bmatrix}$$

Эта матрица будет поворачивать двухмерный вектор на 49 градусов против часовой стрелки. Мы можем сделать код нашей игры Asteriods более элегантным, используя матрицы вроде этой. Например, функция поворота нашего корабля может выглядеть так:

```
void RotateShip(float degrees){
    Matrix2x2 R = GetRotationMatrix(degrees);
    for(int i=0; i<num_points; ++i){
        rotated_point[i] = R * point[i];
    }
}
```

Однако, наш код будет ещё более элегантным, если мы сможем также включить в эту матрицу перемещение корабля в пространстве. Тогда у нас будет единая структура данных, которая будет заключать в себе и применять информацию об ориентации объекта и его местоположении в пространстве.

К счастью есть способ добиться этого, хоть это и выглядит не очень элегантно. Если мы хотим переместиться с помощью вектора (e, f) , мы лишь включаем его в нашу матрицу преобразования:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

И добавляем дополнительную единицу в конец каждого вектора, определяющего местоположение объекта, например так:

$$\begin{bmatrix} x & y & 1 \end{bmatrix}$$

Теперь, когда мы перемножаем их, мы получаем:

$$(a, c, e) \cdot (x, y, 1) + (b, d, f) \cdot (x, y, 1) + (0, 0, 1) \cdot (x, y, 1)$$

Что, в свою очередь, может быть записано как:

$$x(a, b) + y(c, d) + (e, f)$$

Теперь у нас есть полный механизм трансформации, заключённый в одной матрице. Это важно, если не принимать в расчёт элегантность кода, так как с ней мы теперь можем использовать все стандартные манипуляции с матрицами. Например перемножить матрицы, чтобы добавить нужный эффект, или мы можем инвертировать матрицу, чтобы получить прямо противоположное положение объекта.

Трёхмерные матрицы

Матрицы в трёхмерном пространстве работают так-же как и в двухмерном. Я приводил примеры с двухмерными векторами и матрицами, так как их просто отобразить с помощью дисплея, показывающего двухмерную картинку. Нам просто надо определить три колонки для базисных векторов, вместо двух. Если базисные вектора это (a,b,c) , (d,e,f) and (g,h,i) то наша матрица будет выглядеть так:

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Если нам нужно перемещение (j,k,l), то мы добавляем дополнительную колонку и строку, как говорили раньше:

```
[a d g j
 b e h k
 c f i l
 0 0 0 1]
```

И добавляем единицу [1] в вектор, как здесь:

```
[x y z 1]
```

Вращение в двухмерном пространстве

Так как в нашем случае у нас только одна ось вращения (расположенная на дисплее), единственное, что нам надо знать, это угол. Я говорил об этом ранее, упоминая, что мы можем применять тригонометрические функции для реализации функции двухмерного вращения наподобие этой:

```
vec2 rotate(vec2 point, float angle){
    vec2 rotated_point;
    rotated_point.x = point.x * cos(angle) - point.y * sin(angle);
    rotated_point.y = point.x * sin(angle) + point.y * cos(angle);
    return rotated_point;
}
```

Более элегантно это можно выразить в матричной форме. Чтобы определить матрицу, мы можем применить эту функцию к осям (1, 0) и (0, 1) для угла Θ , а затем включить полученные оси в колонки нашей матрицы. Итак, начнём с координатной оси X (1, 0). Если мы применим к ней нашу функцию, мы получим:

$$(1 * \cos(\Theta) - 0 * \sin(\Theta), 1 * \sin(\Theta) + 0 * \cos(\Theta)) = (\cos(\Theta), \sin(\Theta))$$

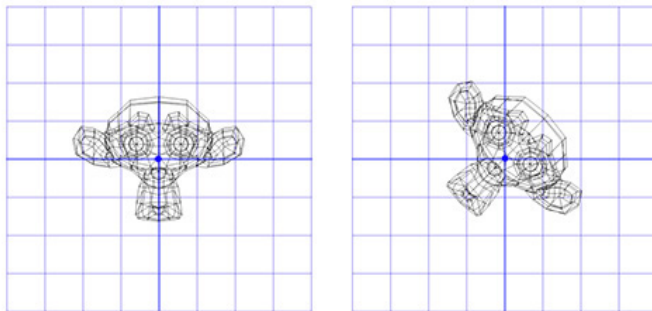
Затем, мы включаем координатную ось Y (0, 1). Получим:

$$(0 * \cos(\Theta) - 1 * \sin(\Theta), 0 * \sin(\Theta) + 1 * \cos(\Theta)) = (-\sin(\Theta), \cos(\Theta))$$

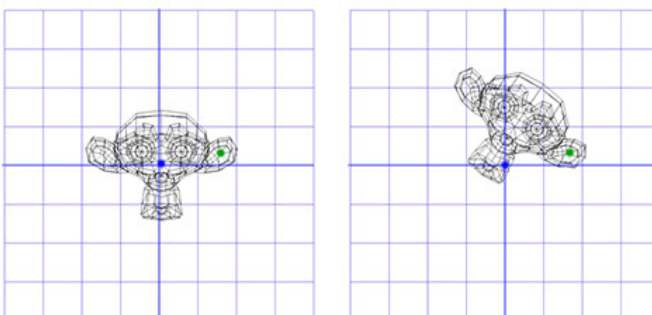
Включаем полученные координатные оси в матрицу, и получаем двухмерную матрицу вращения:

```
[cos(Θ)  -sin(Θ)
 sin(Θ)   cos(Θ)]
```

Применим эту матрицу к Сюзанне, мартышке из графического пакета Blender. Угол поворота Θ равен 45 градусов по часовой стрелке.



