Компьютерная Графика

Базовые Знания

Информация

Игорь Таранцев

i.tarantsev@g.nsu.ru

+7 913 918 2186

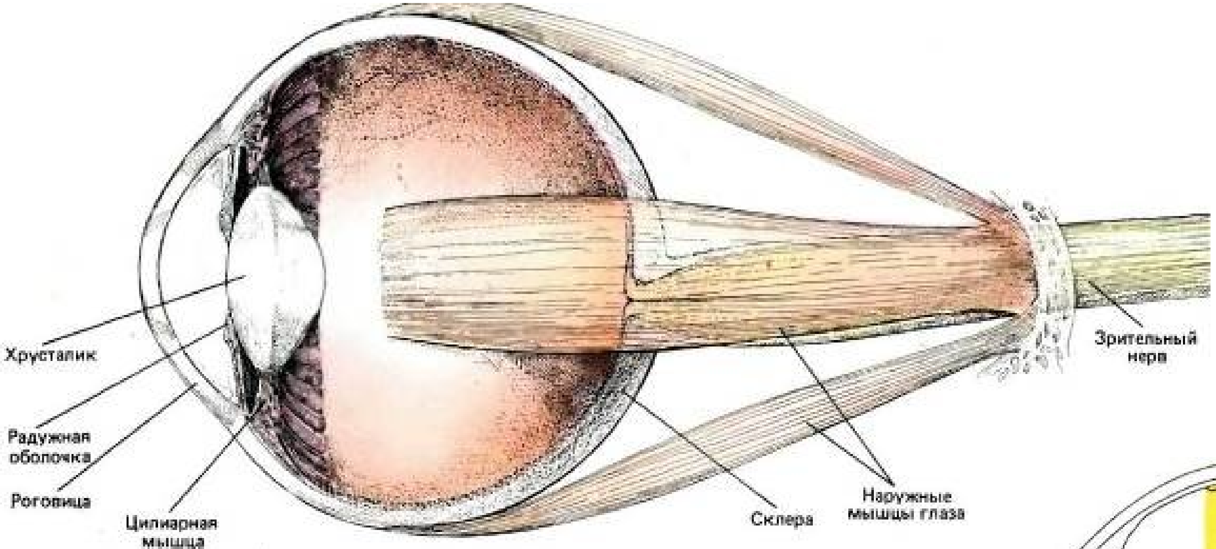
Курс занимает 1 семестр

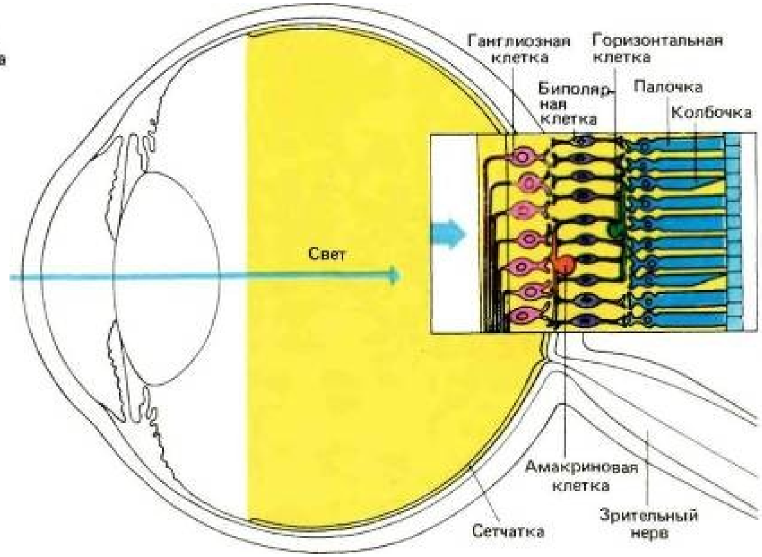
Оценка ставится в зачётку в колонку Экзамены

Оценка состоит из двух частей:

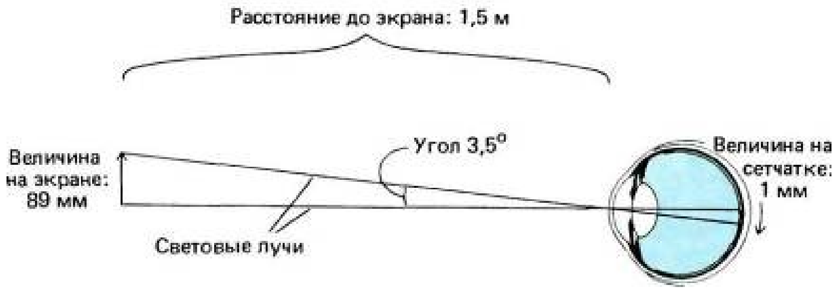
- 2D-графика - билеты с вопросами
- 3D-графика - задачи и финальный проект

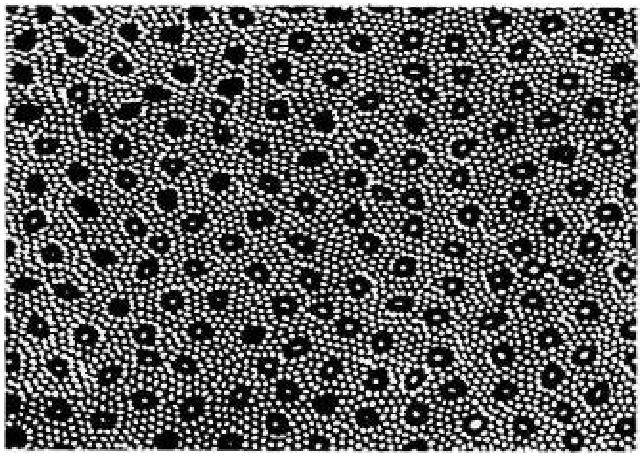
Строение глаза





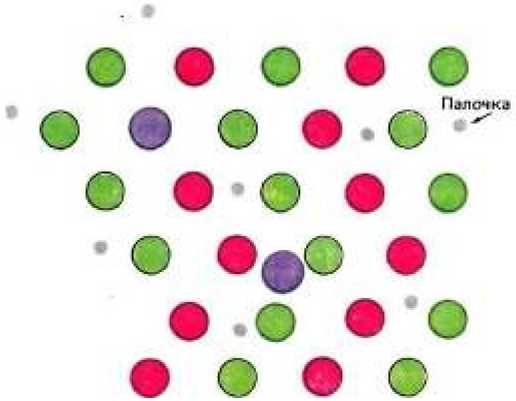
Угловое разрешение





0,5 угловых минуты в центре

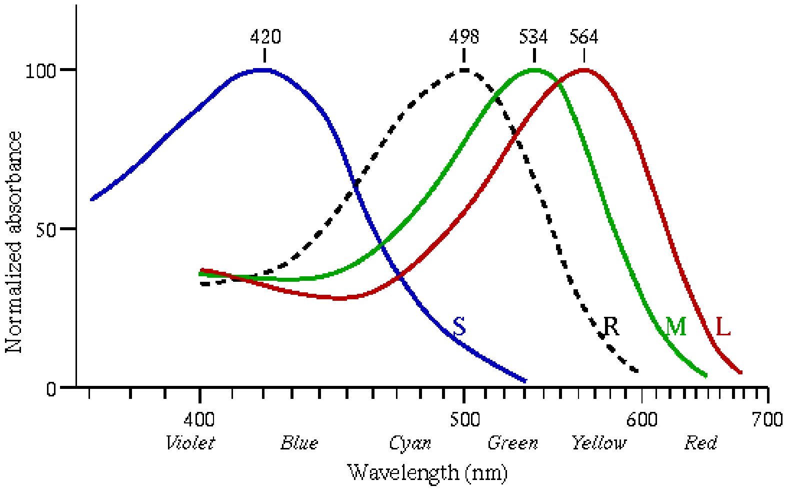
Белые точки – пигмент
Темные пятна – колбочки

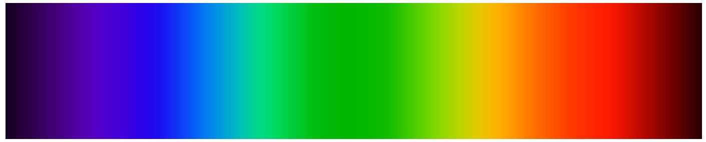


Угловое разрешение по яркости
в разы выше разрешения по цвету

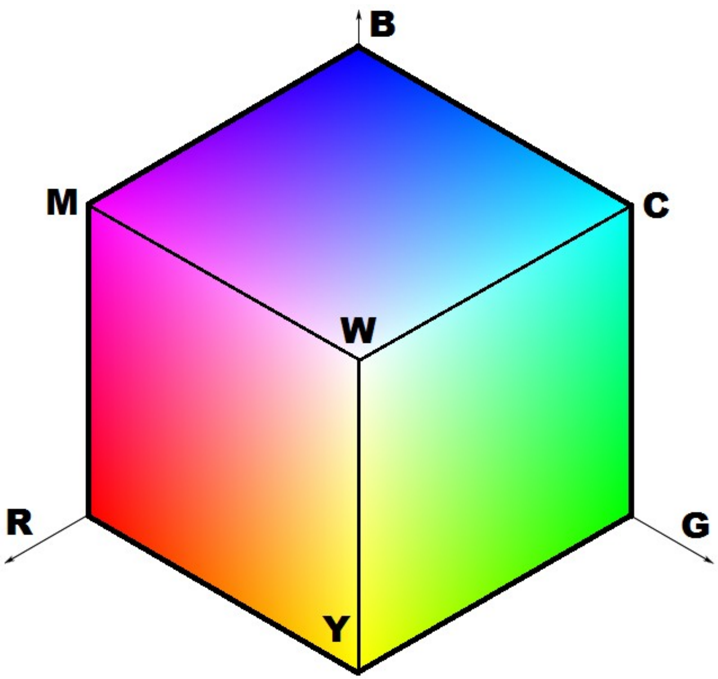
Цветовые модели

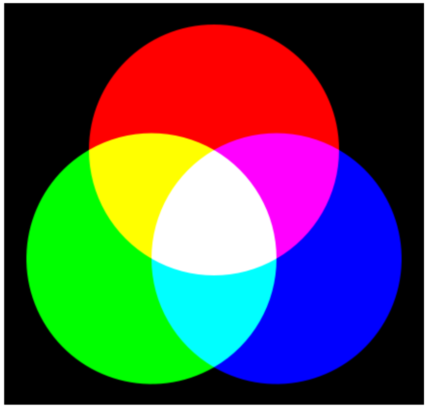
- Аддитивная
  RGB
- Субтрактивная
  CMY/CMYK
- “Человеческая”
  HSV/HLS
- Телевизионная
  YCbCr/YUV
- “Научная”
  L*a*b



