Лекция 1

Формат названий лабораторных: Лабораторная работа № 1. Фамилия Имя Отчество. Компьютерная графика.doc

Краткая история развития компьютерной графики

Компьютерная графика - это наука, которая изучает методы преобразования трехмерных объектов в двумерные изображения. Исторически появление компьютерной графики связано с пакетом Сазерленда в 1962 году. Сазерленд написал программу Sketchpad, которая позволяла изображать на экране простейшие трехмерные объекты. В 60-е годы появились алгоритмы формирования теней и удаления невидимых поверхностей. В 70-е годы появились рабочие станции, которые объединили: технологии ввода-вывода, технологии связи и технологии работы с графическими объектами. В это время появляется трехмерная графика. В 80-е годы появились технологии мультимедиа, т.е., к обработке графики добавился звук и видеоизображение. С 90-х годов по наши дни - развитие реалистичной графики, имитирующей реальный мир. 2018-2019 гг. - Использование технологии трассировки лучей в режиме реального времени. В наше время активно развиваются технологии виртуальной реальности и дополненной реальности.

Технологии компьютерной графики сочетают в себе математические методы (геометрия и мнимая алгебра) работы с объектами и методы аппаратного ускорения.

Аппаратные способы реализации алгоритмов компьютерной графики.

Растровый способ Для растровых изображений характерен способ его хранения в виде матрицы цветных точек (пикселей) Растровые методы включают в себя целочисленные методы построения линий, закраски многоугольников и эллипсов.

Трассировка лучей

Виды компьютерной графики (по способу хранения изображения)

Растровая графика Изображение хранится в виде матрицы, цвет каждой ячейки кодируется тремя байтами. Хранится в файлах BMP (bitmap image); или со сжатием - например, \*.jpeg

Векторная графика Векторные форматы используются в 3д-моделировании, CAD итд. В векторном методе изображение представляется в виде совокупности отрезков, дуг итд. В общем случае вектор - это набор данных о каком-либо объекте. Векторные изображения хорошо масштабируются, в отличие от растровых, но практически не позволяют получать изображения фотографического качества. Векторные изображения занимают в памяти объем, пропорциональный количеству примитивов, объем такого изображения в байтах не зависит напрямую от размера самого изображения.

Фрактальная графика Изображение строится по уравнениям, поэтому хранятся только формулы. Можно эмулировать образы живой природы, воды, ткани, дерева, текстур итд.

Лекция 2

Основные графические объекты Qt

Перо (QPen) - задает стиль рисования контура фигур. Свойства: цвет, толщина линии, тип линии.

Кисти (QBrush) - определяет стиль заполнения фигуры, цвет заливки и тип штриховки.

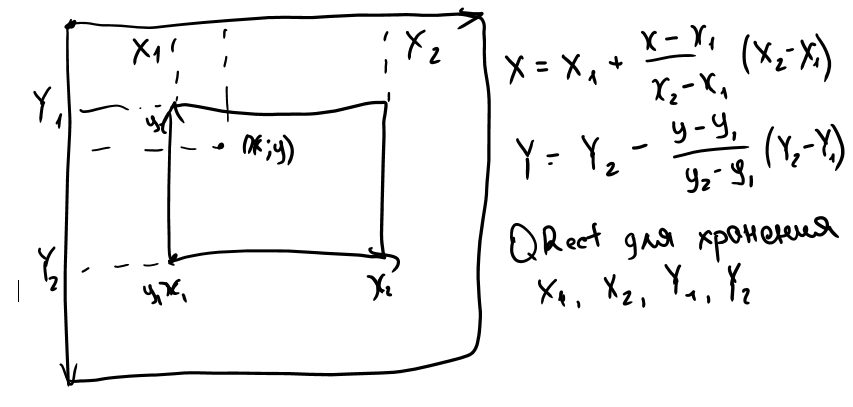
Цвет - чаще всего задается в формате RGB или RGBA. Каждый из каналов задается числом от 0 до 255 (для альфа-канала 255 - непрозрачная фигура). Также можно задавать в виде HSV (Hue, Saturation, Value). H - 0-359; S, V - 0-100. H - основной тон, S - насыщенность, чем больше - тем чище цвет. Модель HSV была создана одним из основателей PIXAR.

Для удобства работы с точками и прямоугольниками в qt предусмотрены QPoint, QPointF, QRect, QRectF. По сути класс QPoint хранит только координаты точки в целочисленном виде, а QPointf - в вещественном виде. QRect - то же самое, но с прямоугольником на плоскости. top(), left(), bottom(), right() возвращают расстояние от границ экрана, width(), height() - габариты прямоугольника

Для рисования используется класс QPainter, содержащий методы для рисования примитивов, инициализации кистей, перьев, цветов и т. д.

Экранная и мировая система координат

При работе с графическими приложениями обычно используют две системы координат:



мировую систему координат.

экранную систему координат - систему координат окна приложения.

Единицами измерения здесь является номер пикселя.

ТУТ КАРТИНКА НО Я НЕ ХОЧУ ЕЕ РИСОВАТЬ

там в общем прямоугольник, слева вверху 0, 0. внутри еще один прямоугольник в котором нарисован график функции суть в том, что график функции выражен в мировой системе координат, а в общем прямоугольнике используется экранная система координат.

Мировая система координат - это система координат, связанная с решаемой задачей; единицей измерения в ней может быть, к примеру, метр.

Чтобы перевести мировые координаты в экранные, надо $X=X\_1+\frac{x-x\_1}{x\_2-x\_1}(X\_2-X\_1)$; $Y=Y\_2-\frac{y-y\_1}{y\_2-y\_1}(Y\_2-Y\_1)$

Внутри Qt удобно хранить координаты с помощью класса QRect.

Лекция 3

Предпочтительный шаг разметки - это расстояние между двумя соседними разметками H (в пикселях).

H = 100 пикселей $h = 110^n, 210^n,5\*10^n$

Действительный шаг в общем случае не равен H, но выбираться он будет таким образом, чтобы при пересчете в мировые координаты полученный шаг был наиболее близок к одному из доступных значений h.