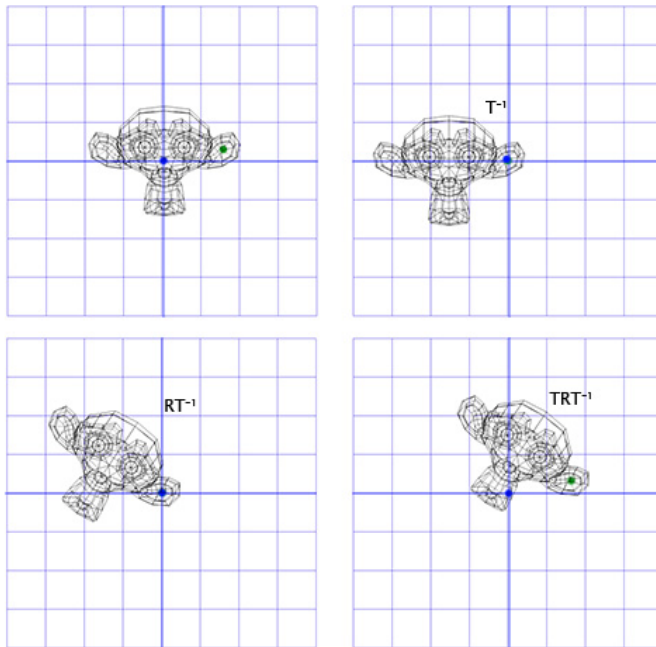
Как видите — это работает. Но что если нам надо осуществить вращение вокруг точки, отличной от (0, 0)? Например, мы хотим вращать голову мартышки вокруг точки, расположенной в её ухе:



Чтобы сделать это, мы можем начать с создания матрицы перемещения (translation matrix) T, которая перемещает объект

из начальной точки в точку вращения в ухе мартышки, и матрицу вращения R , для вращения объекта вокруг начальной точки. Теперь для вращения вокруг точки, расположенной в ухе, мы можем сперва переместить точку в ухе на место начальной точки, с помощью инвертирования матрицы T , записанной как T^{-1} . Затем, мы вращаем объект вокруг начальной точки, с помощью матрицы R , а затем применяем матрицу T для перемещения точки вращения назад, к своему исходному положению.

Ниже дана иллюстрация к каждому из описанных шагов:



Это важный шаблон, который мы будем применять позднее — применение вращения для двух противоположных трансформаций позволяет нам вращать объект в другом «пространстве». Что очень удобно и полезно.

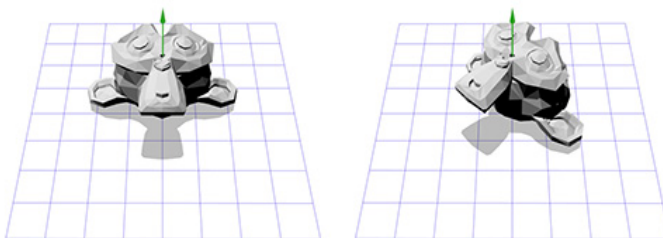
Теперь рассмотрим трёхмерное вращение.

Трёхмерное вращение

Вращение вокруг оси Z работает по тому-же принципу, что и вращение в двухмерном пространстве. Нам лишь нужно изменить нашу старую матрицу, добавив к ней дополнительную колонку и строку:

```
[cos(θ)  -sin(θ)  0
 sin(θ)   cos(θ)  0
 0         0       1]
```

Применим эту матрицу к трёхмерной версии Сюзанны, мартышки из пакета Blender. Угол поворота θ пусть будет равен 45 градусов по часовой стрелке.



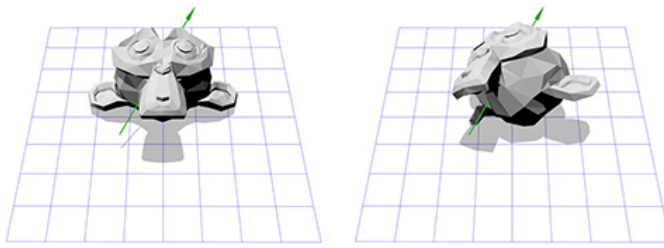
То-же самое. Вращение только вокруг оси Z ограничивает нас, как насчёт вращения вокруг произвольной оси?