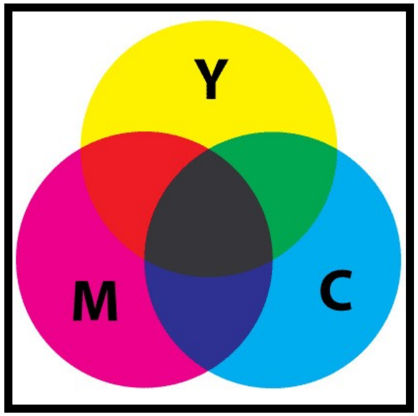


Цветовые модели

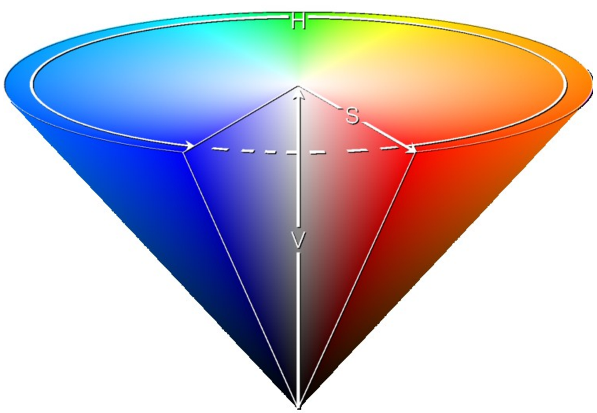




RGB



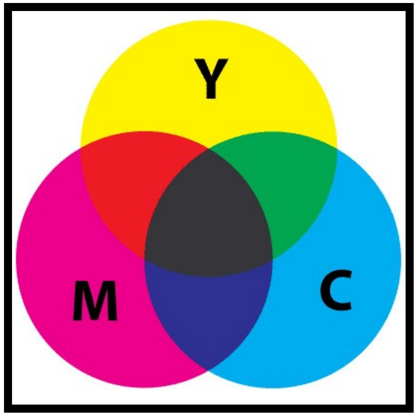
CMY



HSV

Цветовые модели

C,M,Y → C',M',Y',K
K(blacK) = min(C,M,Y)
C' = C – K
M' = M – K
Y' = Y – K



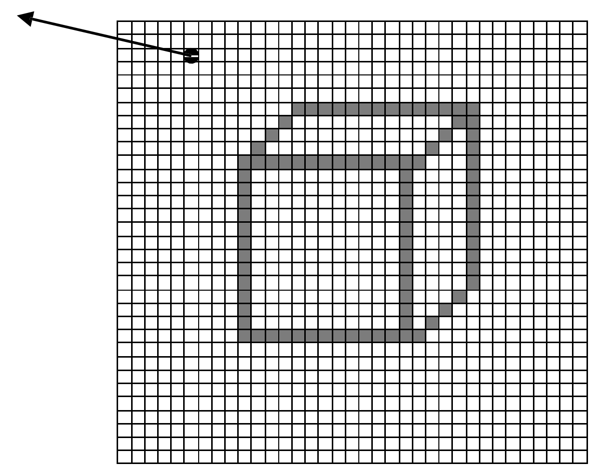
CMYK

Черная краска – дешевая и «точная»
Смесь трех красок в равной пропорции дает грязно-серый, а не черный цвет

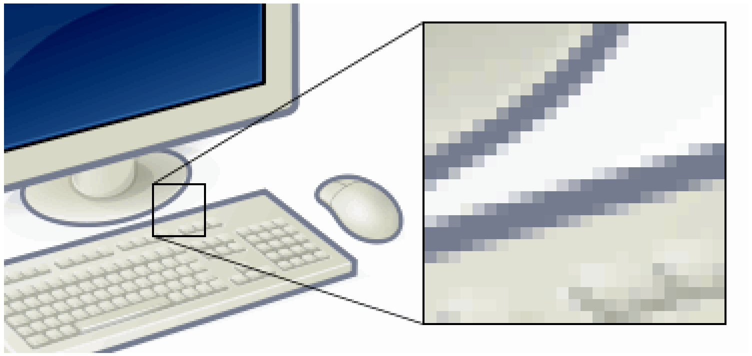
Дисплей и буфер кадра

Изображение – это растр = 2D массив пикселей

`int FB[n][m] – frame buffer  FB[5][2]`



Пиксель (pixel, picture element) мельчайший элемент изображения



Типы пикселей:

- Черно-белый – 1 бит/пиксель
- Монохромные – 2, 4, 8, 12 бит
- Цветные – 2,4,8,15,16,24,32,48,96 б/п
- Супер – 96 б/п

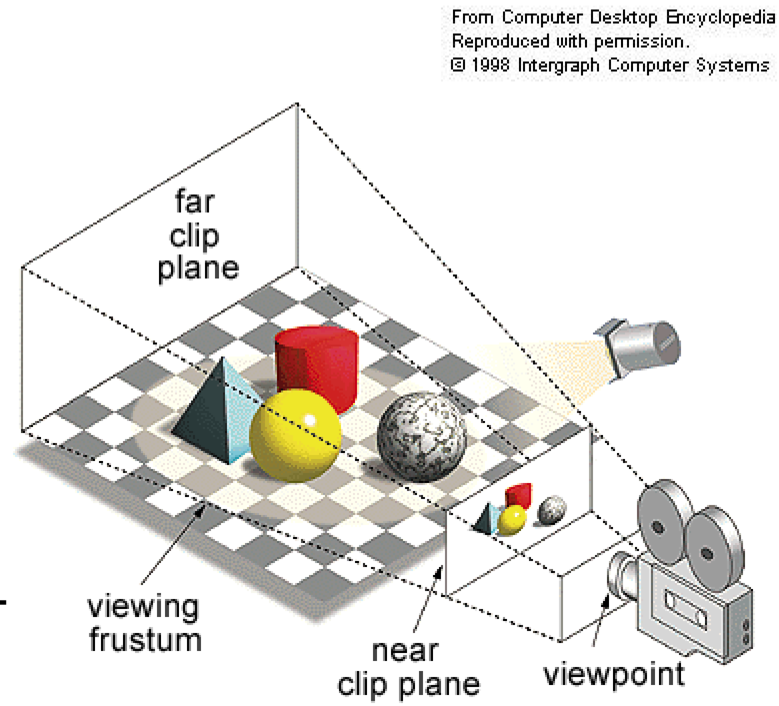
Трехмерная сцена

Сцена обычно содержит:

- объекты
- камера (наблюдатель, игрок)
- источники света

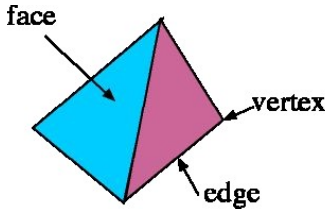
Камера - это виртуальный объект (полный аналог видеокамеры из реального мира)

Она задаёт то окно, через которое вы смотрите на виртуальный мир сцены

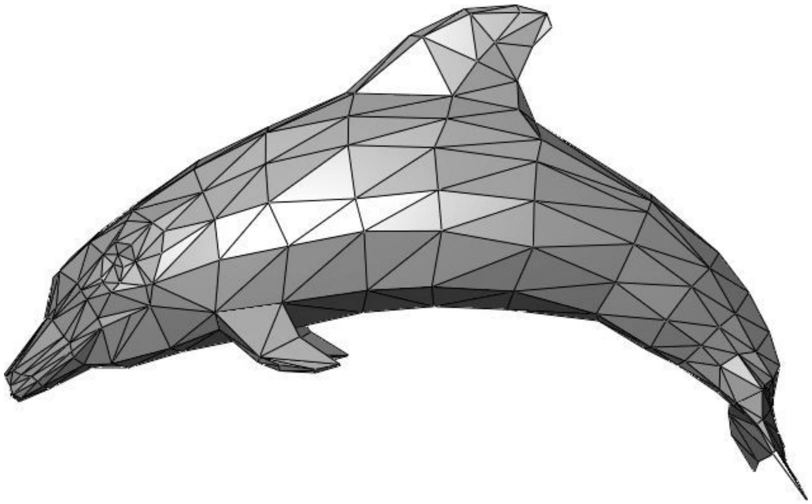


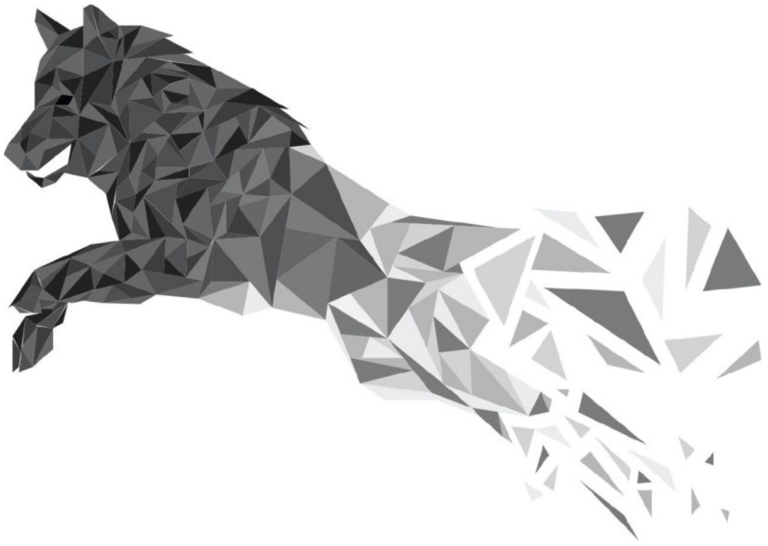
Объекты. Polygon Mesh

Polygon mesh[1] is a collection of **geometric primitives**[2] that defines the shape of a object in 3D computer graphics



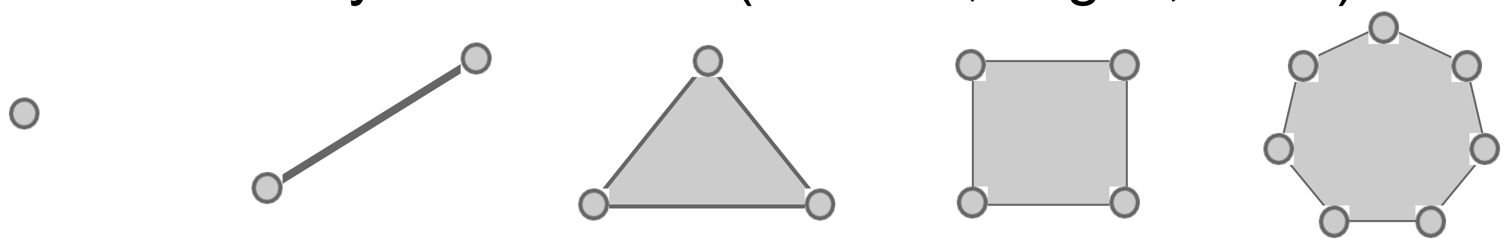
polygon mesh[1] or just **mesh**
geometric primitives[2] is a collection of **vertices**, **edges** and **faces**





