3D 렌더링 파이프 라인 이해와 구현

KGDS2004
2004.08.28
김성익(noerror@hitel.net)
http://digibath.com/noerror

개요

최근 개발 환경에서는 어플리케이션 레벨의 프로그래밍이 중요시 되고 있다. 하지만, 렌더링의 기본적인 부분을 이해한다면 좀 더 유연하고 효율적인 프로그래밍을 할 수 있다.

학습적인 의미로, 고전(?)적인 방법으로 렌더링 파이프 라인의 각 요소를 리뷰해 본다.

강연 대상

1. 3D 프로그램을 처음 접하는 개발자
2. 전체적인 이해는 하고 있지만, 다시 정리를 하고자 하는 개발자

순서

A. Basic

기초 요소 리뷰 (Vector, Matrix)

B. Rendering Pipeline

렌더링 파이프 라인

C. Transformation

월드좌표계, 카메라(뷰) 좌표계
투영, 뷰포트매핑

D. Culling

벡페이스컬링, 클리핑

E. Rasterization

레스터 라이제이션
트라이앵글 렌더링
칼라, 텍스처 블랜딩
Z버퍼링, 알파블랜딩

F. Etc

A. Basic

기본적으로 숙지해야 하는 요소 몇 가지 리뷰

1. 벡터 (Vector)

- 벡터는 스칼라의 조합
- 일반적으로 공간은 직교좌표계를 사용하여 (x, y, z)로 나타낸다
- 위치를 나타내기도 하면 방향을 나타내기도 함

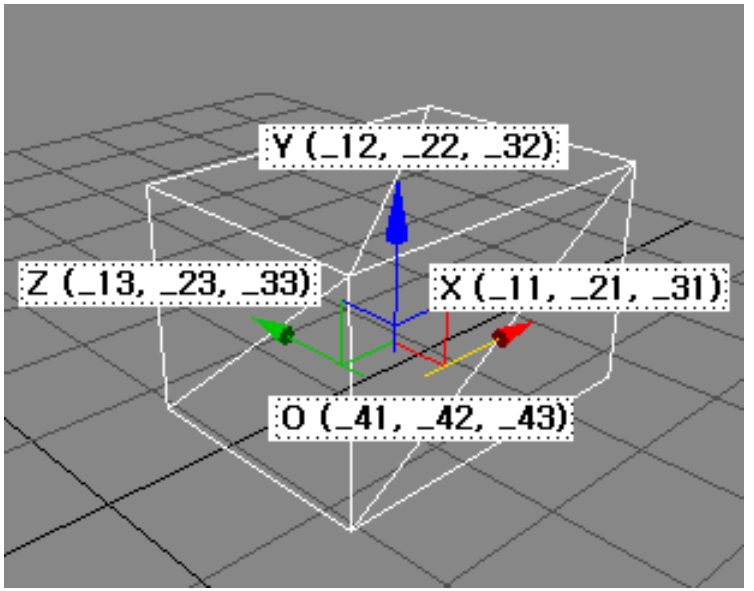
```
struct _VECTOR {
    float x, y, z;
} ;
#define _POINT _VECTOR
```

2. 행렬 (Matrix)

- 행렬은 벡터의 조합
- 일반적인 강체변환(Rigidbody Transformation)을 표현 가능함 (회전, 이동, 확대축소)

- 일반적인 4x4 행렬은 좌표계의 의미를 가진다.

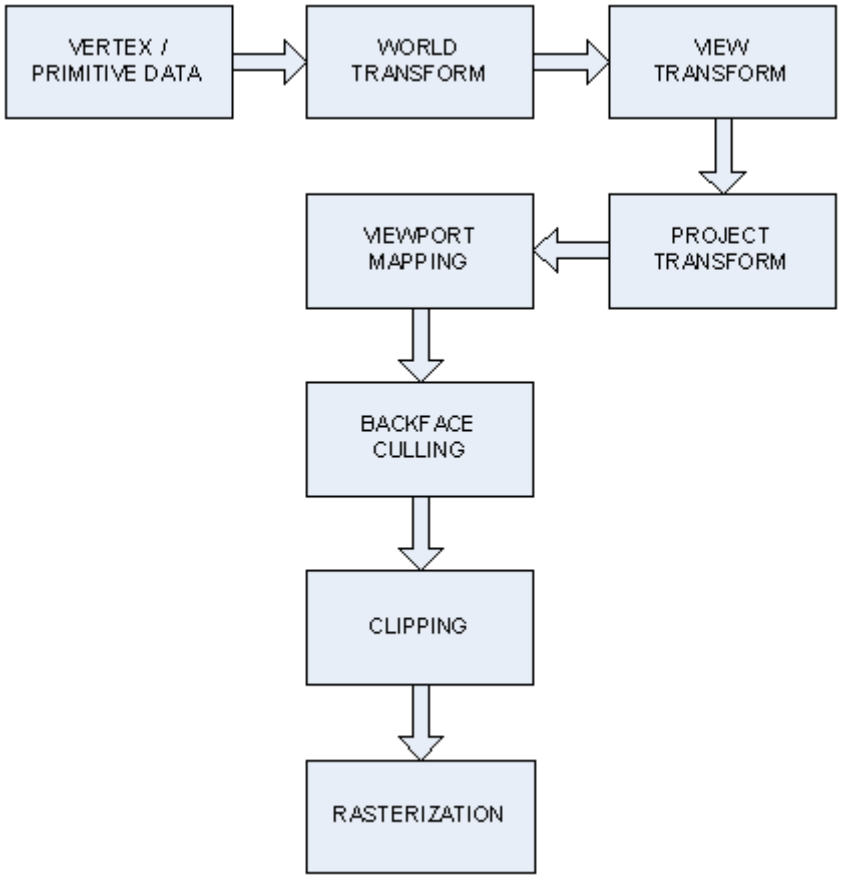
```
struct _MATRIX {
    float _11, _12, _13, _14;
    float _21, _22, _23, _24;
    float _31, _32, _33, _34;
    float _41, _42, _43, _44;
} ;
```