Вращение, определяемое осью и углом (Axis-angle rotation)

Представление вращения, определяемого осью и углом, также известно как вращение в экспоненциальных координатах, параметризованное вращением двух величин. Вектора, определяющего вращение направляющей оси (прямая линия) и угла, описывающего величину поворота вокруг этой оси. Вращение осуществляется согласно правилу правой руки.

Итак, вращение задаётся двумя параметрами (axis, angle), где axis — вектор оси вращения, а angle — угол вращения. Этот приём довольно прост и являет собой отправную точку для множества других операций вращения, с которыми я работаю. Как практически применить вращение, определяемое осью и углом?

Допустим мы имеем дело с осью вращения, показанной на рисунке ниже:



Мы знаем как вращать объект вокруг оси Z, и мы знаем как вращать объект в других пространствах. Итак, нам лишь надо создать пространство, где наша ось вращения будет являться осью Z. И если эта ось будет осью Z, то что будет являться осями X и Y? Займемся вычислениями сейчас.

Чтобы создать новые оси X и Y нам нужно лишь выбрать два вектора, которые перпендикулярны новой оси Z и перпендикулярны друг другу. Мы уже говорили ранее о векторном умножении, которое берёт два вектора и даёт в итоге перпендикулярный им вектор.

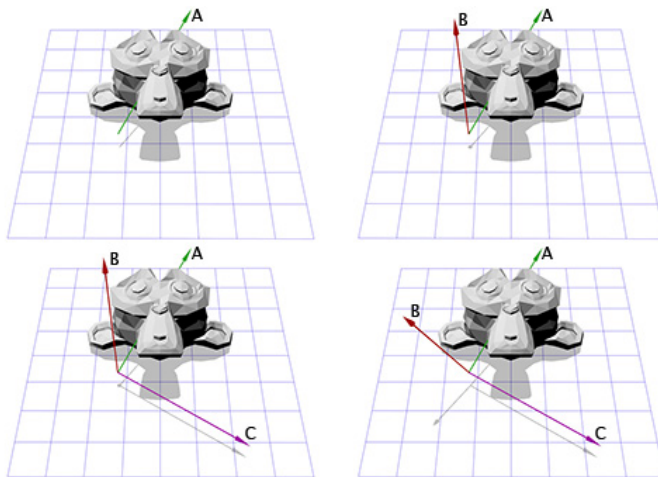
У нас есть один вектор сейчас, это ось вращения, назовём его A. Возьмём теперь случайный другой вектор B, который находится не в том-же направлении, что и вектор A. Пусть это будет (0, 0, 1) к примеру.

Теперь мы имеем ось вращения A и случайный вектор B, мы можем получить нормаль C, через векторное произведение A и B. C перпендикулярен векторам A и B. Теперь мы делаем вектор B перпендикулярным векторам A и C через их векторное произведение. И всё, у нас есть все нужные нам оси координат.

На словах это звучит сложно, но довольно просто выглядит в коде или будучи показанным в картинках. Ниже показано, как это выглядит в коде:

```
B = (0,0,1);
C = cross(A,B);
B = cross(C,A);
```

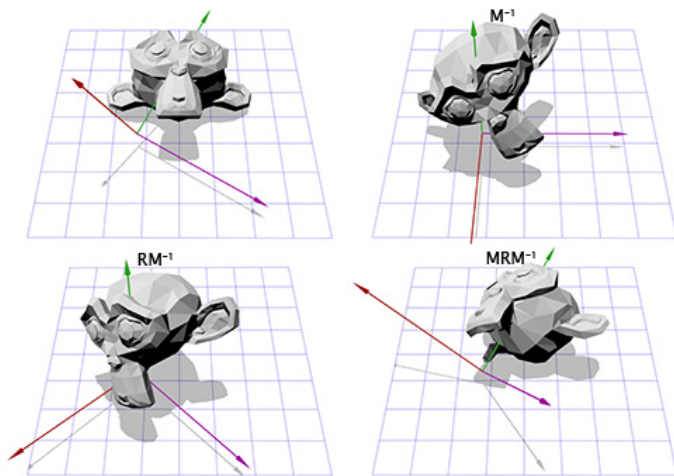
Тут показана иллюстрация для каждого шага:



Теперь, имея информацию о новых координатных осях, мы можем составить матрицу M, включив каждую ось как колонку в эту матрицу. Нам надо убедиться, что вектор A является третьей колонкой, чтобы он был нашей новой осью координат Z.

```
[B0 C0 A0
 B1 C1 A1
 B2 C2 A2]
```

Теперь это похоже на то, что мы делали для поворота в двухмерном пространстве. Мы можем применить инвертированную матрицу M, чтобы переместиться в новую систему координат, затем произвести вращение, согласно матрице R, чтобы повернуть объект вокруг оси Z, затем применить матрицу M, чтобы вернуться в исходное координатное пространство.