Объекты. Базовые примитивы

geometric primitive[1] = **simplest**[2] geometric object that can be **processed**[3] by **video card**[4] (vertices, edges, faces)



вершина	ребро	треугольник	4х-угольник	многоугольник
vertex	edge	triangle, polygon	quad	polygon

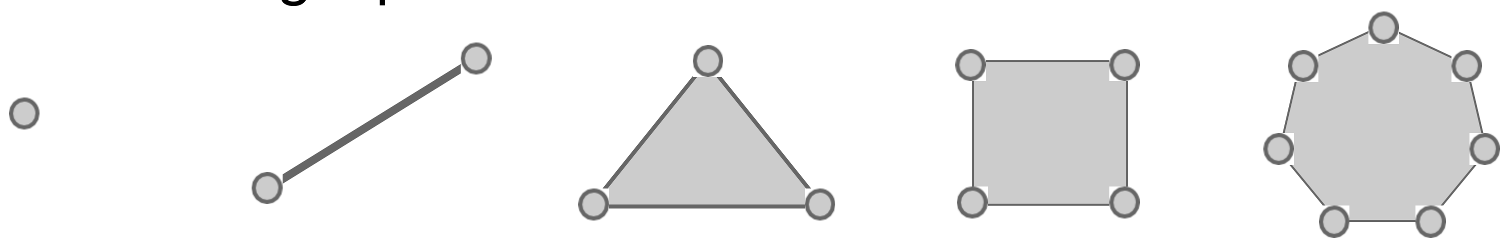
Современные GPU рисуют 1-10 млн. примитивов за каждый кадр приложения

geometric primitive[1] имеет синонимы: graphic primitive, primitive
simplest[2] - минимальный по количеству точек объект (выпуклый)
process[3] - обычно имеется в виду отрисовать на экран
video card[4] имеет синонимы: display card, graphics card, display adapter or graphics adapter

Вершины

geometric primitives is a collection of **vertices**

vertex is a single point



each vertex can have several properties:

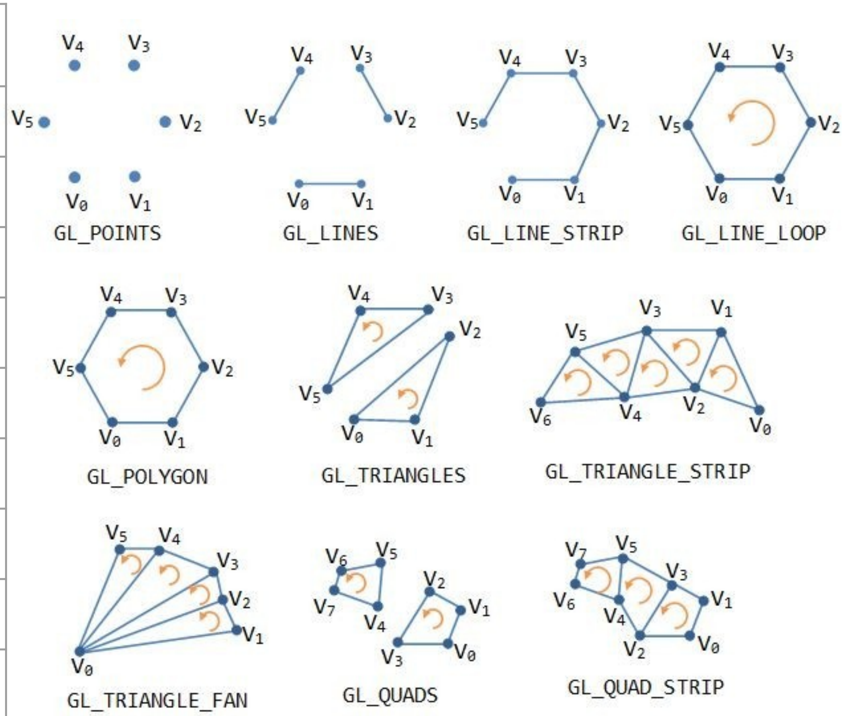
- position (x, y, z)
- color (r, g, b)
- normal vector (nx, ny, nz)
- texture coordinates (u, v)
- ...

Primitive Topology

primitive topology = how to interpret a list of vertices

имеет синонимы: topology, mesh topology, mesh connectivity

Topology	**DirectX**	**OpenGL**
Point list	D3DPT_POINTLIST	GL_POINTS
Line list	D3DPT_LINELIST	GL_LINES
Line strip	D3DPT_LINESTRIP	GL_LINE_STRIP
Line loop		GL_LINE_LOOP
Polygon		GL_POLYGON
Triangle list	D3DPT_TRIANGLELIST	GL_TRIANGLES
Triangle strip	D3DPT_TRIANGLESTRIP	GL_TRIANGLE_STRIP
Triangle fan	D3DPT_TRIANGLEFAN	GL_TRIANGLE_FAN
Quad list		GL_QUADS
Quad strip		GL_QUAD_STRIP



OpenGL Primitives

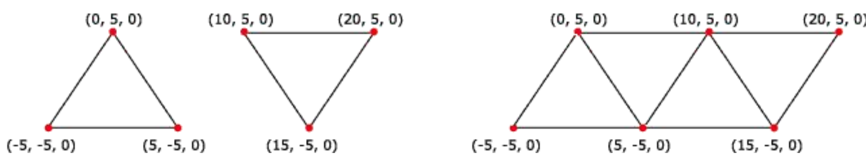
Primitive Topology

primitive topology = how to interpret a list of vertices

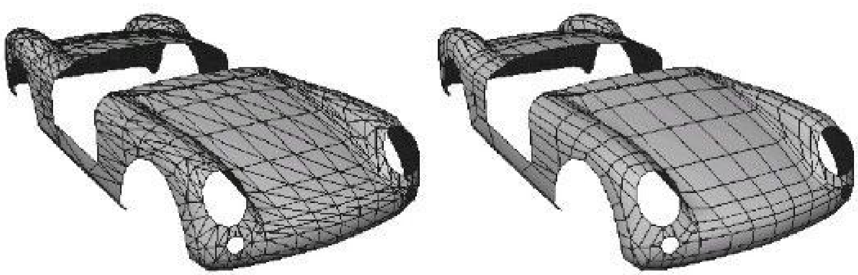
имеет синонимы: topology, mesh topology, mesh connectivity

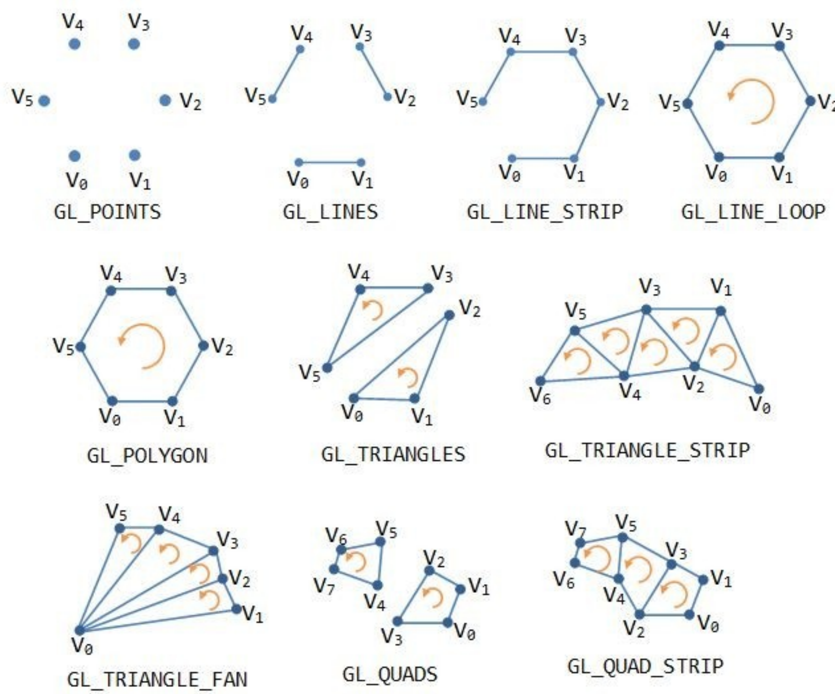
Набор топологий избыточен:

1. triangle list vs triangle strip



2. triangle strip vs quad strip



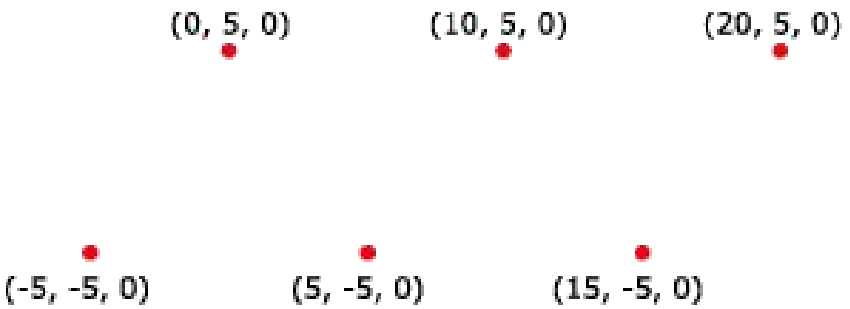


OpenGL Primitives

Primitive Topology

Point Lists - набор не связанных вершин
(пример на OpenGL 2.0)

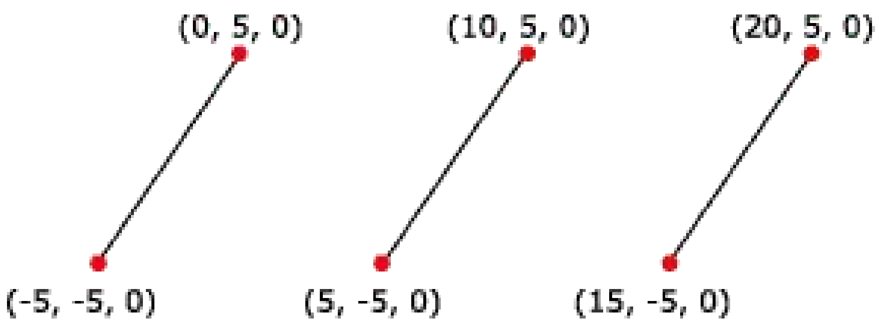
```
glBegin(GL_LINES);
{
   glVertex3f( -5.0,  -5.0, 0.0 );
   glVertex3f(  0.0,   5.0, 0.0 );
   glVertex3f(  5.0,  -5.0, 0.0 );
   glVertex3f( 10.0,  5.0, 0.0 );
   glVertex3f( 15.0, -5.0, 0.0 );
   glVertex3f( 20.0,  5.0, 0.0 );
}
glEnd();
```