- 겹합 법칙
- 역변환

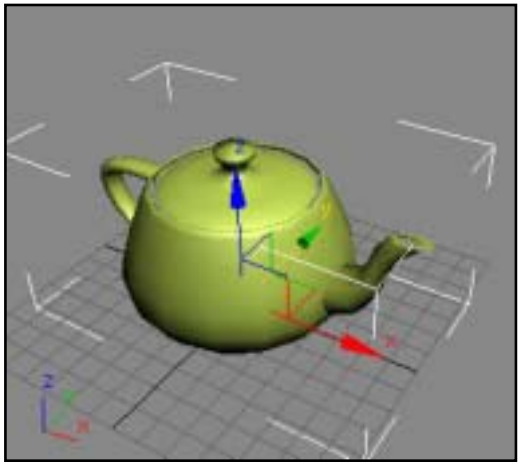
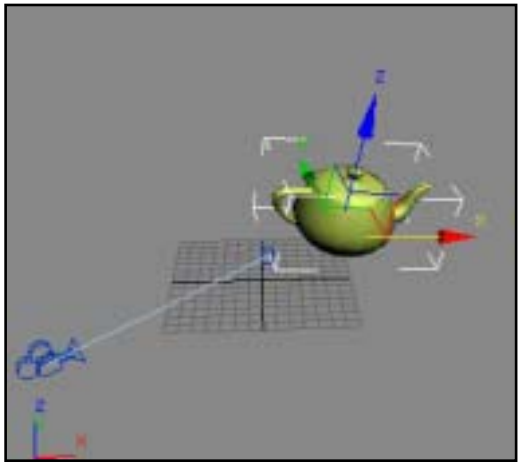
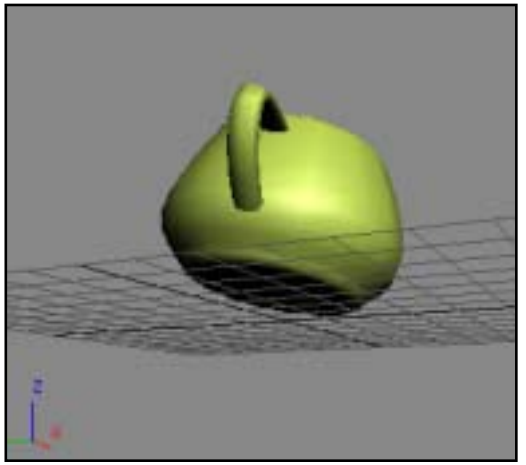
B. Rendering Pipeline

전형적인 렌더링 파이프라인



C. Transformation

입력된 좌표를 화면상의 좌표로 변환

1. 월드(World) 좌표계로 변환

• 일반적으로 입력 벡터는 정적이므로 행렬을 이용해서 월드 좌표계의 좌표로 변환

• V(1) = Vin * TMworld

2. 뷰(View) 좌표계로 변환

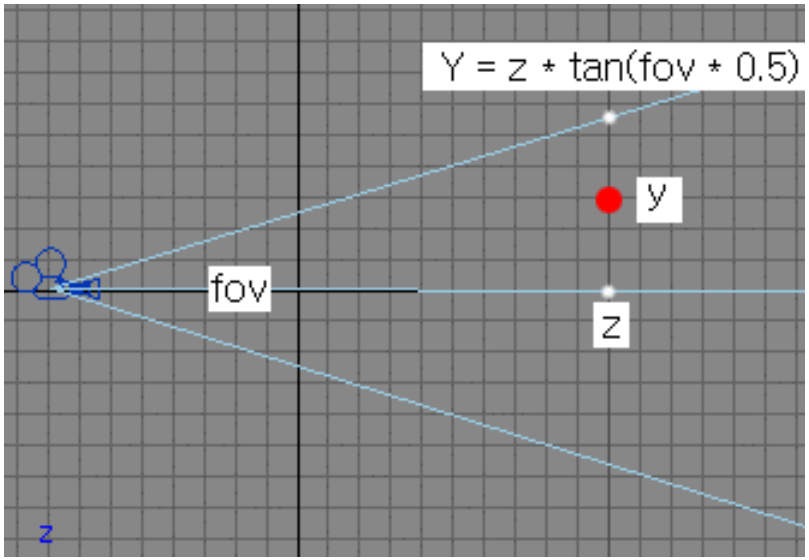
• 렌더링 되는 내용은 카메라 좌표계이므로 카메라 좌표계로 변환

• TMview = Inverse(TMcamera)