Теперь мы можем вращать объект вокруг произвольной оси. В конце концов мы можем просто создать матрицу $T = T = M^{-1}RM$ и использовать её много раз, без дополнительных усилий с нашей стороны. Есть более эффективные способы конвертирования вращений, определяемых осью и углом во вращения, определяемые матрицами. Просто описанный нами подход показывает многое из того, о чём мы говорили ранее.

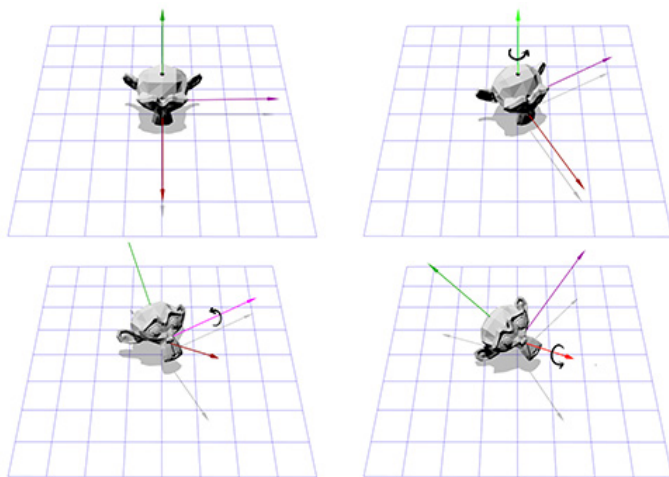
Вращение, определяемое осью и углом, возможно, самый интуитивно понятный способ. Применяя его, очень легко инвертировать поворот, поменяв знак у угла, и легко интерполировать, путём интерполяции угла. Однако тут есть серьёзное ограничение, и заключается оно в том, что такое вращение не является суммирующим. То есть вы не можете комбинировать два вращения, определяемых осью и углом в третье.

Вращение, определяемое осью и углом — хороший способ для начала, но оно должно быть преобразовано во что-то другое, чтобы использоваться в более сложных случаях.

Эйлеровские углы

Эйлеровские углы представляют собой другой способ вращения, заключающийся в трёх вложенных вращениях относительно осей X, Y и Z. Вы, возможно, сталкивались с их применением в играх, где камера показывает действие от первого лица, либо от третьего лица.

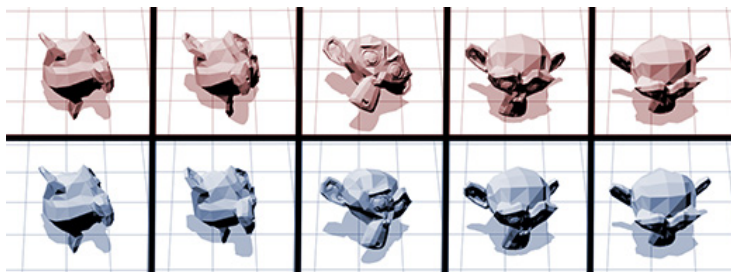
Допустим вы играете в шутер от первого лица и вы повернулись на 30 градусов влево, а затем посмотрели на 40 градусов вверх. В конце-концов в вас стреляют, попадают, и, в результате удара, камера поворачивается вокруг своей оси на 45 градусов. Ниже показано вращение с помощью углов Эйлера (30, 40, 45).



Углы Эйлера — удобное и простое в управлении средство. Но у этого способа есть два недостатка.

Первый, это вероятность возникновения ситуации под названием «блокировка оси» или «шарнирный замок» (gimbal lock). Представьте, что вы играете в шутер от первого лица, где вы можете посмотреть влево, вправо, вверх и вниз или повернуть камеру вокруг зрительной оси. Теперь представьте, что вы смотрите прямо вверх. В этой ситуации попытка взглянуть налево или направо будет аналогична попытке вращения камеры. Всё что мы можем в этом случае, это вращать камеру вокруг своей оси, либо посмотреть вниз. Как вы можете представить, это ограничение делает непрактичным применение углов Эйлера в лётных симуляторах.

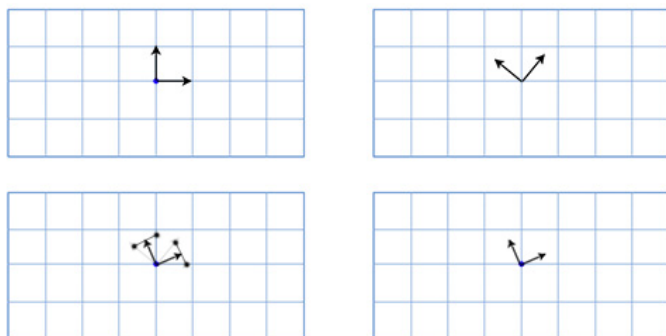
Второе — интерполяция между двумя эйлеровскими углами вращения не даёт кратчайшего пути между ними. Например, у вас две интерполяции между двумя одинаковыми вращениями. Первая использует интерполяцию эйлеровского угла, вторая использует сферическую линейную интерполяцию (spherical linear interpolation (SLERP)), чтобы найти кратчайший путь.