Primitive Topology

Line Lists - набор не связанных отрезков
(пример на DirectX 9)

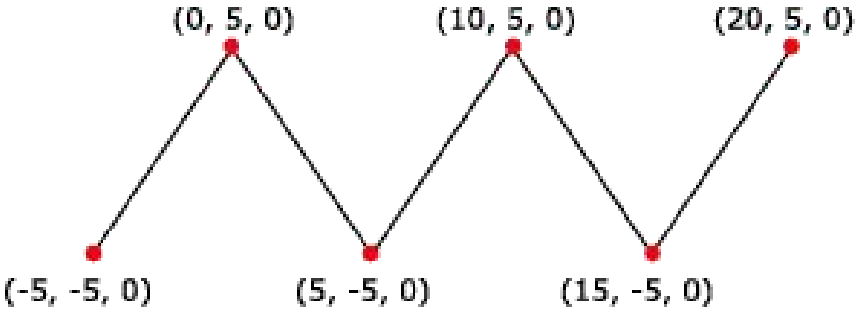
```
struct CUSTOMVERTEX
{
        float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
        {-5.0,  -5.0, 0.0},
        { 0.0,   5.0, 0.0},
        { 5.0,  -5.0, 0.0},
        {10.0,  5.0, 0.0},
        {15.0, -5.0, 0.0},
        {20.0, 5.0, 0.0}
};

d3dDevice->DrawPrimitive( D3DPT_LINELIST, 0, 3 );
```



Primitive Topology

Line Strips - набор связанных отрезков (пример на DirectX 9)

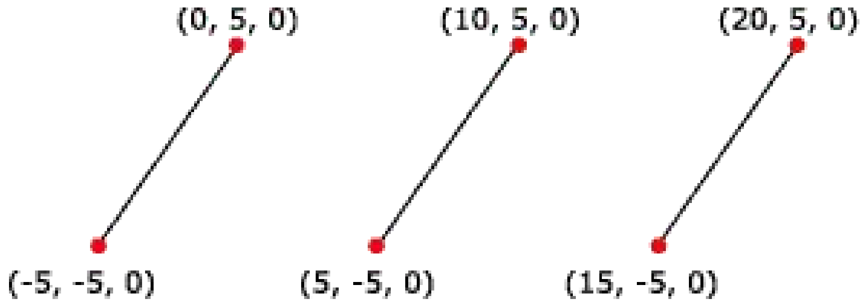
```
struct CUSTOMVERTEX
{
        float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
            {-5.0,  -5.0, 0.0},
            { 0.0,   5.0, 0.0},
            { 5.0,  -5.0, 0.0},
            {10.0,  5.0, 0.0},
            {15.0, -5.0, 0.0},
            {20.0, 5.0, 0.0}
};

d3dDevice->DrawPrimitive( D3DPT_LINESTRIP, 0, 5 );
```



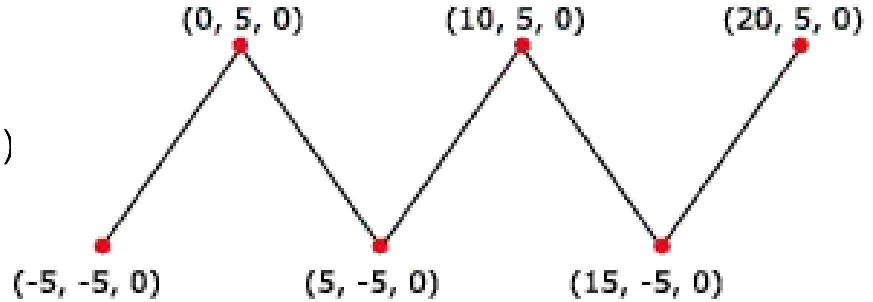
Primitive Topology

Line Lists vs Line Strips (пример на DirectX 9)

d3dDevice->DrawPrimitive(D3DPT_LINELIST, 0, 3);



d3dDevice->DrawPrimitive(D3DPT_LINESTRIP, 0, 5)

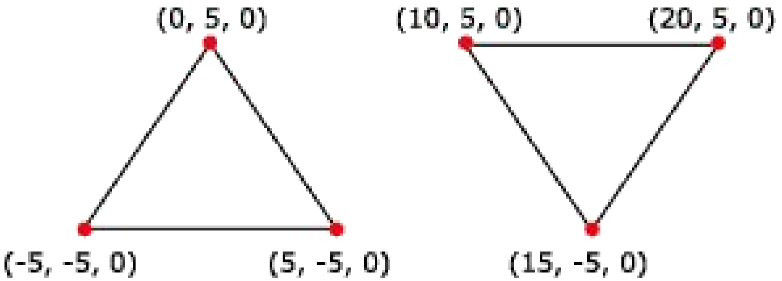


Line Strips позволяет нарисовать больше отрезков (при том же кол-ве вершин) - требует меньше оперативной памяти и меньше времени процессора и видеокарты

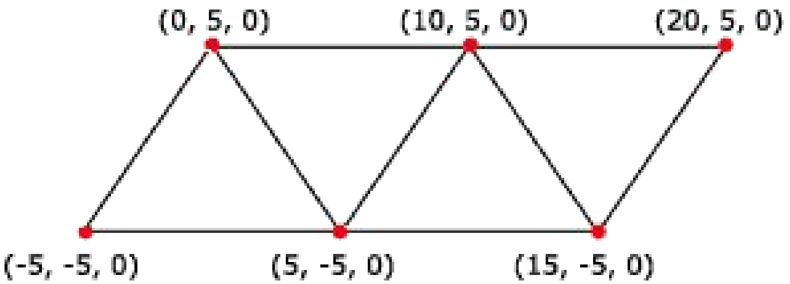
Primitive Topology

Triangle Lists vs Triangle Strips (пример на OpenGL 2.0)

```
glBegin(GL_TRIANGLES);
{
    glVertex3f( -5.0,  -5.0, 0.0 );
    glVertex3f(  0.0,   5.0, 0.0 );
    glVertex3f(  5.0,  -5.0, 0.0 );
      glVertex3f( 10.0,  5.0, 0.0 );
      glVertex3f( 15.0, -5.0, 0.0 );
      glVertex3f( 20.0,  5.0, 0.0 );
}
glEnd();
```

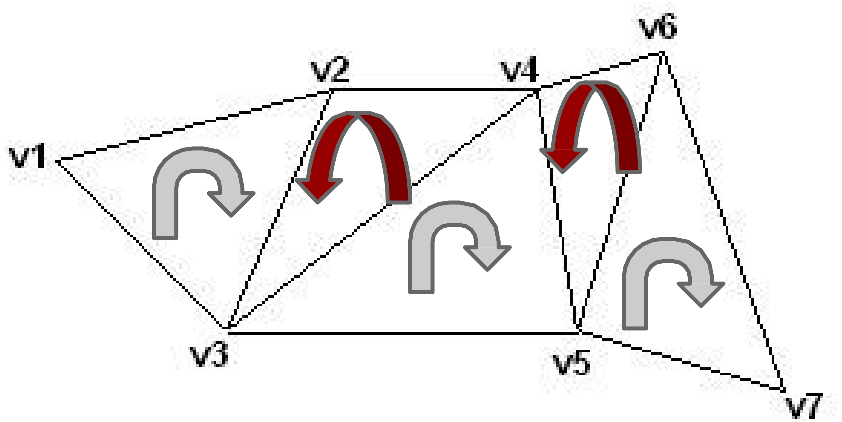



```
glBegin(GL_TRIANGLE_STRIP);
{
    glVertex3f( -5.0,  -5.0, 0.0 );
      glVertex3f( 0.0,   5.0, 0.0 );
        glVertex3f( 5.0, -5.0, 0.0 );
          glVertex3f( 10.0,  5.0, 0.0 );
          glVertex3f( 15.0, -5.0, 0.0 );
          glVertex3f( 20.0,  5.0, 0.0 );
}
glEnd();
```



Primitive Topology

Triangle Strips нужно аккуратно использовать



Вершины **v1**, **v2**, **v3** обходятся по часовой стрелке,
вершины **v2**, **v3**, **v4** обходятся против часовой стрелки,
затем снова по часовой (**v3**, **v4**, **v5**),
потом против часовой (**v4**, **v5**, **v6**),
и т.д.

Пользуйтесь этим правилом при создании массива вершин,
который затем передадите на видеокарту

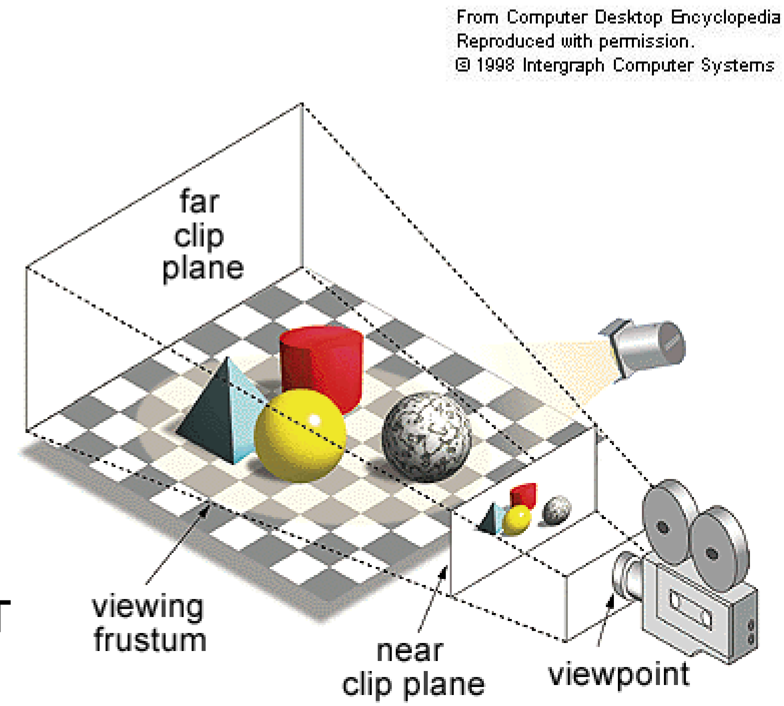
Трехмерная сцена

Сцена обычно содержит:

- объекты
- камера (наблюдатель, игрок)
- источники света

Камера - это виртуальный объект (полный аналог видеокамеры из реального мира)

Она задаёт то окно, через которое вы смотрите на виртуальный мир сцены



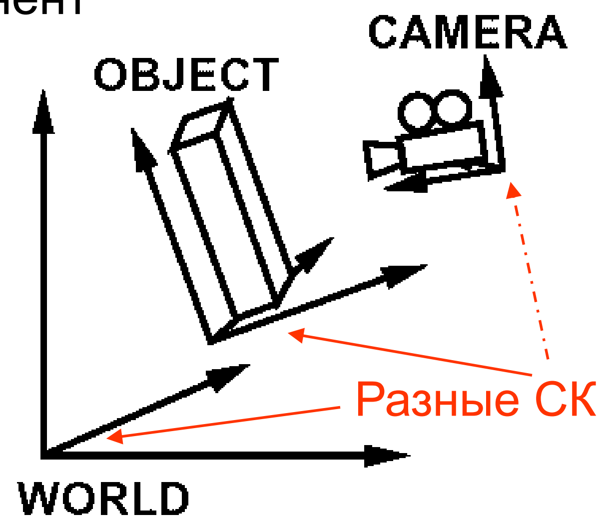
Объекты. Системы координат

Преобразования

Модельные
- Создание сложных моделей из простых компонент путем позиционирования
- Преобразование из объектных координат в мировые

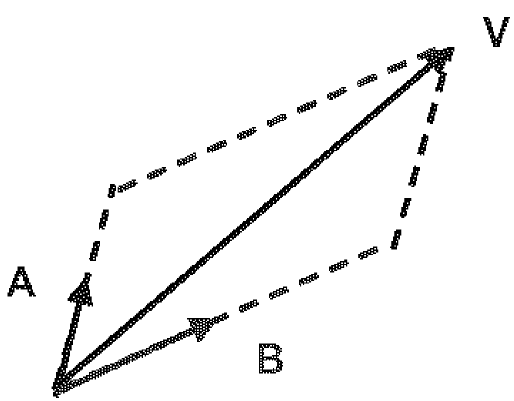
Видовые
- Размещение виртуальной камеры в мире, т.е. Спецификация преобразования мировых координат в координаты камеры

Анимации
- Варьирование координат во времени для создания движений



Векторная алгебра

- Линейная комбинация векторов
- Линейная независимость векторов
- Ортонормированный базис
- Компоненты = координаты вектора
- Вектор в базисе представляется однозначно
- При смене базиса вектор не меняется, меняются его компоненты



$$V = v_1 A + v_2 B;\ v_1, v_2 \in \Re$$

$$V = v_1 E_1 + v_2 E_2 + \ldots + v_n E_n$$

$$V = \begin{pmatrix} v_1 \\ v_2 \\ \ldots \\ v_n \end{pmatrix}$$

Линейные и аффинные преобразования

F линейное, если $F(A+B)=F(A)+F(B)$ и $F(kA)=kF(A)$

Любое линейное преобразование полностью специфицируется его действием на базисные векторы

$$
\begin{aligned}
V &= v_1E_1 + v_2E_2 + v_3E_3 \\
F(V) &= F(v_1E_1 + v_2E_2 + v_3E_3) \\
&= F(v_1E_1) + F(v_2E_2) + F(v_3E_3) \\
&= v_1F(E_1) + v_2F(E_2) + v_3F(E_3)
\end{aligned}
$$

F аффинное, если оно линейное + сдвиг $\Rightarrow$ $y=mx+b$

Линейное преобразование

Преобразование базисных векторов

$$F(E_1) = f_{11}E_1 + f_{21}E_2 + f_{31}E_3$$
$$F(E_2) = f_{12}E_1 + f_{22}E_2 + f_{32}E_3$$
$$F(E_3) = f_{13}E_1 + f_{23}E_2 + f_{33}E_3$$

Преобразование вектора

$$\begin{aligned} F(V) &= v_1F(E_1) + v_2F(E_2) + v_3F(E_3) \\ &= (f_{11}E_1 + f_{21}E_2 + f_{31}E_3)v_1 + (f_{12}E_1 + f_{22}E_2 + f_{32}E_3)v_2 + (f_{13}E_1 + f_{23}E_2 + f_{33}E_3)v_3 \\ &= (f_{11}v_1 + f_{12}v_2 + f_{13}v_3)E_1 + (f_{21}v_1 + f_{22}v_2 + f_{23}v_3)E_2 + (f_{31}v_1 + f_{32}v_2 + f_{33}v_3)E_3 \end{aligned}$$

Или в другой записи

$$\hat{v}_1 = f_{11}v_1 + f_{12}v_2 + f_{13}v_3$$
$$\hat{v}_2 = f_{21}v_1 + f_{22}v_2 + f_{23}v_3$$
$$\hat{v}_3 = f_{31}v_1 + f_{32}v_2 + f_{33}v_3$$

$$\hat{v}_i = \sum_j f_{ij}v_j$$

$$\begin{pmatrix} v_1' \\ v_2' \\ v_3' \end{pmatrix} = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

Вектор-строка или вектор-столбец

$$\begin{pmatrix} v'_1 \\ v'_2 \\ v'_3 \end{pmatrix} = \begin{pmatrix} f_{11} f_{12} f_{13} \\ f_{21} f_{22} f_{23} \\ f_{31} f_{32} f_{33} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \qquad v' = M \cdot v$$

$$(v'_1, v'_2, v'_3) = (v_1, v_2, v_3) \begin{pmatrix} f_{11} f_{21} f_{31} \\ f_{12} f_{22} f_{32} \\ f_{13} f_{23} f_{33} \end{pmatrix} \qquad v' = v \cdot M^T$$

Основные 2D преобразования

Преобразование	**Формула**		**Управляющий параметр**
Сдвиг Translate	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$	$v' = v + t$	t
Масштабирование Scale	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$	$v' = Sv$	S
Поворот Rotate	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$	$v' = R(\theta)v$	θ

Основные 2D преобразования

Преобразование	Формула		Управляющий параметр
Сдвиг Translate	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$	$v' = v + t$ **Нелинейное преобразование**	t
Масштабирование Scale	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$	$v' = Sv$	S
Поворот Rotate	$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$	$v' = R(\theta)v$	θ

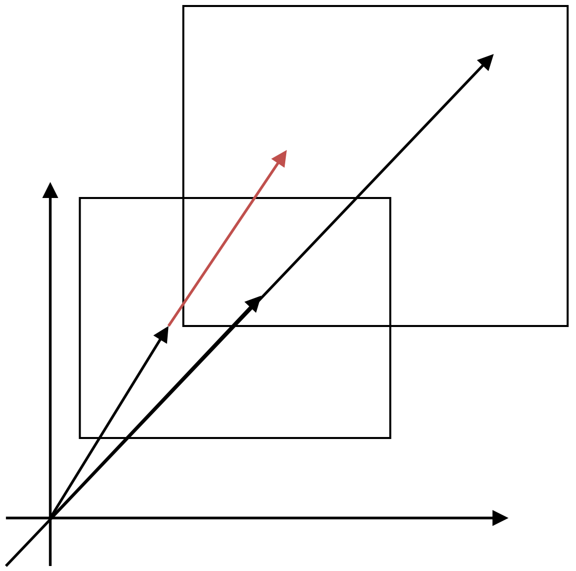
Однородные координаты

- Как бы представить сдвиг в матричном виде
- «трюк» – добавим по лишней координате к векторам

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Это *однородная* координата *w*
- После преобразований отбрасываем её (*w*=1) и всё
- Такие матрицы определяют *однородные* преобразования

W? WHAT?!



- Практический ответ:
  - W – это алгебраический трюк
  - Если w не равно 1.0, то дели на него
- Академический ответ:
  - (x, y, w) координаты из 3D проективного пространства
  - все ненулевые кратные w(x, y, 1) определяют одну и ту же точку в декартовых координатах (x, y)
  - w=0 означает точку в бесконечности, что эквивалентно направлению
  - в 3D графике используются четырехмерные однородные координаты (x, y, z, w)

Грубо: z = w

Однородные 2D преобразования

Translate

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Scale

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotate

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Теперь любая последовательность сдвигов/вращений/масштабирований на плоскости может быть выражена одной матрицей

Однородные 3D преобразования

Translate

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scale

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Любая последовательность сдвигов/вращений/масштабирований в пространстве может быть выражена одной матрицей

Однородные 3D преобразования

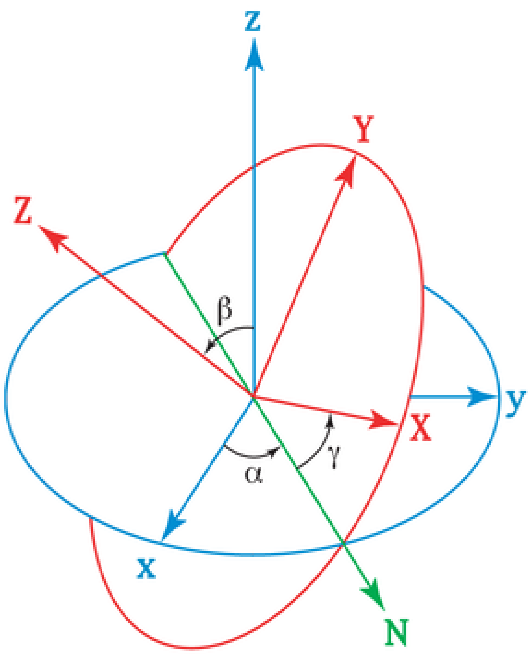
Rotate

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Углы Эйлера



Углы Эйлера определяют три поворота системы, которые позволяют привести любое положение системы к текущему. Обозначим начальную систему координат как (x, y, z), конечную как (X, Y, Z). Пересечение координатных плоскостей xy и XY называется линией узлов N.