• V(2) = V(1) * TMview

3. 투영 변환

• 카메라 좌표계의 좌표를 화면에 투영



• x, y 는 -1 에서 1 사이로 매핑 시킴

• X’ = x * (1.0 / tan(fov * 0.5)) / z / aspect

• Y’ = y * (1.0 / tan(fov * 0.5)) / z

• fov가 카메라의 세로 fov 값인 경우 x 축의 fov를 구할 필요 없이 tan(fov *0.5)에 화면 가로 비율 / 화면 세로 비율을 곱한 값을 스케일 값으로 사용

• z 버퍼에 기입하는 z 값의 범위는 0 에서 1

• 직교 좌표계인 경우 Z’ = (z – Zn) / (Zf – Zn)

• 원근 좌표계에서 이 공식을 사용할 경우 수의 정밀도가 많이 떨어짐. (눈 앞의 1m 크기의 오브젝트 z 정밀도와 300m 뒤의 1m 크기의 정밀도가 같은 데, 300m 뒤의 오브젝트는 실제로 화면에 작게 보이고 찍을 픽셀이 적으므로 정밀도가 적어도 됨)

• 원근 좌표계에서는 z가 가까울 수록 정밀한 값을 얻기 위해 1/z 을 사용한다.

• 1/z 을 0 에서 1로 매핑하기 위해서

• Z’ = (1/z – 1/Zn) / (1/Zf – 1/Zn)

• 이 공식을 정리하면

• W’ = z

• X’ = x * (1.0f / tan(fov*0.5)) / aspect / W’

• Y’ = y * (1.0f / tan(fov*0.5)) / W’

• Z’ = Zn * Zf / (Zf – Zn) / W’ –
Zn / (Zf – Zn) * W’ / W’

• 변환 후 W’ 로 벡터를 나눈다고 가정하고 X, Y, Z 행렬을 구함

• 행렬로 전개함

• V(3) = V(2) * TMproj

• V(4) = V(3) / V(3).w

```
_MATRIX * _MatrixPerspectiveFovLH(_MATRIX *pOut, float fovY,
float aspect, float zn, float zf)
{
    float tanfov = (float) tan(fovY * 0.5f) ;
    pOut->_11 = (1.0f / tanfov) / aspect;
    pOut->_12 = 0.0f;
    pOut->_13 = 0.0f;
    pOut->_14 = 0.0f;
    pOut->_21 = 0.0f;
    pOut->_22 = (1.0f / tanfov);
    pOut->_23 = 0.0f;
    pOut->_24 = 0.0f;
    pOut->_31 = 0.0f;
    pOut->_32 = 0.0f;
    pOut->_33 = zf / (zf - zn);
    pOut->_34 = 1.0f;
    pOut->_41 = 0.0f;
    pOut->_42 = 0.0f;
    pOut->_43 = - zn * zf / (zf - zn);
    pOut->_44 = 0.0f;
    return pOut;
}
```

4. 뷰포트 매핑

• V(4)에서 구해진 x 의 값의 범위가 –1 에서 1 의 값이므로 이를 뷰포트의 위치와 크기에 맞춰서 스케일링함

• X’ = X * (Wvp * 0.5) + Wvp * 0.5 + Xvp

• Y’ = Y * (Hvp * 0.5) + Hvp * 0.5 + Yvp

• Z’ = Z

• 단 이 경우 V(4) 이후에 변환되므로 행렬의 결합 법칙을 사용할 수 없으므로

• X’ = X / W * (Wvp * 0.5) + (Wvp * 0.5 + Xvp) * W / W

• 로 수정해서 W값으로 나누기 이전의 행렬에 적용 하도록 한다

```
void CRenderer :: SetViewport(int x, int y, int w, int h)
{
    _MatrixIdentity(&m_Transform.vp);
    m_Transform.vp._11 = w * 0.5f;
    m_Transform.vp._41 = x + w * 0.5f;
    m_Transform.vp._22 = - h * 0.5f;
    m_Transform.vp._42 = y + h * 0.5f;
}
```

• y 축의 경우 화면의 위쪽이 0이 되도록 음수를 취한다.