Пускай $H = 150, x\in [50, 200], w = 1000$. Тогда $h = \frac{H}{W}(x\_2-x\_1) = 22,5$. К этому значению ближе всего $h=210^1 = 20, m = 2, n = 1$. Тогда $H' = 20 \* \frac{W}{x\_2-x\_1} = 20000/150 = 133\frac{1}{3}$ - действительный шаг разметки. $x'\_1 = (\lfloor\frac{x\_1}{h}\rfloor+1)h = (\lfloor\frac{30}{20}\rfloor+1)\*20 = 60$. Разметка будет 60, 80, 100, 120, …

Можно приравнять возможные значения h к h\* и решить три уравнения с неизвестными n\_1, n\_2, n\_5 и выбрать наименьшее значение из следующих шести значений: $h^\* - 10^{\lfloor n\_1\rfloor}, 10^{\lceil n\_1\rceil} - h^, h^ - 2 \* 10^{\lfloor n\_1\rfloor}, 2 \* 10^{\lceil n\_1\rceil} - h^, h^ - 5 \* 10^{\lfloor n\_1\rfloor}, 5 \* 10^{\lceil n\_1\rceil} - h^\*$

Для вычисления шагов разметки необходима отдельная функция.

step(double &k, double \* H, int &m, int &n) вход $k = \frac{W}{b-a}$

выход

$k = h, k = m \* 10^2$

k - действительный шаг разметки в мировой системе координат, H - действительный шаг разметки в мировой системе координат.

Все вычисления лучше производить с использованием вещественных чисел, чтобы не накапливались ошибки. Вещественные числа нужно округлять, когда риски будут закрашены на графике. Параметр n может помочь при округлении значения разметок. Если n < 0, то нужно оставить -n знаков после запятой. QString("%1").arg(s, 0, 'f', -n);

Табулирование функций

При рисовании графика количество линий на нем должно быть прямо пропорционально W.

dx=(b-a)/W; x = a; QPoint P1, P2; converter; px = converter.X(x); PY = converter.Y(f(x));

for (int j = 1; j < W, j++){

x += dx;

P2X = converter.X(x);

P2Y=converter.Y(y);

painter.drawline(PX, PY, P2X, P2Y);

PX = P2X, PY = P2Y;

}

Столько линий, сколько содержится в одной строке по оси X (?). Для плавности можно увеличить в 2-4 раза.

Лекция 4

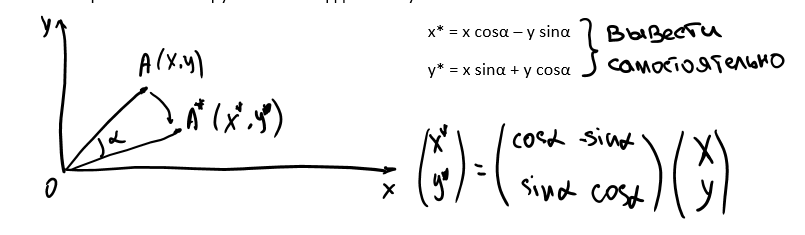
Аффинные преобразования на плоскости

Афинные преобразования упрощают выполнения геометрических операций избавляя от необходимости выводить геометрические формулы для расчета сложных движений в пространстве.

В компьютерной графике рассматривают 4 аффинных преобразования:

Поворот точки $A(x,y)$ вокруг начала координат на угол $\alpha$.

@todo График 4\_1



ВАЖНО. УМЕТЬ ВЫВОДИТЬ ФОРМУЛУ.

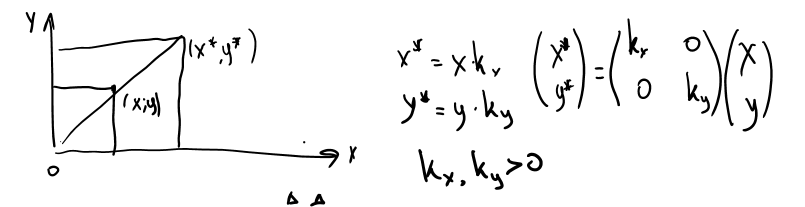
$$ x^\* = x\cos{\alpha}-y\sin{\alpha}$$

$$y^\* =x\sin{\alpha}y\*cos{\alpha}$$

$$ \left( \begin{array}{ccc} x^\y^ \end{array} \right) = \left( \begin{array}{ccc} \cos{\alpha} ,-\sin{\alpha}\\sin{\alpha}, \cos{\alpha} \end{array} \right) \left( \begin{array}{ccc} x \ y \end{array} \right) $$

Если $\alpha = 0$ матрица становится единичной.

Маштабирование относительно начала координат

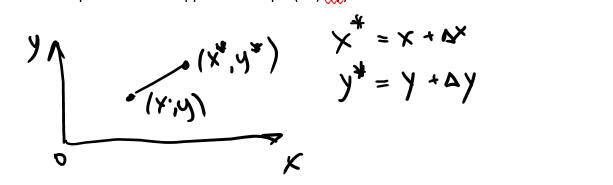


@todo График 4\_2

$$x^=xk\_x$$

$$y^\* = y \*k\_y$$

Перенос точки вдоль вектора $(\Delta x,\Delta y)$

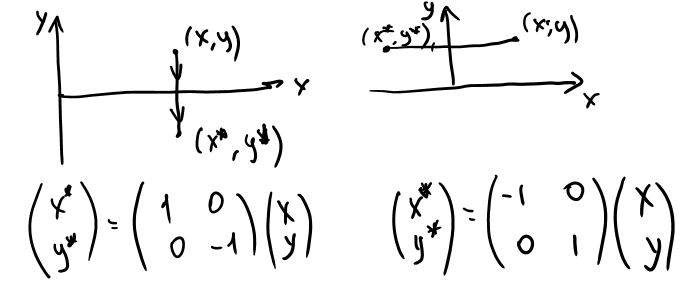


@todo График 4\_3

$$x^\*=x+\Delta x$$

$$y^\* = y + \Delta y$$

Отражение относительно оси абцисс и относительно оси ординат.