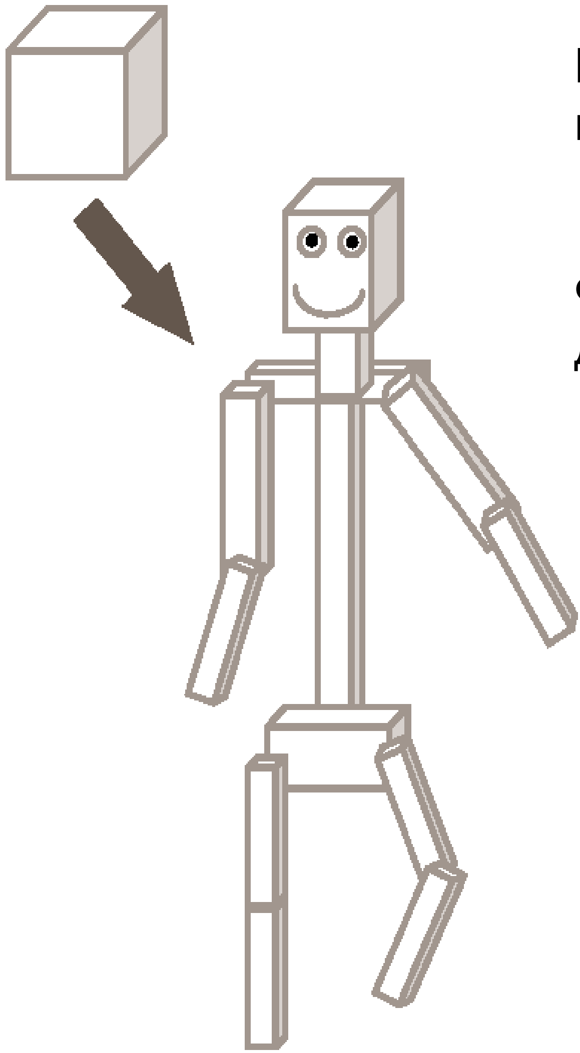
Угол α между осью x и линией узлов.

Угол β между осями z и Z.

Угол γ между осью X и линией узлов.

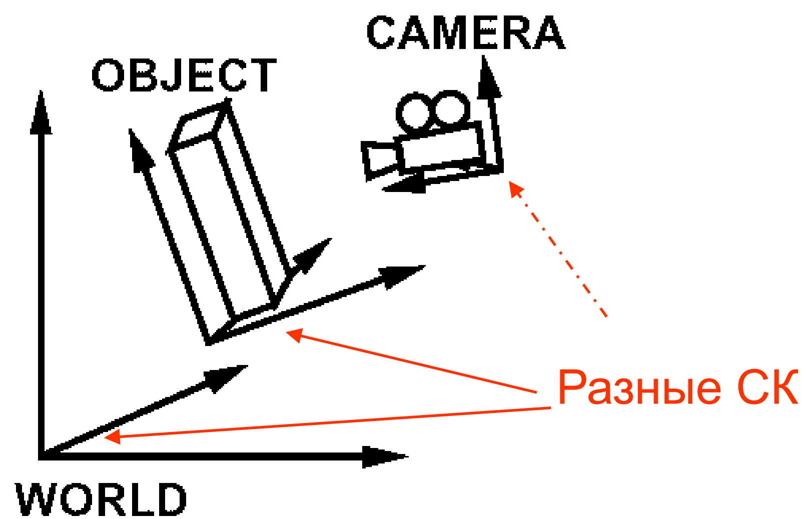
Повороты системы на эти углы называются прецессия, нутация и поворот на собственный угол (вращение). Такие повороты некоммутативны и конечное положение системы зависит от порядка, в котором совершаются повороты. В случае углов Эйлера это последовательность 3, 1, 3 (z, N, Z).

Модельные преобразования



Все из одного базового кубика: повороты, масштабы, сдвиги

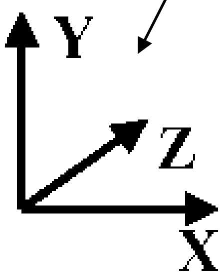
Физически корректные деформации

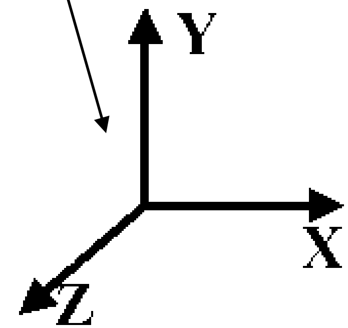


Соглашения:

вектор – столбец	вектор – строка
$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$	$\begin{pmatrix} x' & y' & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{pmatrix}$
$p' = ABCDp$	$p'^T = p^T D^T C^T B^T A^T$

Левосторонние и правосторонние системы координат



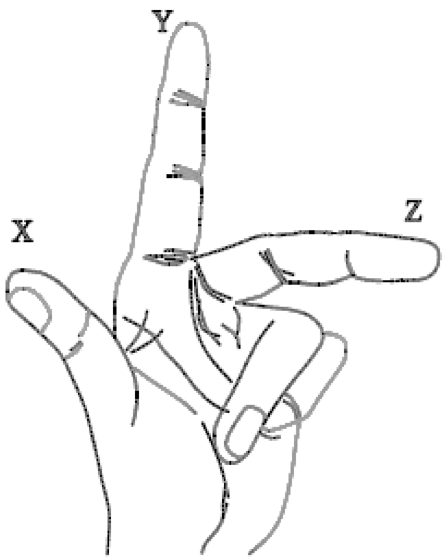


$$Z = X \times Y = \begin{pmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{pmatrix}$$

Ось Z определяется через оси X и Y на основе векторного произведения, которое определяется правилом левой или правой руки (согласно выбранной системе)

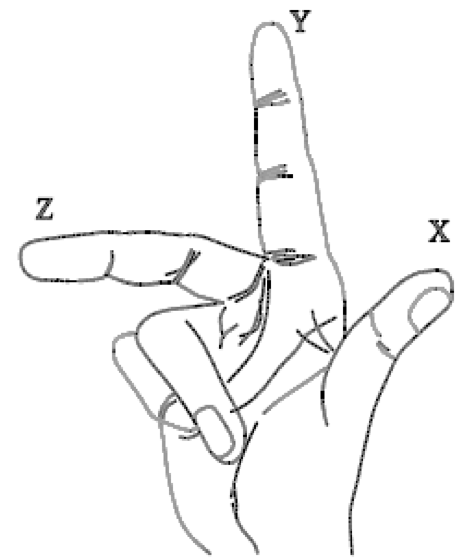
Левосторонние и правосторонние системы координат

Левая рука



$$\vec{Y} \times \vec{X} = \vec{Z}$$

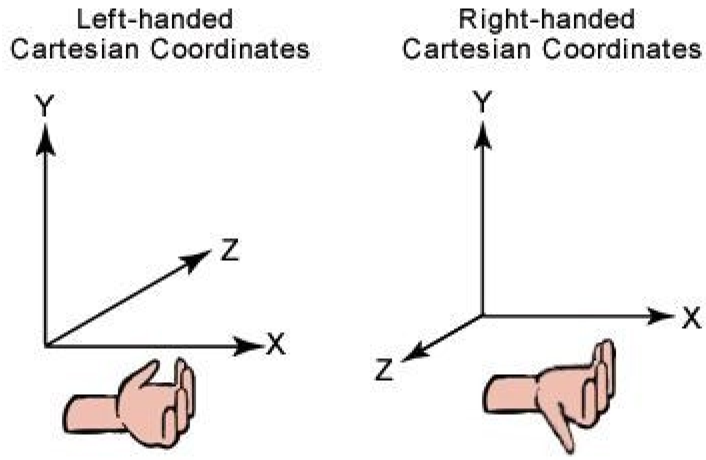
Правая рука



$$\vec{X} \times \vec{Y} = \vec{Z}$$

Системы координат

По-умолчанию	
в **DirectX** - левая с.к.	в **OpenGL** - правая с.к.



Многие предпочитают левую с.к., т.к. она интуитивно более удобна - ось Z направлена от нас

Если хотим сместится вперёд - увеличиваем Z-координату

Вектор-строка или вектор-столбец

$M_W = M_S * M_R * M_T = M_S * (M_Y * M_X * M_Z) * M_T$ (DirectX)

$M_W = M_T * M_R * M_S = M_T * (M_Z * M_X * M_Y) * M_S$ (OpenGL)

Правила:

1. $(p * M)^T = M^T * p^T$, где M - матрица, p - вектор-строка, p^T - вектор-столбец.
2. $(M_A * M_B)^T = M_B^T * M_A^T$, где M_A и M_B - матрицы.
3. $M^{TT} = M$.

(DirectX) $p * M_W = [p * (M_S * M_R * M_T)]^{TT} =$

$= [(M_S * M_R * M_T)^T * p^T]^T =$

$= [(M_T^T * M_R^T * M_S^T) * p^T]^T$ (OpenGL)

Помним:

1. Любая матрица из DirectX == траспонированная матрица из OpenGL.
2. Вектор - это строка в DirectX, вектор - это столбец в OpenGL.

Получается, что **в DirectX и OpenGL результат достигается одинаковый**.

Scene. Objects and Camera

Объект имеет свойства:

позиция ориентация масштаб	position rotation scale	(px, py, pz) (rx, ry, rz) (sx, sy, sz)	три координаты три угла эйлера три коэф-та

World Transform
convert vertices:
- from **model** space
- to **world** space

Камера имеет свойства:

позиция ориентация масштаб	position rotation scale	(px, py, pz) (rx, ry, rz) (sx, sy, sz)	три координаты три угла эйлера три коэф-та
угол обзора аспект ближняя пл-ть дальняя пл-ть	fov aspect near far	field of view aspect of screen near plane far plane	число в градусах ширина / высота число в метрах число в метрах

View Transform
convert vertices:
- from **world** space
- to **camera** space

Projection Transform
convert vertices:
- from **camera** space
- to **screen** space

Transforms. Translation Matrix

Translation matrix - set object **position**	translates vertices	M_T	DirectX $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{pmatrix}$ OpenGL $\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$

DirectX

$$p_{new} = position * M_T = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{pmatrix} = \begin{pmatrix} x+Tx & y+Ty & z+Tz & 1 \end{pmatrix}$$

OpenGL

$$p_{new} = M_T * position = \begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+Tx & y+Ty & z+Tz & 1 \end{pmatrix}$$

Transforms. Translation Matrix

Translation matrix - set object **position**	translates vertices	M_T	DirectX $$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{pmatrix}$$	OpenGL $$\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

DirectX

```
D3DXMATRIX * D3DXMatrixTranslation(
    D3DXMATRIX * pOut,
    FLOAT x, FLOAT y, FLOAT z
);
```

OpenGL

```
void glMatrixMode(GLenum mode);
void glTranslatef(GLfloat x, GLfloat y, GLfloat  z);
glm::mat4x4 glm::translate(glm::mat4x4 const & m, glm::vec3 const & v);
void glLoadMatrixf(const GLfloat * m);
```

Transforms. Rotation Matrix

Rotation matrix - set object **rotation**	rotate vertices	M_R	$M_R = M_Y * M_X * M_Z$ (DirectX) $M_R = M_Z * M_X * M_Y$ (OpenGL)