• V(4) = V(3) * TMvp

• Vout = V(4) / V(4).w

• 최종 결과는 뷰 포트의 x, y 좌표와 Zn 에서 0 Zf 에서 1인 z 값

• 결합법칙에 의해

• TMtotal = Tmworld * TMcamera * Tmproj * TMvp

• Vout = Vin * TMtotal

• 의 최종 TM 매트릭스를 구할 수 있다.

```
void CRenderer :: SetTransform(_TMSTATETYPE state, const
_MATRIX & mat)
{
    switch(state)
    {
    case _VIEW :
        m_Transform.view = mat;
        break;
    case _PROJECTION :
        m_Transform.proj = mat;
        break;
    case _WORLD :
        m_Transform.world = mat;
        break;
    }
    _MATRIX temp[2];
    _MatrixMultiply(&temp[0], m_Transform.world,
                    m_Transform.view);
    _MatrixMultiply(&temp[1], temp[0], m_Transform.proj);
    _MatrixMultiply(&m_Transform.tm, temp[1],
                    m_Transform.vp);
}
```

• 내부적으로 사용하는 _VTX 형

```
struct _VTX
{
    float x, y, z, rhw;
    unsigned long color;
    float tu, tv;
    unsigned char clipcode;
} ;

const CRenderer::_VTX * CRenderer :: Transform(const
_VERTEX *v, int vn)
{
    static _VTX * vout = AllocVTX(vn);;
    int i;
    for(i=0; i<vn; i++)
    {
        float x, y, z, w, 곶;
```

```
        x = v[i].v.x * m_Transform.tm._11 + v[i].v.y *
m_Transform.tm._21 + v[i].v.z * m_Transform.tm._31 +
m_Transform.tm._41;
        y = v[i].v.x * m_Transform.tm._12 + v[i].v.y *
m_Transform.tm._22 + v[i].v.z * m_Transform.tm._32 +
m_Transform.tm._42;
        z = v[i].v.x * m_Transform.tm._13 + v[i].v.y *
m_Transform.tm._23 + v[i].v.z * m_Transform.tm._33 +
m_Transform.tm._43;
        w = v[i].v.x * m_Transform.tm._14 + v[i].v.y *
m_Transform.tm._24 + v[i].v.z * m_Transform.tm._34 +
m_Transform.tm._44;

        rhw = 1.0f / w;
        vout[i].x = x * rhw;
        vout[i].y = y * rhw;
        vout[i].z = z * rhw;
        vout[i].rhw = rhw;

        vout[i].tu = v[i].tu;
        vout[i].tv = v[i].tv;

        vout[i].color = v[i].diffuse;

        vout[i].clipcode = CalcClipcode(vout[i].x, vout[i].y,
vout[i].z);
    }

    return vout;
}
```

D. Culling

화면상에 보이는 폴리곤만 골라내는 일종의 컬링 처리

1. 백페이스컬링(Backface Culling)

• 모델링 시 겉에서 봤을 때의 페이스의 감는(wind) 방향을 정한 후 렌더링 시에 페이스의 버텍스 감긴 방향으로 겉에서 보는 앞면인지 반대쪽에서 보는 뒷면인지 판단한다.

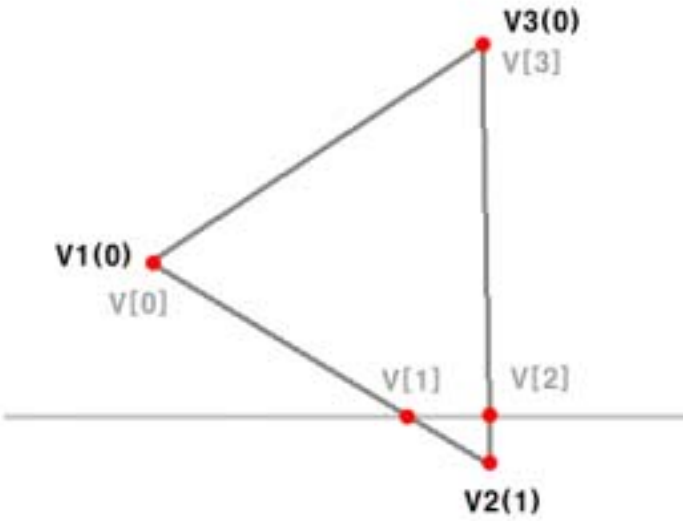
• 찍을 폴리곤의 변환된 화면상의 x, y 좌표를 이용해서 두 엣지의 크로스 프로덕트한 결과를 이용한다.

```
bool CRenderer :: ISCCW(const _VTX *v1, const _VTX *v2,
const _VTX *v3)
{
    return (v2->x-v1->x)*(v3->y-v1->y) –
            (v2->y-v1->y)*(v3->x-v1->x) < 0.0f ? true : false;
}
```

2. 클리핑 (Clipping)

• 레스터라이즈하는 단계에서는 클리핑을 신경 쓰지 않아도 되도록 화면안에 들어오는 버텍스로 만든다.

• Cohen-Sutherland 알고리즘을 이용하여 클리핑되는 지점의 버텍스를 만들어 준다.