Итак, что-же больше подойдет для интерполяции вращений? Может быть матрицы?

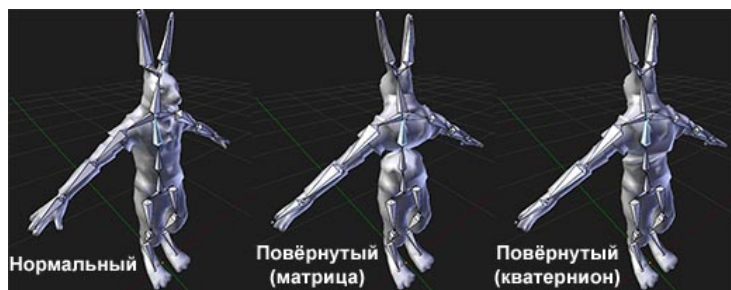
Вращение с помощью матриц

Как мы уже говорили ранее, матрицы вращения хранят в себе информацию о трёх осях. Это означает, что интерполяция между двумя матрицами лишь линейно интерполирует каждую ось. В результате это даёт нам эффективный путь, то так-же приносит новые проблемы. Например, тут показаны два вращения и одно интерполированное полу-вращение:



Как вы можете заметить, интерполированное вращение значительно меньше, чем любое из исходных вращений, и две оси более не перпендикулярны друг другу. Это логично, если вдуматься — середина отрезка, соединяющего любые две точки на сфере будет расположена ближе к центру сферы.

Это в свою очередь порождает известный «эффект фантика» (candy wrapper effect), при применении скелетной анимации. Ниже показана демонстрация этого эффекта на примере кролика из нашей игры *Overgrowth* (прим. переводчика: обратите внимание на середину туловища кролика).



Вращение, основанное на матричных операциях, очень полезно, так как они могут аккумулировать вращения без всяких проблем, вроде блокировки оси (gimbal lock), и может очень эффективно применяться к точкам сцены. Вот почему поддержка вращения на матрицах встроена в графические карты. Для любого типа трёхмерной графики матричный формат вращения — это всегда итоговый применяемый способ.

Однако, как мы уже знаем, матрицы не очень хорошо интерполируются, и они не столь интуитивно понятны.

Итак, остался только один главный формат вращения. Последний, но тем не менее, важный.

Кватернионы

Что-же такое кватернионы? Если очень кратко, то это альтернативный вариант вращения, основанный на оси и угле (axis-angle rotation), который существует в пространстве.

Подобно матрицам они могут аккумулировать вращения, то есть вы можете составлять из них цепочку вращений, без опаски получить блокировку оси (gimbal lock). И в то-же время, в отличие от матриц, они могут хорошо интерполироваться из одного положения в другое.

Являются-ли кватернионы лучшим решением, нежели остальные способы вращений (rotation formats)?

На сегодняшний день они комбинируют все сильные стороны других способов вращений. Но у них есть два слабых места, рассмотрев которые, мы придём к выводу, что кватернионы лучше использовать для промежуточных вращений. Итак, каковы недостатки кватернионов.

Во-первых кватернионы непросто отобразить на трёхмерном пространстве. И мы вынуждены всегда реализовывать вращение более простым способом, а затем конвертировать его. Во-вторых, кватернионы не могут эффективно вращать точки, и мы вынуждены конвертировать их в матрицы, чтобы повернуть значительное количество точек.

Это означает, что вы скорее всего не начнете или не закончите серию вращений с помощью кватернионов. Но с их помощью можно реализовать промежуточные вращения более эффективно, нежели при применении любого другого подхода.

«Внутренняя кухня» механизма кватернионов не очень понятна и не интересна мне. И, возможно, не будет интересна и вам, если только вы не математик. И я советую вам найти библиотеки, которые работают с кватернионами, чтобы облегчить вам решение ваших задач с их помощью.

Математические библиотеки «Bullet» или «Blender» будут хорошим вариантом для начала.

линейная алгебра, игростроение				
+278	95003	1382	David Rosen	Rafael 59,3

Бриться


**Бриться или
ходить с бородой?**

С бородой


Похожие публикации

- Несколько слов о «линейной» регрессии 2 сентября в 03:05
- Линейный криптоанализ для чайников 25 августа в 05:39
- Решение задачи линейной регрессии с помощью быстрого преобразования Хафа 20 июня в 13:30
- Линейная регрессия на пальцах в распознавании 10 декабря 2013 в 09:06
- Оценка результатов линейной регрессии 25 сентября 2013 в 14:00
- Интерполяция + (линейная | логарифмическая) шкала + C++ 16 ноября 2012 в 15:00
- Испытания протокола TCP с линейным сетевым кодированием (TCP/NC) 24 октября 2012 в 17:43
- Основы реляционной алгебры 16 июня 2012 в 19:51
- Конкурс на решение целочисленной системы линейных уравнений 1 февраля 2011 в 11:39
- Wason And Eggs. Велосипед с яйцами и линейной алгеброй! 28 января 2010 в 00:30