@todo График 4\_4

$$ \left( \begin{array}{ccc} x^\y^ \end{array} \right) = \left( \begin{array}{ccc} 1,0\0,-1 \end{array} \right) \left( \begin{array}{ccc} x \ y \end{array} \right) $$

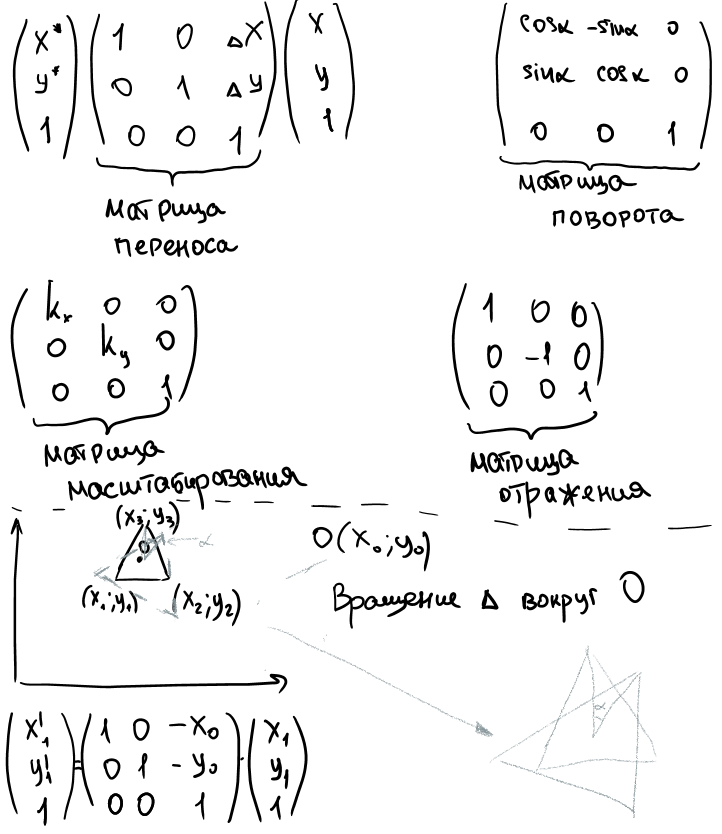
$$ \left( \begin{array}{ccc} x^\y^ \end{array} \right) = \left( \begin{array}{ccc} -1,0\0,1 \end{array} \right) \left( \begin{array}{ccc} x \ y \end{array} \right) $$

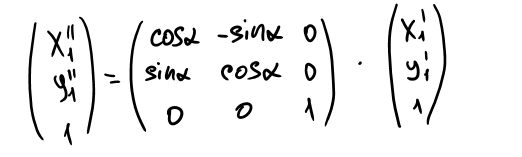
Операция переноса пока не выражена в матричном виде, данную операцию невозможно записать с использованием матриц размером $22$ , по этой причине переходят к матрицам размера $33$.

Однородными координатами точки $(x,y)$ называют тройку чисел $(ac)$ или $(x\_1:x\_2:k)$ которые связаны с исходными декартовыми координатами следующими соотношениями: $x = \frac{x\_r}{k}$; $ y = \frac{x\_2}{k}$

Использование однородных координат охватить матричными операциями все 4 аффинных преобразования.

Переход к однородным координатам сокращает количество операций деления в геометрических преобразованиях, что выгодно снижает вычислительные затраты. Операция деления заменяется умножением а последнее выполняется процессором гораздо быстрее.





Операция переноса в однородных координатах.

$$ \left( \begin{array}{ccc} x^\y^\1 \end{array} \right) = \left( \begin{array}{ccc} 1,0,\Delta x\0,1,\Delta y\0,0,1 \end{array} \right) \left( \begin{array}{ccc} x \ y\1 \end{array} \right) $$

Матрица поворота

$$ \left( \begin{array}{ccc} x^\y^\1 \end{array} \right) = \left( \begin{array}{ccc} \cos{\alpha},-\sin{\alpha},0\\sin{\alpha},\cos{\alpha},0\0,0,1 \end{array} \right) \left( \begin{array}{ccc} x \ y\1 \end{array} \right) $$

Матрица масштабирования

$$ \left( \begin{array}{ccc} x^\y^\1 \end{array} \right) = \left( \begin{array}{ccc} k\_x,0,0\0,k\_y,0\0,0,1 \end{array} \right) \left( \begin{array}{ccc} x \ y\1 \end{array} \right) $$

Матрица масштабирования

$$ \left( \begin{array}{ccc} x^\y^\1 \end{array} \right) = \left( \begin{array}{ccc} 1,0,0\0,-1,0\0,0,1 \end{array} \right) \left( \begin{array}{ccc} x \ y\1 \end{array} \right) $$

Чтобы повернуть треугольник нужно выполнить преобразования:

Перенос всех точек вдоль вектора.

Поворот всех точек треугольника на угол $\alpha$

Перенос всех точек вдоль вектора $()$

class Matrix3x3{

static Matrix3x3

rotate(float angle){

Matrix3x3 M;

M.m[0][0] = ...

return M;

}

}

Matrix3x3 R = Matrix3x3::rotate(30);

// Все матрицы:

scale(float kx, float ky);

translate(float dx, float dy);

reflectionX();

reflectionY();

Операции над матрицами размера 3x3 лучше реализовать без циклов для большего быстродействия.

Matrix3x3 R, M1, M2;

QVector3D V1[16], v2[16];

...

// Здесь лучше умножить матрицы, а потом уже формировать вектор

Matrix3x3 D = M1\*R\*M2;

for (int i = 0; i<16; i++){

V2[i] = P\*V[i];

}

При выполнении аффинных преобразований необходимо по возможности предварительно выполнить умножение матриц.

Аффинные преобразования в пространстве.