• 프래그 값은 자르고자 하는 평면보다 안에 있으면 0, 밖에 있으면 1

• 폴리곤이 감긴 방향 V1에서 V3로 진행하면서 안에 있면 버텍스 유지, 만약 프래그가 0에서 1로, 혹은 1에서 0으로 바뀌는 (두 값을 ExclusiveOR 해서 1이 되는 경우) 경우에는 평면위의 교차 버텍스를 추가한다.

```
if ((v3->clipcode^v1->clipcode)&flags)
                LerpVTX(&v[n++], v3, v1, flags);
if (!(v1->clipcode&flags))
                v[n++] = *v1;

if ((v1->clipcode^v2->clipcode)&flags)
                LerpVTX(&v[n++], v1, v2, flags);
if (!(v2->clipcode&flags))
                v[n++] = *v2;

if ((v2->clipcode^v3->clipcode)&flags)
                LerpVTX(&v[n++], v2, v3, flags);
if (!(v3->clipcode&flags))
                v[n++] = *v3;
```

• 클리핑된 버텍스에 대해서 다른 평면에 대해서도 클리핑을 계속한다.

```
enum _CLIPBIT {
      _C_NEAR = 0x01,
      _C_FAR = 0x02,
      _C_LEFT = 0x04,
      _C_RIGHT = 0x08,
      _C_TOP = 0x10,
      _C_BOTTOM = 0x20,
   } ;
```

```
unsigned char CalcClipcode(float x, float y, float z)
{
   unsigned char ccode = 0;

   if (z < m_Viewport.nz)
      ccode |= _C_NEAR;
   else
   if (z > m_Viewport.fz)
      ccode |= _C_FAR;

   if (x < m_Viewport.x)
      ccode |= _C_LEFT;
   else
   if (x >= m_Viewport.x + m_Viewport.w)
      ccode |= _C_RIGHT;

   if (y < m_Viewport.y)
      ccode |= _C_TOP;
   else
   if (y >= m_Viewport.y + m_Viewport.h)
      ccode |= _C_BOTTOM;
   return ccode;
}
```

• 6면에 대해서 클리핑을 한다.

```
void ClipAndDrawTriangle(const _VTX *v1, const _VTX *v2,
                         const _VTX *v3, unsigned char mask)
{
   int i, flags, n=0;
   _VTX v[6];
   if (mask == 0) {
      ....
   }   else
   {
      for(flags=1; flags<(1<<6); flags<<=1) {
         if (mask&flags) {
            if ((v3->clipcode^v1->clipcode)&flags)
               LerpVTX(&v[n++], v3, v1, flags);
            if (!(v1->clipcode&flags))
               v[n++] = *v1;
            if ((v1->clipcode^v2->clipcode)&flags)
               LerpVTX(&v[n++], v1, v2, flags);
            if (!(v2->clipcode&flags))
               v[n++] = *v2;
            if ((v2->clipcode^v3->clipcode)&flags)
               LerpVTX(&v[n++], v2, v3, flags);
            if (!(v3->clipcode&flags))
               v[n++] = *v3;
            for(i=2; i<n; i++)
               ClipAndDrawTriangle(&v[0], &v[i-1], &v[i],
                                   mask^flags);
            return ;
         }
      }
   }
}
```

• 클리핑 => 투영한 폴리곤의 버텍스를 (m_Viewport.x, m_Viewport.y, 0) 에서 (m_Viewport.x+m_Viewport.w, m_Viewport.y+m_Viewport.h, 1) 사이의 정육면체안에 들어오도록 한다.

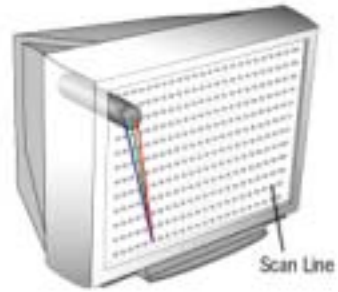
```
void LerpVTX(_VTX * vout, const _VTX * v1, const _VTX *v2,
unsigned char clipbit)
{
    float t = 1.0f;
    switch(clipbit)
    {
        case _C_NEAR :
            t = (m_Viewport.nz - v1->z) / (v2->z - v1->z);
            break;
        case _C_FAR :
            t = (m_Viewport.fz - v1->z) / (v2->z - v1->z);
            break;
        case _C_LEFT :
            t = (m_Viewport.x - v1->x) / (v2->x - v1->x);
            break;
        case _C_RIGHT :
            t = ((m_Viewport.x + m_Viewport.w - 1) - v1->x) /
                (v2->x - v1->x);
            break;
 case _C_TOP :
            t = (m_Viewport.y - v1->y) / (v2->y - v1->y);
            break;
        case _C_BOTTOM :
            t = ((m_Viewport.y + m_Viewport.h - 1) - v1->y) /
                (v2->y - v1->y);
            break;
    }

    vout->x = Lerp(v1->x, v2->x, t);
    vout->y = Lerp(v1->y, v2->y, t);
    vout->z = Lerp(v1->z, v2->z, t);
    vout->rhw = Lerp(v1->rhw, v2->rhw, t);
    vout->color = LerpColor(v1->color, v2->color, t);
    vout->tu = Lerp(v1->tu, v2->tu, t);
    vout->tv = Lerp(v1->tv, v2->tv, t);
    vout->clipcode = CalcClipcode(vout->x, vout->y, vout->z);
}
```