DirectX

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

вокруг оси X **вокруг оси Y** **вокруг оси Z**

OpenGL

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

вокруг оси X **вокруг оси Y** **вокруг оси Z**

Transforms. Rotation Matrix

Rotation matrix - set object **rotation**	rotate vertices	M_R	$M_R = M_Y * M_X * M_Z$ (DirectX) $M_R = M_Z * M_X * M_Y$ (OpenGL)

DirectX

D3DXMATRIX * D3DXMatrixRotationX(D3DXMATRIX * pOut, FLOAT Angle);
D3DXMATRIX * D3DXMatrixRotationY(D3DXMATRIX * pOut, FLOAT Angle);

D3DXMATRIX * D3DXMatrixRotationZ(D3DXMATRIX * pOut, FLOAT Angle);

OpenGL

void glMatrixMode(GLenum mode);

void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

https://open.gl/transformations

glm::mat4x4 glm::rotate(glm::mat4x4 const & m, T angle, glm::vec3 const & axis);

void glLoadMatrixf(const GLfloat * m);

Transforms. Scale Matrix

Scale matrix - set object **scale**	scale vertices	M_S	DirectX $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	OpenGL $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

DirectX

$$p_{new} = position * M_S = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \times \begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x * Sx & y * Sy & z * Sz & 1 \end{pmatrix}$$

OpenGL

$$p_{new} = M_S * position = \begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x * Sx & y * Sy & z * Sz & 1 \end{pmatrix}$$

Transforms. Scale Matrix

Scale matrix - set object **scale**	scale vertices	M_S	DirectX $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ OpenGL $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

DirectX

```
D3DXMATRIX * D3DXMatrixScaling(
    D3DXMATRIX * pOut,
    FLOAT sx, FLOAT sy, FLOAT sz
);
```

OpenGL

```
void glMatrixMode(GLenum mode);
void glScalef(GLfloat x, GLfloat y, GLfloat  z);
glm::mat4x4 glm::scale(glm::mat4x4 const & m, glm::vec3 const & v);
void glLoadMatrixf(const GLfloat * m);
```

Transforms. World Matrix

World Transform World matrix: **translation,** **rotation**, **scale**	convert vertices: - from **model** space - to **world** space	M_W	$M_W = M_S * M_R * M_T$ (DirectX) $M_W = M_T * M_R * M_S$ (OpenGL)
Translation matrix - set object **position**	translates vertices	M_T	DirectX $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{pmatrix}$ OpenGL $\begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Rotation matrix - set object **rotation**	rotate vertices	M_R	$M_R = M_Y * M_X * M_Z$ (DirectX) $M_R = M_Z * M_X * M_Y$ (OpenGL)
Scale matrix - set object **scale**	scale vertices	M_S	DirectX $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ OpenGL $\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Transforms (Преобразования)

Любой Transform реализован в виде матрицы 4x4 (внутри DirectX и OpenGL)

World Transform World Matrix	convert vertices: - from **model** space - to **world** space	M_W
View Transform View Matrix	convert vertices: - from **world** space - to **camera** space	M_V
Projection Transform Projection Matrix	convert vertices: - from **camera** space - to **screen** space	M_P

$position_{screen}$ = position * Mw * Mv * Mp,
где position - позиция вершины в системе координат модели (px, py, pz, 1)
$position_{screen}$ - позиция вершины на экране

Transforms. View Transform

View Transform View Matrix	convert vertices: - from **world** space - to **camera** space	M_V

Содежит:

- позицию камеры в глобальной с.о.
- направление взгляда
- направление вектора Up

DirectX

D3DXMatrixLookAtLH() // для левой с.к. (по-умолчанию)

D3DXMatrixLookAtRH() // для правой с.к.

OpenGL

gluLookAt() // для правой с.к. (по-умолчанию)

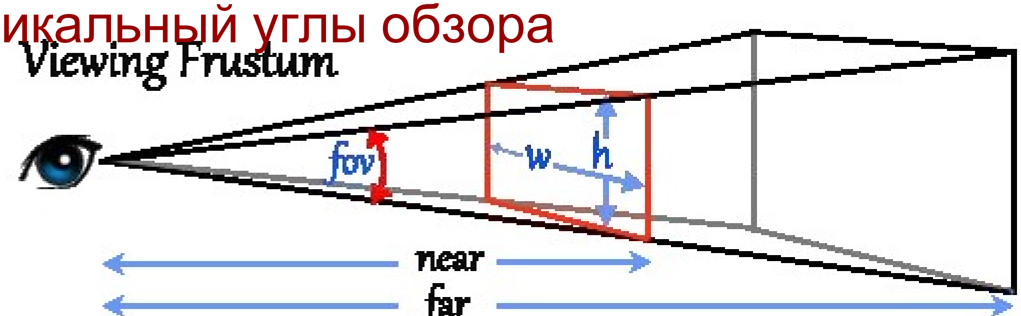
glm::lookAt()

Transforms. Projection Transform

[Projection Transform] Projection Matrix	convert vertices: - from **camera** space - to **screen** space	M_P

Содежит:

- ближнюю и дальнюю отсекающие плоскости
- горизонтальный и вертикальный углы обзора



DirectX

D3DXMatrixPerspectiveLH() // для левой с.к. (по-умолчанию)

D3DXMatrixPerspectiveRH() // для правой с.к.

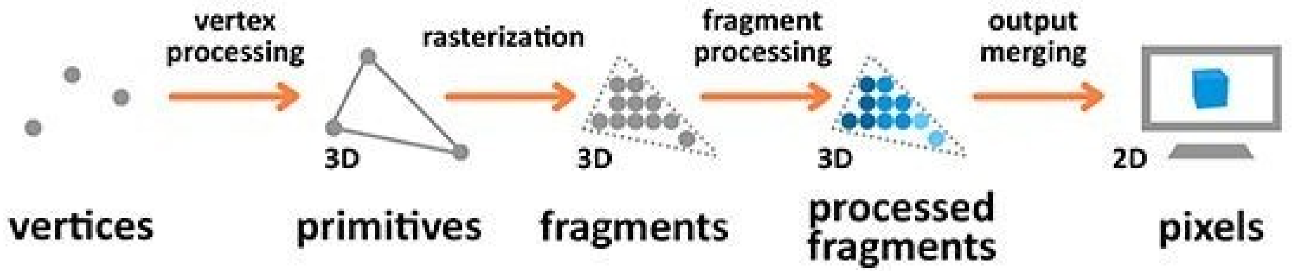
OpenGL

glFrustum() // для правой с.к. (по-умолчанию)

glm::perspective()

Transforms. Graphics Pipeline

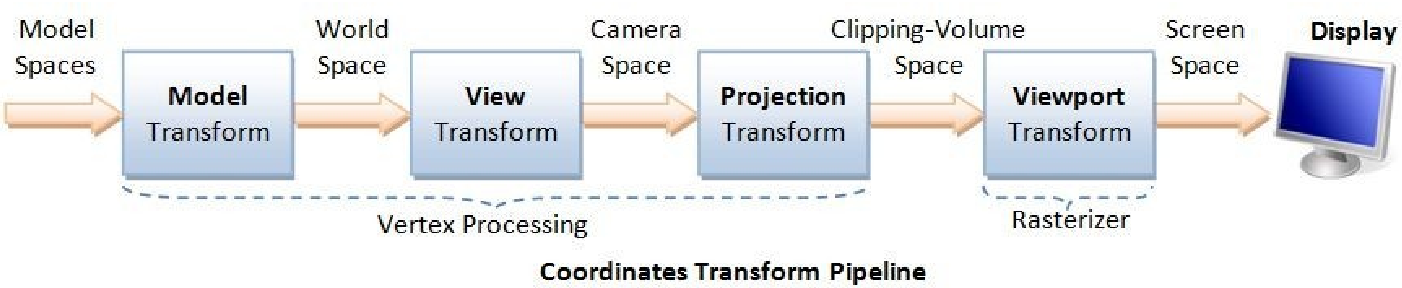
Graphics Pipeline (DirectX 9, DirectX 11, OpenGL)



$position_{screen}$ = position * Mw * Mv * Mp (стадия **vertex processing** на рис.),

где position - позиция вершины в системе координат модели (px, py, pz, 1)