Введем однородные координаты в трехмерном пространстве. Заменим координатную тройку x, y, z четверкой чисел (hx, hy, hz, k). Однородные координаты связаны с обычными следующими соотношениями: $x=\frac{hx}{k}$, $y=\frac{hy}{k}$, $z=\frac{hz}{k}$. $k\ne 0$. Данный переход дает возможность воспользоваться матричной записью в более сложных трехмерных задачах. Любое аффинное преобразование в трехмерном пространстве может быть представлено в виде комбинации вращения, растяжения и переносов. Матрицы этих преобразований:

Матрица вращения, или поворота вокруг оси абсцисс $$R\_x = \left( \begin{array}{ccc} 1,0,0,0\0,\cos{\alpha},-\sin{\alpha}, 0\0,\sin{\alpha},\cos{\alpha},0\0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица вращения вокруг оси ординат $$R\_y = \left( \begin{array}{ccc} \cos{\beta},0,-\sin{\beta},0\ 0,1,0,0\ \sin{\beta},0,\cos{\beta},0\0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица вращения вокруг оси аппликат $$R\_z = \left( \begin{array}{ccc} \cos{\gamma},\sin{\gamma},0,0\-\sin{\gamma},\cos{\gamma},0, 0\0,0,1,0\0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица растяжения (сжатия) $$D = \left( \begin{array}{ccc} k\_x,0,0,0\ 0,k\_y,0,0\ 0,0,k\_z,0\ 0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица переноса

$$D = \left( \begin{array}{ccc} 1,0,0,d\_x\ 0,1,0,d\_y\ 0,0,1,d\_z\ 0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица отражения (Плоскость $O\_{xy}$) $$R\_{xy} = \left( \begin{array}{ccc} 1,0,0,0\ 0,1,0,0\ 0,0,-1,0\ 0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

Матрица отражения (Плоскость $O\_{xz}$) $$R\_{xz} = \left( \begin{array}{ccc} 1,0,0,0\ 0,-1,0,0\ 0,0,1,0\ 0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right)$$

QMatrix4x4::QMatrix4x4(const float \*v){

float m[16] = {

cos(a), -sin(a), 0, 0,

sin(a), cos(a), 0, 0,

0, 0, 1, 0,

0, 0, 0, 1

};

QMatrix R(m);

float \*data(); //Возвращает указатель на свободный массив, в котором записаны координаты

double determinant(); //возвращает определитель матрицы

void fill(float a); //Заполняет матрицу значениями a

void rotate(float angle, const QVector3D &vector);

// Умножает текущую матрицу на матрицу поворота. angle - угол поворота, второй параметр - вектор, вокруг которого поворачиваются точки

QMatrix4x4 A;

A.setToIdentity();

A.rotate(30, QVector3D(1, 0, 0));

void QMatrix4x4::scale(float kx, float ky, float kz); // Функция scale умножает текущую матрицу на матрицу масштабирования. kx, ky, kz - коэффициенты растяжения вдоль координатных осей.

void QMatrix4x4::translate(const QVector3D& vector); //умножает текущую матрицу на матрицу переноса

void QMatrix4x4::translate(float dx, float dy, float dz); // (dx, dy, dz) - вектор, вдоль которого выполняется перенос

// Для выполнения отражения можно использовать функцию scale, передавая ей отрицательные значения

void QMatrix4x4::transposed() const; // Выполняет транспонирование матрицы относительно главной диагонали.

void QMatrix4x4::setToIdentity(); // инициализирует текущую матрицу как единичную.

void QMatrix4x4:inverted(bool \*is\_inverted = nullptr) const; // Возвращает обратную матрицу. Если матрицу нельзя обратить, возвращается единичная матрица (определитель равен 0)

// Если аргумент is\_inverted не равен 0, то в него записывается результат вычисления: истина, если обратная матрица вычислена, иначе ложь.

}

Повернем фигуру относительно своего центра вокруг осей $\alpha\_x, \alpha\_y$. Пускай у фигуры есть некоторое количество вершин $v\_0..v\_{n-1}$. Во-первых, надо вычислить центр фигуры.

$$x\_0 = \frac{\sum^{n=1}{i=0}V{ix}}{n}, y\_0 = \frac{\sum^{n=1}{i=0}V{iy}}{n}, z\_0 = \frac{\sum^{n=1}{i=0}V{iz}}{n}$$

Точка $x\_0, y\_0, z\_0$ называется центром тяжести фигуры.

QMatrix4x4 M; // Единичная матрица

M.translate(-x0, -y0, -z0);

M.rotate(ax, 1, 0, 0);

M.rotate(ay, 0, 1, 0);

M.translate(x0, y0, z0);

Изменим данный фрагмент для вывода фигуры на экран.

Сначала узнаем размеры фигуры. Для этого вычислим минимальные и максимальные значения координат x, y, z в массиве вершин $v\_i$.

$r\_x$ - длина фигуры, $r\_y$ - ширина, $r\_z$ - высота.

Выбор коэффициента масштабирования:

$r = max(r\_x, r\_y, r\_z)$

$k = \frac{min(H, W)}{2r}$

QMatrix4x4 M; // Единичная матрица

M.translate(-x0, -y0, -z0);

M.rotate(ax, 1, 0, 0);

M.rotate(ay, 0, 1, 0);

M.translate(x0, y0, z0);

M.scale(k, -k, k); //чтобы походя перевернуть фигуру для правильного положения осей

M.translate(W/2, H/2, 0);

for (int i = 0, i<=n, i++){

W[i] = M\*V[i];

}

// В данном цикле все координаты вершин подвергаются афинному преобразованию. В результате будет выведена трехмерная фигура по центру экрана.

// Для вывода изображения нужно использовать компоненты x, y массива W. Компонента z при выводе игнорируется

Формат хранения.