E. Rasterization

실제 화면에 그리는 단계

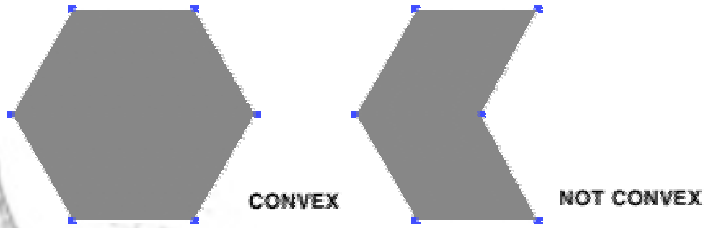
1. 레스터라이제이션 (Rasterization)

• 레스터라이제이션이란

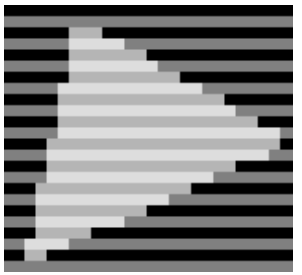


• 레스터라이제이션에 적합한 기하형태는 볼록한 폴리곤 (convex polygon)

• 볼록(convex)의 조건 – 임의의 평면(혹은 선)으로 나누었을 때 항상 두개로 나뉜다

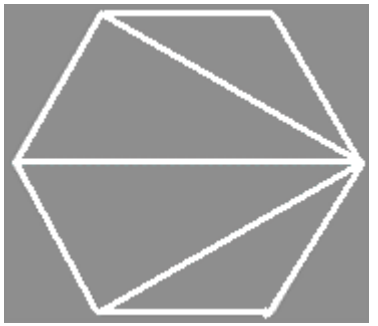


• 즉, 화면의 세로방향으로 가로 라인의 교차점을 렌더링하여 도형을 렌더링 할 수 있다.



2. 삼각형 (Triangle)

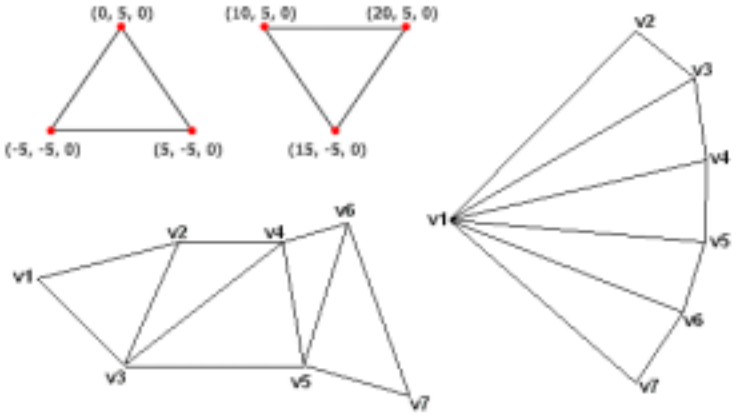
• 삼각형(Triangle)은 3개의 정점(Vertex)과 3개의 엣지(Edge)를 가지는 기하 도형으로 Convex 한 다각형(Polygon)은 삼각형으로 조합 가능하다.



• 삼각형은 레스터라이징 최소 단위로 적합하다.

3. 프리미티브 타입

• 폴리곤의 타입에 따라 삼각형으로 분리한다.




```
void DrawPrimitive(_PRIMITIVE p, const _VERTEX *v, int pn)
{
    const _VTX * vtx;
    int i;
    switch(p)
    {
    case _TRIANGLELIST :
        vtx = Transform(v, pn*3);
        for(i=0; i<pn; i++)
            DrawTriangle(&vtx[i*3], &vtx[i*3+1], &vtx[i*3+2]);
        break;
    case _TRIANGLESTRIP :
        vtx = Transform(v, pn+2);
        for(i=0; i<pn; i++) {
            if ((i&1) == 0)
                DrawTriangle(&vtx[i], &vtx[i+1], &vtx[i+2]);
            else
                DrawTriangle(&vtx[i+2], &vtx[i+1], &vtx[i+0]);
        }
        break;
    case _TRIANGLEFAN :
        vtx = Transform(v, pn+2);
        for(i=0; i<pn; i++)
            DrawTriangle(&vtx[0], &vtx[i+1], &vtx[i+2]);
        break;
    }
}
```

4. 선형 보간 (Linear Interpolation)

•기울기가 일정하게 변하는 두 값사이의 값을 결정하는 것

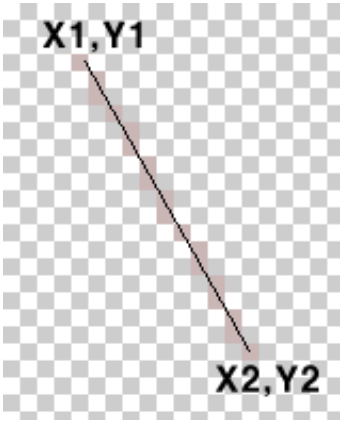
• $F(t)=F1*(1-t) + F2*t = F1 + (F2-F1)*t$

```
#include <stdio.h>
void Interpolation(int x1, int x2, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", x1 + (x2-x1) * i / (len-1));
}
void main()
{
    Interpolation(5, 20, 20);
}
```