$position_{screen}$ - позиция вершины на экране

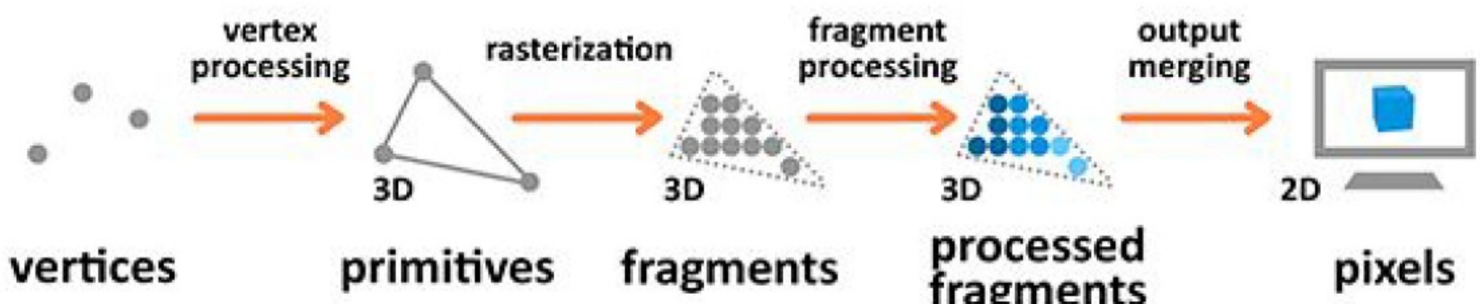


Конвейерная обработка

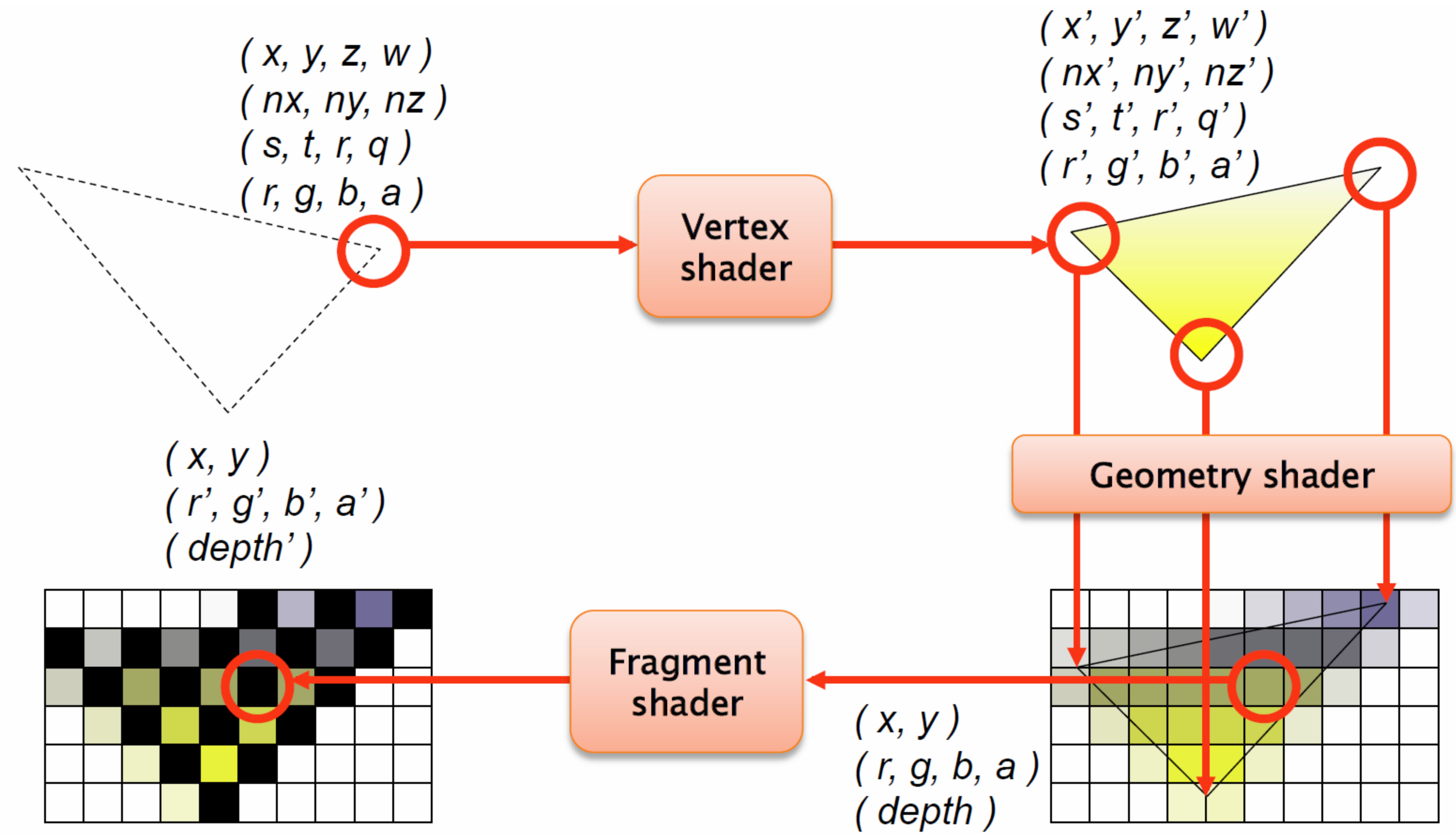
1. Формирование 3D-сцены
   (анимация, выбор, детализация…)
2. Декомпозиция на примитивы
3. Модификация вершин
   (вершинные шейдеры)
4. Обход примитивов
5. Рендеринг пикселов
   (пиксельные шейдеры)
6. Формирование итогового изображения

Конвейерная обработка

- Обход и рендеринг примитивов:
- Умножение вершин на матрицы преобразований (включая видовое преобразование)
- Перебор всех «треугольников» по набору вершин
- Клипирование и растрирование
- Текстурирование

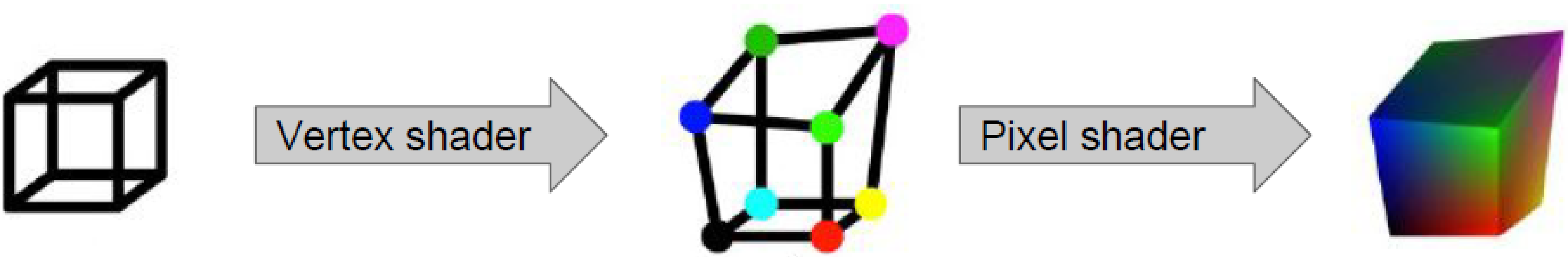


Шейдеры



(x, y, z, w)
(nx, ny, nz)
(s, t, r, q)
(r, g, b, a)
Vertex shader
(x', y', z', w')
(nx', ny', nz')
(s', t', r', q')
(r', g', b', a')
Geometry shader
(x, y)
(r', g', b', a')
(depth')
Fragment shader
(x, y)
(r, g, b, a)
(depth)

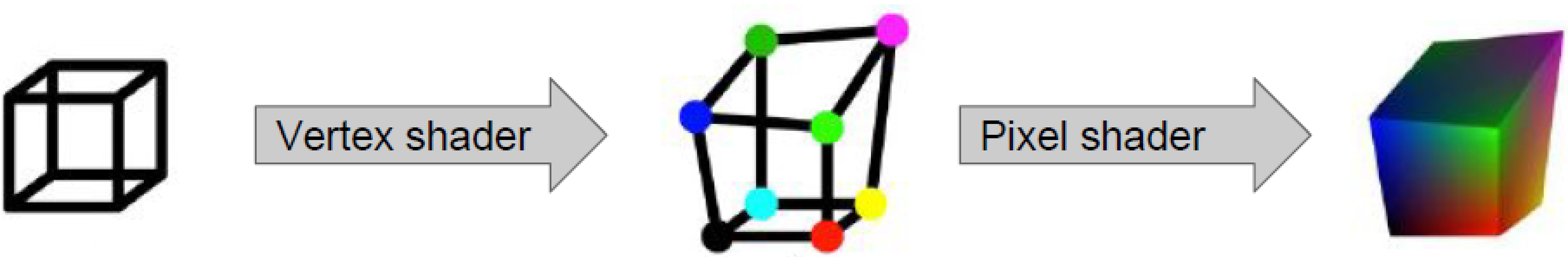
Вершинный шейдер



Небольшая программа, которая исполняется в GPU для каждой вершины:

- Изменяет свойства вершин – позиция, цвет, нормаль, текстурные координаты, …
- Не создает новые и не удаляет существующие вершины

Пиксельный шейдер

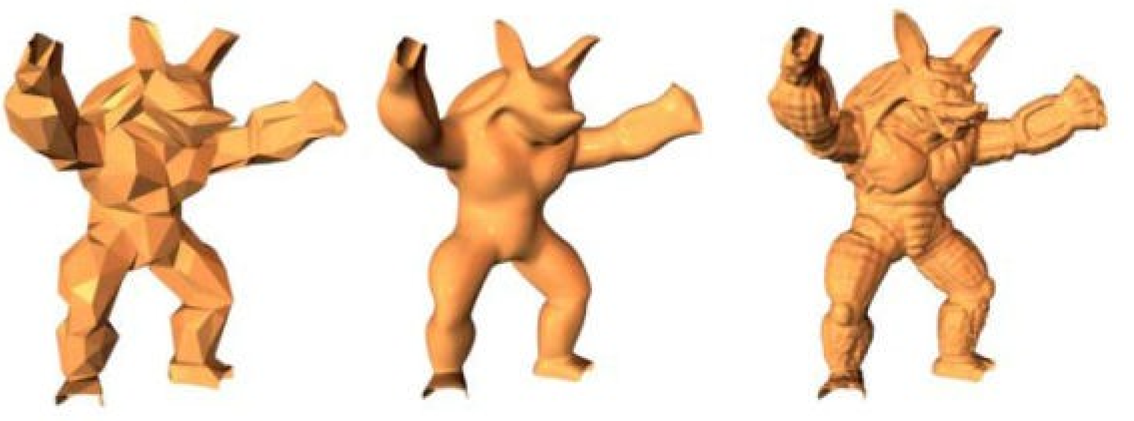


Небольшая программа, которая исполняется в GPU для каждого пикселя:

- обрабатывает данные, связанные с пикселями (например, цвет, глубина, текстурные координаты)
- мультитекстурирование

Геометрический шейдер

Изменяет геометрию сцены (порождает новые примитивы), например N-patch

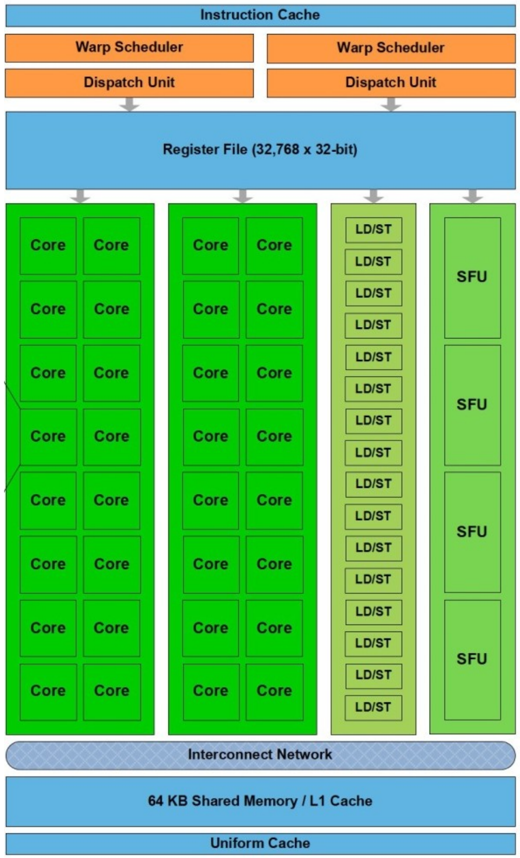


Аппаратная реализация

- Много АЛУ
- Сотни локальных регистров
- Блочная «двумерная» память (медленная), но быстрый кэш

Идеально для параллельной обработки одинаковым кодом разных данных

LD/ST = Load/Store Units
SFU = Special Functions Unit (sin, cosine, square root …)



Fermi Streaming Multiprocessor (SM)