Комментарии (56)

-  Skiminok, 6 ноября 2011 в 15:42 #


+13

Как-то все поверхностно. Матбаза не выходит за рамки первого семестра первого курса произвольного технического вуза. Я надеялся, что хоть про кватернионы человек поведает с той же степенью детальности, а он отделался «Извините, сам фишку не секу» и свалил.
-  s2erg, 6 ноября 2011 в 15:46 # ↑


+18

Возможно, для кого-то этот материал и кажется поверхностным. Но вот в образовательных школах, ИМХО, так учить алгебру было бы гораздо привлекательнее.
-  shimapa23, 6 ноября 2011 в 19:32 # ↑


-4

В среднеобразовательных школах не учат линейную алгебру (по крайней мере в моей). А вот на первом курсе фин универа учат =)
-  define_rak, 6 ноября 2011 в 20:01 # ↑

+9

Начала т.н. «линейки» — геометрия 9-го класса среднеобразовательной школы (по крайней мере в Украине).
-  MAXHO, 6 ноября 2011 в 21:25 # ↑

0

Как вы учили ФИЗИКУ? Нет преподаватели разные бывают, но механика без векторов... Я в шоке. ЗЫ. Ну и очевидно — личный опыт не универсален.
-  sl4mmer, 6 ноября 2011 в 21:22 # ↑

-4

Статья имхо ни о чем — все сведения не выходят за рамки 1го курса института. Изложено конечно хорошо, но зачем?
- Для гуманитария решившего внезапно заняться геймдевелопментом — будет полезнее взять книжку по линейной, векторной алгебре и аналитической геометрии. Там все это есть (и много чего еще).



dmitriid, 7 ноября 2011 в 01:42 # ↑

+11

> Статья имхо ни о чем — все сведения не выходят за рамки 1го курса института.

У меня первый курс института был 11 лет тому назад. С тех пор линейная алгебра не понадобилась практически ни разу (ну, векторы — уж точно).

Если я вдруг решу заниматься игроделанием, то такой вот обзор мегаполезен для того, чтобы освежить все это в памяти.



NickLion, 7 ноября 2011 в 10:50 # ↑

+2

Статеек такого уровня вагон и маленькая тележка. А если первый курс был давно и вспомнить хорошие книги по линейке трудно, то берётся книга, например «Интерактивная компьютерная графика. Вводный курс на базе OpenGL» Эдварда Эйнджела, где всё это расписано (есть раздел по математике) и значительно лучше. И пощупать на реальной программке можно. Описание книги немного ориентировано на UNIX системы, но перенести на Windows несложно, потому что в примерах используется GLUT.

В этой статье даже не рассмотрены такие базовые вещи, как переход к 4-мерным векторам для единообразного описания вектора и точки в пространстве.



NickLion, 7 ноября 2011 в 10:51 # ↑

0

PS как переводчик автор поработал хорошо, но изначальная статья — не стоит внимания



dmitriid, 7 ноября 2011 в 11:04 # ↑

0

И хорошо, что таких статей вагон и маленькая тележка. Именно для того, чтобы потом перейти к той же книге (или не переходить, если уже вспомнил) :)



NickLion, 7 ноября 2011 в 11:09 # ↑

-2

Зачем увеличивать энтропию?



dmitriid, 7 ноября 2011 в 11:13 # ↑

+1

Нет единственно верного способа что-то рассказать и показать.



NickLion, 7 ноября 2011 в 11:28 # ↑

-1

С этим утверждением согласен, но как по мне данная статья (исходный материал) не блещет вообще ничем. Общие сведения сумбурны, детально не раскрыто.



dmitriid, 7 ноября 2011 в 11:37 # ↑

0

Тут и так несколько экранов текста :) Детально потянуло бы на брошюру. А так — базовые сведения, и более-менее понятно, куда копать дальше.



Arris, 8 ноября 2011 в 08:11 # ↑

0

Напишите лучше.



NickLion, 8 ноября 2011 в 15:05 # ↑

0

А этот материал и вправду нужен? Может я не прав, но его вполне достаточно в интернете. И хороших книг достаточно. Одну из них я привёл.



NickLion, 7 ноября 2011 в 11:09 # ↑

0

PPS и да, лёгкое упоминание о 4-й компоненте с неясным утверждением, что к вектору добавляется 1 — это не объяснение перехода

Тем более, что для вектора 0, а 1 — это точка, поэтому при переносе вектор не меняется, а точка меняется, далее логично при сложении 2-х векторов получается вектор $0+0=0$, а при сложении вектора с точкой — точка $0+1=1$, при сложении двух точек — НЕХ — $1+1=2$, а выпуклая комбинация любого количества точек — снова точка.



NickLion, 7 ноября 2011 в 10:41 # ↑

0

Да никто не будет так читать в школах — во-первых, не все учителя математики в школах смогут это осилить, во-вторых, не все ученики вообще сколь-нибудь интересуются математикой. Поэтому максимум в технических лицезах такое можно изучать и, возможно, изучается. А в университете — на первом курсе такое учится и так.