5. 라인(Line) 그리기

• 간단한 선형을 보간을 이용한 라인 그리기

• y 값을 증가하면서 x의 위치를 선형보간으로 구한다



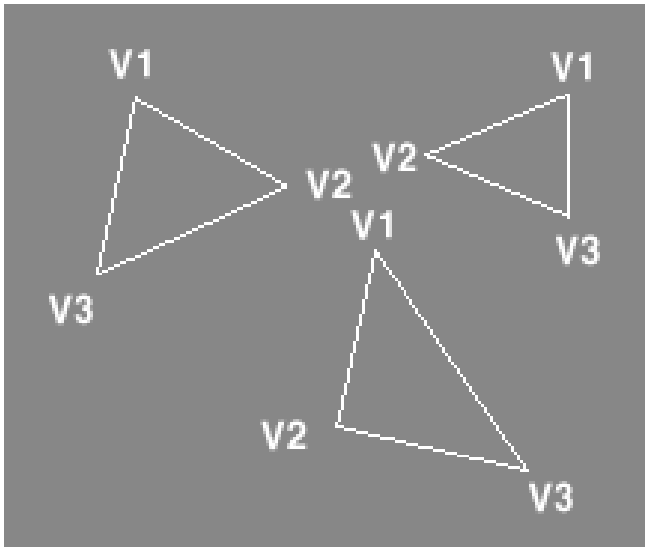

```
void DrawLine(int x1, int y1, int x2, int y2, unsigned long color)
{
    if (y1 > y2)
        DrawLine(x2, y2, x1, y1, color);
    else
        {
            int x, y;
            for(y=y1; y<=y2; y++)
            {
                x = x1 + (int)((x2 – x1) * (y-y1+0.5f) / (y2-y1+1));
                PutPixel(x, y, color);
            }
        }
}
```

• (단, x 축의 길이가 더 길 때는 x를 작은 값에서 큰 값으로 증가시키면서 각 픽셀의 y를 구해서 찍는다)

6. 삼각형 그리기

• 삼각형의 구성 요소 : 버텍스 3개, 엣지 3개

• y의 크기로 정렬했을 경우 형태와 상관없이 엣지는 v1->v2->v3 부분과 v1->v3 로 나눌 수 있다.



• 중간 V2 라인을 중심으로 두 부분으로 나눌 수 있다.

• y 를 V1 에서 V3 로 이동하면서 각 라인의 양 엣지의 각 x를 구해서 엣지를 채운다.

• y 값을 증가하면서 x의 위치를 선형 보간으로 구한다

```
void DrawTriangle(HDC hdc, const _VTX *v1, const _VTX *v2,
const _VTX *v3)
{
    if (v1->y > v2->y) {
        DrawTriangle(hdc, v2, v1, v3);
    }   else
    if (v1->y > v3->y) {
        DrawTriangle(hdc, v3, v2, v1);
    }   else
    if (v2->y > v3->y) {
        DrawTriangle(hdc, v1, v3, v2);
    }   else {
        int y;
        int x1, x2;
        float t;

        for(y=v1->y; y<v2->y; y++) {
            t = (y - v1->y + 0.5f) / (v2->y - v1->y);
            x1 = v1->x + (int)((v2->x - v1->x) * t);

            t = (y - v1->y + 0.5f) / (v3->y - v1->y + 1);
            x2 = v1->x + (int)((v3->x - v1->x) * t);

            DrawLine(hdc, x1, x2, y);
        }
```

```
        for(; y<=v3->y; y++) {
            t = (y - v2->y + 0.5f) / (v3->y - v2->y + 1);
            x1 = v2->x + (int)((v3->x - v2->x) * t);

            t = (y - v1->y + 0.5f) / (v3->y - v1->y + 1);
            x2 = v1->x + (int)((v3->x - v1->x) * t);

            DrawLine(hdc, x1, x2, y);
        }
    }
}
```

• 먼저 y 값의 순서대로 v1, v2, v3를 정렬한다.

• 상위 부분은 v2 까지 v1->v2 로, v1->v3 의 엣지에 대해서 x값을 구하며, 하위 부분은 v2->v3 로, v1->v3 의 엣지에 대해서 x값을 구한다