Геометрическую модель лучше хранить в виде массива многоугольников (QPolygon). Для вывода изображения можно пока использовать только функцию drawline, т.е., не заливать полигоны, а выводить только каркас. Можно задать координаты вершин модели таким образом, чтобы $-1 \le v\_{ix}, v\_{iy}, v\_{iz} \le 1$. Это упростит выполнение аффинных преобразований. Можно умножать либо матрицы на вектора, либо наоборот - вектора на матрицы. В одном из случаев аффинные преобразования записываются в прямом порядке, в другом случае - в обратном порядке. При этом матрицы в обоих случаях будут взаимно транспонированными.

Проектирование

Изображение объектов на экранной плоскости связано с еще одной геометрической операцией - проектирование при помощи пучка прямых. В компьютерной графике используется несколько различных видов проектирования. Чаще всего используется параллельное и центральное (перспективное) проектирование.

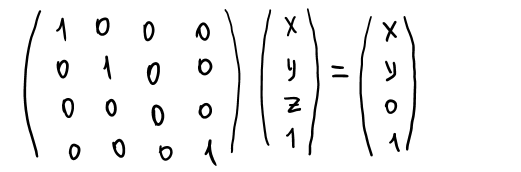
Лекция 5

Ортографичекое проектирование

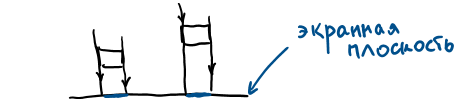
Для получения проекции объекта на экранную плоскость необходимо провести через каждую его точку прямую из заданного проектирующего пучка и затем найти координаты точки пересечения этой прямой с плоскостью изображения. В случае центрального проектирования все прямые исходят из одной точки - центра схода. При параллельном проектировании центр (точка схода) считается лежащим на бесконечности.

@todo КГ рисунок 5.1

Самый простой вид проектирования - ортографическое проектирование. В данном случае компонента $z$ объекта обнуляется, а фигура получается соединением линий на плоскости $0xy$ (т.е. используются только компоненты $x$ и $y$). Чтобы обнулить компоненту $z$ нужно умножить однородные координаты на матрицу ортографического проектирования.

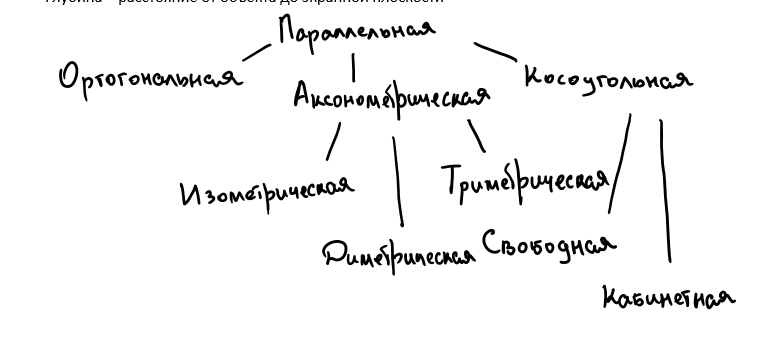


@todo КГ рисунок 5.2



При ортографическом проектировании невозможно оценить глубину объекта т.е. наблюдатель не понимает, насколько далеко расположение объекты за картинной(экранной) плоскостью.

Глубина - расстояние от объекта до экранной плоскости (координата $z$ объекта).



Параллельные проекции:

Ортографическая

Аксонометрическая

Изометрическая (все 3 угла между нормалью картинной плоскости и координатными осями равны)

Диметрическая (2 угла равны)

Триметрическая (нормальный вектор картинной плоскости образует с ордами координатных осей попарно различные углы)

Косоугольная

Свободная

Кабинетная

Каждый из 3-х видов этой проекции получается комбинацией поворотов за которой следует параллельное проектирование.

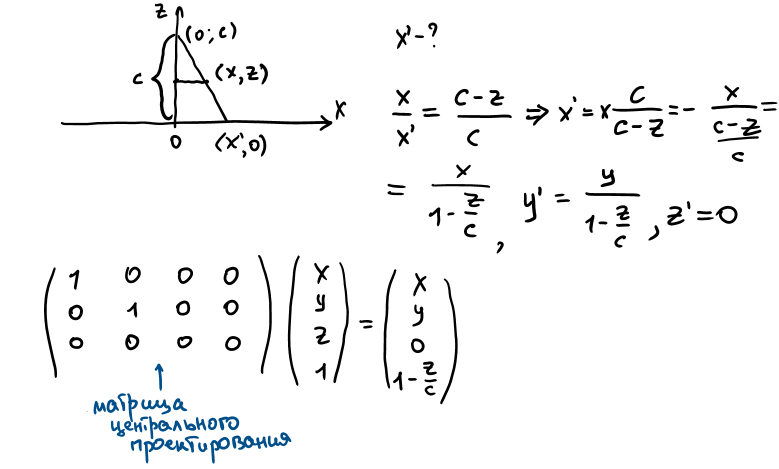
Для получения данных проекций необходимо сначала умножить однородные координаты на матрицу поворота, а затем на матрицу ортографического проектирования.

Центральное проектирование

Для того чтобы устранить недостатки ортографического проектирования и придать изображению глубину используют перспективные проекции.

Главный параметр центральной проекции - глубина точки схода.

@todo КГ рисунок 5.3 $$ \frac{x}{x'}=\frac{c-z}{c} $$



Запишим аналогичным образом выражение для нахождения $y'$ $$ y' = \frac{y}{1-\frac{z}{c}} $$ Матрица центрального проектирования

$$\left( \begin{array}{ccc} 1,0,0,0\ 0,1,0,0\ 0,0,0,0\ 0,0,-\frac{1}{c},1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right) = \left( \begin{array}{cccc}x\y\0\1-\frac{z}{c} \end{array} \right)$$

При выполнении центрального проектирования 4-ая координата уже не равна 1, поэтому для вычисления обычных координат нужно обязательно поделить $x,y,z$ на 4-ую компоненту однородных координат, в противном случае фигура не изменит свой вид.

При центральном проектировании объект должен находится на оси $z$ для избежания перспективных искажений.