Boba_Fett, 6 ноября 2011 в 17:56 # ↑

+5

Удивлён, что комментарий заминусовали.

Пробежался глазами по заголовкам. Половину упомянутого рассказывали в школе, «матрицы» — первый курс института. Хабр забит гуманитариями?

Про кватернионы и вращения в английской википедии неплохо написано. Книжку не посоветую: сам учил по сложной.



ComodoHacker, 6 ноября 2011 в 22:29 # ↑

0

Вы знаете, для многих закончивших технические вузы и это черезчур много. :) Так что все ОК. Когда человек созреет, возьмет серьезные книжки.



andrewsch, 6 ноября 2011 в 15:44 #

+8

Вот-бы нам так вышку в универе читали — с мартышками и стражниками. А то такая нудятина была...



JagaJaga, 8 ноября 2011 в 18:23 # ↑

+1

Согласен, сейчас как раз изучали аналитическую геометрию, ох как приятнее было бы с использованием таких примеров.



Lof, 6 ноября 2011 в 15:54 #

+17

Статья вредная, с кучей ошибок и просто не правды.

Про кватернионы лучше почитать тут www.gamedev.ru/code/articles/?id=4215

wat.gamedev.ru/articles/quaternions



Rafael, 6 ноября 2011 в 17:01 # ↑

+10

Про кучу ошибок и неправду можно поподробнее?

Спасибо.



HomoLuden, 7 ноября 2011 в 10:53 # ↑

+3

> Эйлеровские углы представляют собой другой способ вращения, заключающийся в трёх вложенных вращениях относительно осей X, Y и Z.

Углы Эйлера это три последовательных поворота вокруг OZ, потом OX, и в конце опять вокруг OZ. Удобно описывать и строить матрицы направляющих косинусов, но такая схема имеет проблему "Gimball Lock"

И вообще тема поворотов не раскрыта... Как строится матрица конечного поворота (элементарная и совокупная по трем поворотам)? Где уравнения перехода от одной системы координат в другую? Каково физический (геометрический) смысл элементов матрицы направляющих косинусов?



HomoLuden, 17 ноября 2011 в 10:04 # ↑

0

Почитал еще раз посты в сети...

Формулировки расхожие. В одних источниках вообще все схемы поворотов называют «Эйлеровскими углами», в других только такие схемы, где первый и третий повороты осуществляются вокруг осей с одним и тем же индексом, а второй вокруг одной из двух оставшихся осей. В третьих источниках «Эйлеровской» называют совершенно конкретную схему.

Есть еще так называемая схема углов «Эйлера-Крылова», в которой все три поворота осуществляются вокруг разных осей.



ertaquo, 6 ноября 2011 в 16:08 #

+3

А еще есть хорошая статья по двумерному пространству. Там тоже самые азы, но по большей части именно эти азы и нужны.



Krovosos, 6 ноября 2011 в 18:23 # ↑

+16

Как ни странно, я — автор статьи по ссылке.

Могу попробовать ответить на вопросы, если у кого возникнут :)



Calvrack, 6 ноября 2011 в 16:21 #

+2

К сожалению алгебру невозможно изучить вот так — по кейс-стади. То-есть соответствующий раздел всегда хорошо проиллюстрировать практикой. Но роль алгебры в разработке игр идет куда дальше, чем «по-вращать».



Rafael, 6 ноября 2011 в 17:08 # ↑

0

По примерам невозможно изучить ни одну научную дисциплину, да и с ненаучными тоже не всё слава богу.

Лично для меня, это шпаргалка, которая может быть полезной.



KirEv, 12 ноября 2011 в 03:39 # ↑

0

Не согласен, в статье показано, каким образом можно применить теоретическую работу с векторами, пусть даже в Декартовой системе координат — эти простые примеры хорошо демонстрируют и ассоциации возникают, а запоминать формулы с ассоциациями гораздо проще, чем сплошную теорию.



KirEv, 12 ноября 2011 в 03:37 # ↑

0

Там где «куда дальше» — уже не алгебра, а серьезный матан и выше.



mclander, 6 ноября 2011 в 17:04 #

-3

вот именно поэтому многие алгебру не знают, а те кто знали, позабывали большую часть.

+2

**Yakud**, 6 ноября 2011 в 17:26 #

Давно искал подобный материал, для усвоения и применения изученного в школе и универе. Большое спасибо за перевод!

**Rafael**, 6 ноября 2011 в 17:30 # ↑

0

Пожалуйста.

**irafa**, 6 ноября 2011 в 18:12 #

+2

когда писал свою первую игру потратил месяц на повторение всего что связано в линейной алгеброй и геометрией и пытался найти применение этому... если бы тогда была такая статья я бы сэкономил кучу времени, для новичков очень полезно для понимания и старта!