Косоугольное проектирование



Матрица косоугольного проектирования

$$\left( \begin{array}{ccc} 1,0,k,0\ 0,1,k,0\ 0,0,0,0\ 0,0,0,1 \end{array} \right) \left( \begin{array}{cccc} x\y\z\1 \end{array} \right) = \left( \begin{array}{cccc}x+zk\y+zk\0\1 \end{array} \right)$$ $$ \text{если } k = \cos{\frac{\pi}{4}} \text{ - свободная косоугольная проекция} $$

$$ \text{если }k = \frac{1}{2}\*\cos{\frac{\pi}{4}} \text{ - кабинетная косоугольная проекция} $$

Лекция 6

Алгоритмы сортировки по глубине

После выполнения операции проектирования и афинных преобразований требуется выполнить следующую операцию: Определить, какие грани находятся ближе к наблюдателю (к экранной, или картинной плоскости). Для этого существует несколько методов:

Метод z-буфера

Сортировка граней по $Z\_p$ - расстояние от экранной плоскости до центра многоугольника.

Полная сортировка и разделение граней.

Метод сортировки, использующий трассировку лучей

Метод z-буфера

В видеопамяти хранится матрица размером $N\_x x N\_y$ ($N\_x, N\_y$ соответствуют разрешению экрана).

В данной матрице хранится расстояние до ближайшей грани и ее цвет. Матрица заполняется следующим образом: каждому пикселю экранной плоскости (x, y) сопоставляется цвет и расстояние по направлению проектирования (его глубину). Изначально массив глубин инициализируется максимальными значениями. Для вывода на картинную плоскость произвольной грани она переводится в ее растровое представление и для каждого пикселя этой грани находится его глубина. В случае, если эта глубина меньше значения глубины, хранящейся в z-буфере, этот пиксель рисуется, и его глубина заносится в z-буфер.

Метод z-буфера аппаратно реализован в современных видеопроцессорах. Это основной метод для удаления невидимых поверхностей. Но z-буфер не может правильно сортировать прозрачные грани. Для этого используются алгоритмы Брезенхейма.

@todo заснул памагитя

Метод Z-буфера работает исключительно в пространстве картинной плоскости и не требует никакой предварительной обработки данных.

Если на сцене появляются прозрачные объекты, то нужно использовать другие методы сортировки.

Рассмотрим самый простой способ сортировки, основанный на разбиении грани. Основная идея сортировки следующая:

Для каждого полигона найдем координаты его центра $(x\_i^{(0)}, y\_i^{(0)}, z\_i^{(0)})$ и отсортируем все полигоны по значению $z\_i^{(0)}$.

В пятой лабораторной работе надо будет только отсортировать полигоны по этому значению. После всех преобразований, но до проектирования, надо вычислить все эти координаты, а затем отсортировать.

Алгоритм сортировки

Задать фигуру в виде выпуклых оболочек, которые состоят из граней, которые не пересекаются. Фигура не должна содержать слишком вытянутых полигонов.Если полигон слишком вытянутый, его нужно разбить на небольшие полигоны.

Перед выполнением проектирования нужно вычислить центры полигонов. Глубиной полигона можно считать значение z центра тяжести полигона: $z\_p = \frac{1}{n}\sum\_{i=1}^n z\_i$. $(\frac{1}{n}\sum\_{i=1}^n x\_i, \frac{1}{n}\sum\_{i=1}^n y\_i, \frac{1}{n}\sum\_{i=1}^n z\_i)$ - центр тяжести.

Отсортировать массив полигонов по значению $z\_p$. Для сортировки лучше использовать методы, имеющие сложность, близкую к mlog\_2 n (m - количество граней).

Полигоны отрисовываются в порядке от дальних к ближним функцией drawPolygon.

Программу лучше отлаживать поэтапно Для отладки удобно оставить два полигона и поворачивать их Нужно назначить каждому полигону свой цвет Все грани обязательно должны быть выпуклыми. При сортировке граней в программе требуется сохранить два набора координат:

Координаты исходной модели

Координаты преобразованной фигуры.

Сортировать грани нужно обязательно до операции проектирования, иначе координаты z обнулятся.

Лекция 7

Лабораторная работа № 5

Алгоритм удаления невидимых граней.

Цель работы:

Реализация простейшего алгоритма сортировки многоугольников по глубине.

Задания для выполнения к работе

Реализовать алгоритм сортировки выпуклых многоугольников по глубине $Z$, где $Z - $центр тяжести многоугольника (при помощи qStableSort).

На экран вывести только 1 проекцию модели (центральную или ортографическую). Пользователь должен иметь возможность поворачивать фигуру, с использованием мыши или клавиш, удалять и приближать фигуру.

Фигура должна состоять из набора небольших граней. Грани должны быть соизмеримы.

При выводе изображения предоставить возможность пользователю видеть грани прозрачными.

Алгоритм принадлежности точки выпуклому многоугольнику.

В компьютерной графике часто возникает задача - нужно выяснить находится ли точка внутри выпуклого многоугольника.

К примеру: пользователь наводит курсор мыши на объекты на экране. Нужно определить на какие объекты указывает курсор.

@todo pic and math comp\_graph lec7

Так как правее вектора $a\_4$ находится вектор $a\_0$ целесообразнее заменить импульс $i+1$ на импульс $(i+1)\text{ mod }n$ , где $n$ - кол-во вершин многоугольника.

Если все компоненты $z$ векторных произведений будут иметь одинаковый знак, значит точка O находится внутри многоугольника, иначе вне.

Вершины многоугольника должны быть обязательно упорядоченны по или против часовой стрелки в противном случае алгоритм будет работать неправильно.

Поворот объектов с помощью клавиши и(или) курсора мыши.

В конструкторе виджета инициализировать глобальную матрицу поворота в виде единичной матрицы.

void MainWindow::mouseMoveEvent(QMouseEvent \*m\_event){

double k=10;

QPoint dp = m\_event->pos() - mousePrevPos;

mousePrevPos = m\_event->pos();

R.rotate(dp.x()/k,QVector3D(0,1,0));

R.rotate(dp.y()/k,QVector3D(1,0,0));

}

При выполнении аффиных преобразований после переноса в начало координат нужно умножить текущую матрицу на матрицу $r$ или транспонированную матрицу $r$, в зависимости от порядка умножения матриц.