**aml**, 6 ноября 2011 в 19:07 # ↑

+4

Самый смак — это когда сначала поиграешься с 3D-графикой и векторным кун-фу на компьютере, а потом приходишь на лекции по аналитической геометрии и линейке. Вот тогда теория идеально ложится в мозг.

**Bro. fuzz**, 6 ноября 2011 в 20:50 # ↑

+5

После геймдева первый курс любого технического вуза заходит с восторгом и чистым понимаем. Матан + анализ после графики, физика (теор. мех) после копания с любой физической либой. Благодаря геймдеву был отличником на первых курсах.

**namespace**, 6 ноября 2011 в 21:15 #

0

Спасибо. То что надо. Как раз надумал игру писать на OpenGL и литературу искал. А тут такое нашел)

**MAXHO**, 6 ноября 2011 в 21:29 # ↑

+3

Внимательно прочитайте предыдущие комментарии. Это основа для 9-10 классников, а не серьезное пособие. Мне оно пригодится как учителю, а вот разработчику реальной игры — вряд ли.

**dmitriid**, 7 ноября 2011 в 01:44 # ↑

0

Оно может помочь хотя бы вспомнить, как это все называется, если универ был лет нцать тому назад :)

**MAXHO**, 6 ноября 2011 в 21:32 #

+4

Спасибо за перевод. Американцы очень хорошие популяризаторы. Первая часть статьи вообще идеально ложится для школьного урока.

**ComodoHacker**, 6 ноября 2011 в 22:31 #

-1

Вот это я понимаю, Хабр. Хотя и перевод. Самоубийцы отдыхают :)

**vladvoron**, 7 ноября 2011 в 10:50 #

0

Да уж... Когда в школе и в универе пичкали нас всем этим, помню, постоянно находились те, кто ворчал «Ну нафик это вообще нужно? Ну как это мне в жизни пригодится?». Но не нашлось ни одного преподавателя, который бы сказал «А вот для чего, например!». :)

**HomoLuden**, 7 ноября 2011 в 10:57 # ↑

+1

Тот кто так ворчит, скорее всего, и не будет этим пользоваться, а универ нужен «ради корочки». Тот кто жадно поглащает эти знания, сможет в итоге превратить их и в \$, и в интерес.

**GreatRash**, 7 ноября 2011 в 11:18 #

+6

Вы как хотите, а мне полезно. Дашь больше таких статей! А то раздел забит сплошными историями успеха...

**HomoLuden**, 7 ноября 2011 в 11:42 # ↑

-1

> Дашь больше таких статей!

Согласен... но только не скопипащенных с одного источника и проверенных на практике. А то баги копируются, заблуждения насаиваются...

НЛО прилетело и опубликовало эту надпись здесь

НЛО прилетело и опубликовало эту надпись здесь

**vovkasolovev**, 7 ноября 2011 в 13:06 #

-1

А по прикладной тригонометрии шпаргалка есть?


А то иногда надо запрограммировать вращение какое-то, и начинаешь с нуля все функции выводить.

**moadib**, 7 ноября 2011 в 14:30 # ↑


-1

Все вещи шпаргалками не охватить.


К примеру, если захотите найти нормаль в точке кривой, заданной функцией, понадобится вспоминать производные :)

- 


Acristi, 7 ноября 2011 в 13:08 #

-1
- Спасибо за перевод, интересно. ХОть и пишут что фигня, и все в школе изучается, а спустя цать лет самое оно вспомнить основы и найти в комментах направления для дальнейшего копания :)
- 


ikirin, 7 ноября 2011 в 16:41 #

-1
- Спасибо, отличный материал!
- 

shaman4d, 7 ноября 2011 в 16:47 #

-1
- Спасибо.
- 

egormerkushev, 7 июня 2014 в 10:41 (комментарий был изменён) #

0
- Спасибо! С помощью вашей статьи и [вот этого обсуждения](#) сделал определение попадания во вращающийся sprite в cocos2d.
- 

Calc, 16 октября 2014 в 03:55 #

0
- $$-1*(0.66,0.75) + -1*(-0.75, 0.66) = (0.1, -1.4)$$

по всем показателям должно получиться (-1.4, 0.1)

Только зарегистрированные пользователи могут оставлять комментарии. Войдите, пожалуйста.

- Разработчик PHP

Android разработчик

Тестировщик

Менеджер проектов / Project manager

Ведущий разработчик Ruby/ Rails

Инженер по автоматизированному тестированию

Ведущий инженер-программист в группу разработки сис...

Тестировщик ПО

Oracle DBA в отдел разработки Naumen Service Desk

Главный PHP-разработчик

все вакансии
- Уменьшить вес фото без потери качества

Сайт на Wordpress

Поправить PSD mock-up книги

Доработать UI/UX для приложения на Windows Phone

Доработка сайта: расширить функционал или написать ...

Создание программы для сканера отпечатка пальца

Парсер similarweb

Разработка доски объявлений

Консультация по ElasticSearch

Конвертнуть вектор Photoshop Cs 5 -> SVG и нарезать д...

все заказы