Современные 3D API

Современные видеоускорители содержат больше транзисторов, чем ЦП.

У каждой видеокарты есть свой ассемблер и специальные команды, через которые ей можно управлять. У каждого ускорителя есть свои особенности и программная последовательность для одного ускорителя не обязательно будет работать на другом, поэтому для упрощения программирования графического процессора существую 3D API. Выполняют следующие функции:

Взаимосвязь с ресурсами ЦП и памятью

Взаимосвязь с драйвером ГПУ

Взаимосвязь с операционной системой (Например, через winAPI)

Предоставление библиотечных функций для создания трехмерной сцены.

Обеспечение через функции библиотеки полного прохождения конвейера обработки изображения.

OpenGL (Open Graphics Library) - универсальное, аппаратно независимая библиотека, которая поддерживает разнообразные 3D объекты и конструкции, начиная с примитивов типа треугольника и заканчивая NURBS-поверхностями. Трехмерный OpenGL был создан летом 1992 года внутри корпорации Silicon Graphics. OpenGL - спецификация, а все детали ее реализации полностью возлагаются на производителей аппаратного обеспечения. Основной ЯП в OpenGL - c++. При использовании OpenGL нужно создавать программный код, учитывающий многие детали. От программиста требуется глубокое знание этапов визуализации сцены. Зато архитектура OpenGL дает возможность реализовывать нестандартные. В нем также есть механизм расширений, то есть в библиотеку можно добавлять функции, не реализованные в базовой версии API.

Начиная с версии 2.0 получил встроенное дополнение OpenGL Shader language. Шейдер - подпрограмма для обработки графического объекта (вершина, пиксель). Шейдерная подпрограмма выполняется параллельно для массива объектов.

OpenGL является кроссплатформенной библиотекой. Ядро не поддерживает никаких функций обработки окон, вывода на экран или устройств ввода. Для этого используются дополнительные библиотеки (например, библиотека GLUT).

Direct3D был разработан компанией Microsoft для расширения мультимедийных возможностей ОС Windows. Direct3D пользуется популярностью у разработчиков компьютерных игр, также имеет язык программирования шейдеров HLSL (High Level Shader Language). Direct3D может работать в двух режимах: Retained mode, Immediate mode. В режиме intermediate mode direct3d дает наибольшую производительность, т.к. работает с аппаратурой напрямую. Режим retained - это режим абстракции. Это менее производительный режим, но в нем легче программировать.

Лекция 9

Вершины в OpenGL

Положение вершины определяется заданием координат в 2-х,3-х,4-х(однородные координаты) мерном пространстве.

//точка

void glVertex/\*2 - плоскость,3 - пространство, 4 - однородные координаты\*//\*s - short \i - int \f -float \d - double\*/()

glVertex3f(1.0f,0,0)

//цвет

void glColor[3,4][b,s,i,f]()

//треугольник

glBegin(GL\_TRIANGLES);

glColor3f(0,0,1);

glVector2f(x1,y1);

glColor3f(0,1,0);

glVector2f(x2,y2);

glColor3f(1,0,0);

glVector2f(x3,y3);

glEnd();

В трехмерной графике может быть использовано правосторонее. Для создания освещения необходимо, сообщить системе OpenGL координаты векторов нормалей.

Лекция 10

Освещение в OpenGL

Существует несколько различных типов источников света, каждый из которых имеет свои свойства и поведение.

directional (направленный) - солнце

ambient (рассеянный) - освещение в пасмурную погоду

point (точечный) - лампочка

spot (прожектор) -

area (прямоугольный) - прямоугольный источник испускает лучи из прямоугольной области, а не из одной точки пространства.

Свет, выходящий с поверхности конкретной точки в сторону наблюдателя, представляет собой совокупность компонентов, связанных с материалом и цветом поверхности в данной точке. К таковым компонентам относятся: свет, пришедший с обратной стороны поверхности, то есть преломлленный свет (refracted), свет, рассеянный поверхностью (diffused), зеркально отраженный свет (reflected), блики (то есть, отраженный свет источника) (specular), собственное свечение поверхности (self\_illumination). В модели OpenGL свет, испускаемый источниками, состоит из трех компонентов:

Фоновый, или рассеянный свет (ambient light). Этот тип освещения моделирует свет, который отражается от других поверхностей и освещает всю сцену.

Рассеиваемый свет (diffused light). Этот свет распространяется в заданном направлении, сталкиваясь с поверхностью, он отражается равномерно во всех направлениях. Поэтому интенсивность достигшего глаза зрителя света не зависит от точки, с которой просматривается сцена, и местоположение зрителя можно не учитывать.

Отражаемый свет (specular light). Этот свет распространяется в заданном направлении. Сталкиваясь с поверхностью, он отражается строго в одном направлении, формируя блики, которые видны только при взгляде на объект под определенным углом. Отражаемый свет используется для создания бликов, которые появляются при освещении полированных поверхностей. glEnable(GL\_LIGHTNING) включает освещение. Самый простой тип освещения - фоновое освещение (рассеянное освещение). float ambient[] = {0.5, 0.5, 0.5, 1} glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, ambient) При такой организации освещения лучше отключить нулевую лампу glEnable(GL\_LIGHT0).

Установка точечных источников света float pos[] = {3, 0, -3, 1}; float color[] = {1, 1, 1, 1}; float spec[] = {1,1,1,1}; glEnable(GL\_LIGHT1); glLightfv(GL\_LIGHT1, GL\_SPECULAR, spec); glLightfv(GL\_LIGHT1, GL\_DIFFUSE, color); glLightfv(GL\_LIGHT1, GL\_POSITION, pos);

Тип источника света зависит от 4-го элемента массива pos. pos[3] = 0, 1: точечный или направленный. Если точечный, то это координаты, а если направленный - вектор, по которому направлены лучи света.

Материалы.

Материал может рассеивать, отражать и излучать свет. Свойства материала устанавливаются следующим образом: glMaterialfv(face, phame, params); face может быть: GL\_BACK GL\_FRONT GL\_FRONT\_AND\_BACK Второй параметр может принимать следующие значения: GL\_AMBIENT - рассеянный GL\_DIFFUSE - собственный GL\_SPECULAR - отраженный GL\_EMISSION - излучаемый GL\_SHININESS - степень отраженного света

float c1[] = {0, 1, 0, 1} float c2[] = {1, 0, 0, 1} gluQuadricObj \*q; q = glunew Quadric(); glMaterialfv(GL\_FRONT, GL\_DIFFUSE, c1); glMaterialfv(GL\_BACK, GL\_DIFFUSE, c2); gluCylinder(1, 0.5 2, 10, 10); gluDeleteQuadric(q);

Большие модели трехмерных объектов, состоящие из большого количества полигонов, обычно поделены на фрагменты, каждый из которых представляет собой отдельный массив полигонов. Фрагмент может сопровождаться информацией о свойствах материалов. Для модели предусмотрены файлы \*.obj, для материалов - \*.mtl. .obj - файлы содержат информацию о полигонах модели, то есть, информация в них представлена в виде набора полигонов, состоящих из вершин. Каждая вершина сопровождается информацией по ее координатам, координатах вектора нормали. Текстуры хранятся в отдельнах Файлы с расширением .mtl, где хранятся параметры материаов и ссылки на текстуры.

ПЕРЕНЕСТИ В АВС

Рассмотрим случай скалярного произведения векторов. a = x1x2 + y1y2 + z1z2; При последовательном исполнении порядок выполнения команд следующий:

t1 = x1x2 t2 = y1y2 t3 = z1\*z2 t4 = t1+t2 t5 = t4+t3

Для вычисления выражения потребовалось пять машинных циклов.

При параллельном исполнении порядок будет следующим: t1, t2, t3 - в одно и то же время t4 t5

Потребовалось три машинных цикла.

Л/р 8 Текстурирование Цель работы: создание текстур с помощью библиотеки OpenGL с использованием Qt Creator.

Текстурирование

При описании процесса обработки изображения в некоторых случаях необходимо выполнить текстурирование. Благодаря текстурированию, объекты получают реалистичность.

Текстура - это двумерное изображение, которое накладывается на полигон и изображает его поверхность. То, что нельзя смоделировать с помощью полигонов, можно нарисовать. Причем если на поверхности, которая моделируется текстурой, нет сильно выдающихся деталей, оно будет смотреться реалистично.

По аналогии с изображениями, которые состоят из пикселей, текстура состоит из текселей. Текстура накладывается строго по координатам, например, в opengl координаты текселей включают сопроводительную информацию о полигонах. Процесс сопоставления координат текстур алгоритмически выведен для объектов типа "сфера", "цилиндр", "конус" и многоугольник.

При работе с текстурами существует несколько проблем. У экрана есть свое разрешение и определенное количество пикселей, которые на нем можно отобразить в данной плоскости экрана. Но в соответствующей области области виртуального мира количество текселей может быть и больше, и меньше, чем на экране. Отсюда появляются проблемы отображения текстурированного объекта. Для их решения разработаны различные методы фильтрации:

Билинейная фильтрация. При данном способе цвет пикселя получается в результате интерполяции при соединении цветов четырех соседних текселей. Если объект расположен далеко от камеры, , его текстура почти не искажается. Когда объект находится недалеко от камеры, и текселей не хватает, интерполяция создает расплывчатое изображение в этой области.

Билинейная фильтрация хорошо работает только для полигонов, которые параллельны или почти параллельны экрану. Дело в том, что четыре соседних текселя образуют круг. Если плоскость телефона наклоняется, эти четыре текселя образуют эллипс., но интерполируются тексели по-прежнему по кругу.. От этого постоянно накапливаются небольшие ошибки. После определенного угла наклона ошибки становятся уже заметнее. Текстура фильтруется геометрически неправильно, а пользователь наблюдает сильные искажения.

Трилинейная фильтрация. Текстура сама по себе может искажаться фильтрацией. Для решения проблемы используют трилинейную фильтрацию и технологию, которая называется MIP Mapping. Для одной поверхности разработчик создает несколько копий текстуры с различным разрешением. Каждая следующая версия текстуры меньше или больше предыдущей в 4 раза. Когда камера удаляется от текстуры, последняя сменяется на текстуру с меньшим разрешением. а когда приближается - на текстуру с большим разрешением. Теперь текстура отображается без искажений, но в памяти нужно хранить несколько копий текстуры.

В трилинейной фильтрации цвет конкретного пикселя на экране определяется в результате интерполяции цветов пикселей двух соседних уровней (двух версий текстур). При этом переход между уровнями (текстурами) становится плавным.

Анизотропная фильтрация. Предыдущие способы фильтрации работают работают хорошо для таких текстур, которые накладываются на объекты, параллельные экрану. Если текстура сильно наклонена, фильтрация в одном направлении выполняется больше, чем в другом. Под термином "анизотропная фильтрация" подразумевается несколько различных и сложных технологий.

Анизотропная фильтрация оперирует не менее, чем 8-ью текселями, несколькими версиями текстуры с различным разрешением, , при этом используется модель заранее неопределенной формы. В результате убираются шумы и искажение объектов, а изображение становится более качественным. Анизотропная фильтрация работает с текселями как с эллипсами, и для расчета цвета одного пикселя обрабатывает до 32 текселей.

Качество текстуры при анизотропной фильтрации даже на дальней дистанции остается схожим с оригинальным. Анизотропная фильтрация, как и трилинейная, уменьшает неровность текстур. Но при использовании анизотропной фильтрации качество получается наилучшим. Версия текстуры с конкретным разрешением называется MIP-уровнем, набор текстур одной и той же поверхности, которые имеют различное разрешение, называют MIP-каскадом.