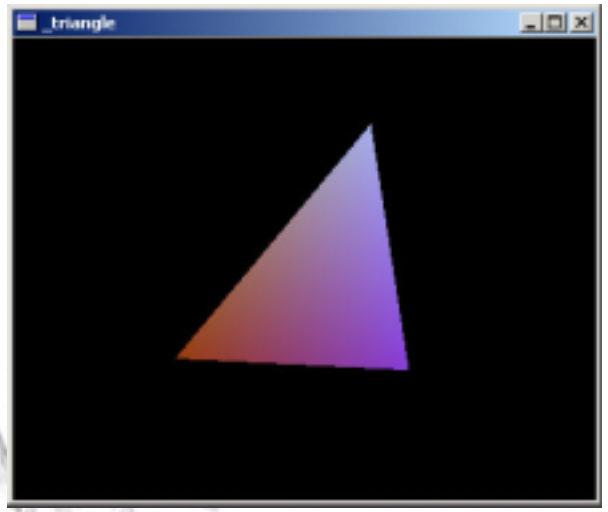
7. 버텍스 칼라 블랜딩

• 양 엣지에 r, g, b 값을 선형 보간한다

• 가로 렌더링 시 엣지의 보간된 rgb를 x에 따라 마찬가지로 보간한다

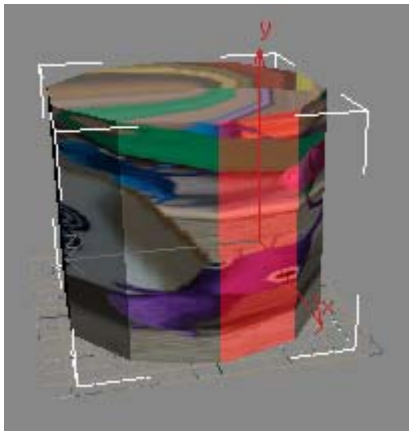
```
#define _R(C) (float)((C>>16)&0xFF)
#define _G(C) (float)((C>>8)&0xFF)
#define _B(C) (float)(C&0xFF)

struct _EDGE {
    int x;
    float r, g, b;
} ;

void DrawTriangle(HDC hdc, const _VTX *v1, const _VTX *v2,
const _VTX *v3)
{
    if (v1->y > v2->y) {
        DrawTriangle(hdc, v2, v1, v3);
    }   else
    if (v1->y > v3->y) {
        DrawTriangle(hdc, v3, v2, v1);
    }   else
    if (v2->y > v3->y) {
        DrawTriangle(hdc, v1, v3, v2);
    }   else {
        int y;
        _EDGE e1, e2;
        float t;
        for(y=v1->y; y<v2->y; y++) {
            t = (y - v1->y + 0.5f) / (v2->y - v1->y);
            e1.x = Lerp(v1->x, v2->x, t);
            e1.r = Lerp(_R(v1->color), _R(v2->color), t);
            e1.g = Lerp(_G(v1->color), _G(v2->color), t);
            e1.b = Lerp(_B(v1->color), _B(v2->color), t);
            t = (y - v1->y + 0.5f) / (v3->y - v1->y + 1);
            e2.x = Lerp(v1->x, v3->x, t);
            e2.r = Lerp(_R(v1->color), _R(v3->color), t);
            e2.g = Lerp(_G(v1->color), _G(v3->color), t);
            e2.b = Lerp(_B(v1->color), _B(v3->color), t);

            DrawLine(hdc, &e1, &e2, y);
        }
        for(; y<=v3->y; y++) {
            t = (y - v2->y + 0.5f) / (v3->y - v2->y + 1);
            e1.x = Lerp(v2->x, v3->x, t);
            e1.r = Lerp(_R(v2->color), _R(v3->color), t);
            e1.g = Lerp(_G(v2->color), _G(v3->color), t);
            e1.b = Lerp(_B(v2->color), _B(v3->color), t);
            t = (y - v1->y + 0.5f) / (v3->y - v1->y + 1);
            e2.x = Lerp(v1->x, v3->x, t);
            e2.r = Lerp(_R(v1->color), _R(v3->color), t);
            e2.g = Lerp(_G(v1->color), _G(v3->color), t);
            e2.b = Lerp(_B(v1->color), _B(v3->color), t);

            DrawLine(hdc, &e1, &e2, y);
        }
    }
}
```

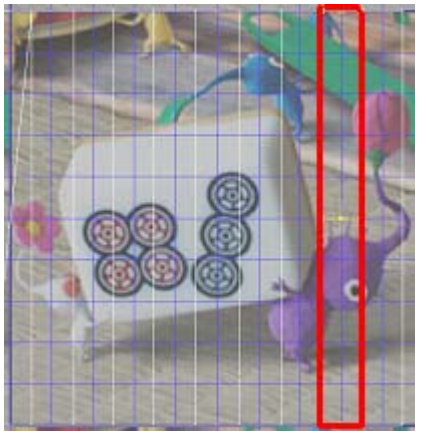
```
void DrawLine(HDC hdc, const _EDGE *e1, const _EDGE *e2,
int y)
{
    if (e1->x > e2->x) {
        DrawLine(hdc, e2, e1, y);
    }   else {
        int x;
        float r, g, b;
        for(x=e1->x; x<=e2->x; x++) {
            float t = (x - e1->x + 0.5f) / (e2->x - e1->x + 1);
            r = Lerp(e1->r, e2->r, t);
            g = Lerp(e1->g, e2->g, t);
            b = Lerp(e1->b, e2->b, t);
            ::SetPixel(hdc, x, y, RGB((int)r, (int)g, (int)b));
        }
    }
}
```



8. 텍스처 렌더링

• 각 폴리곤 별로 매핑시킬 텍스처의 영역을 설정한다



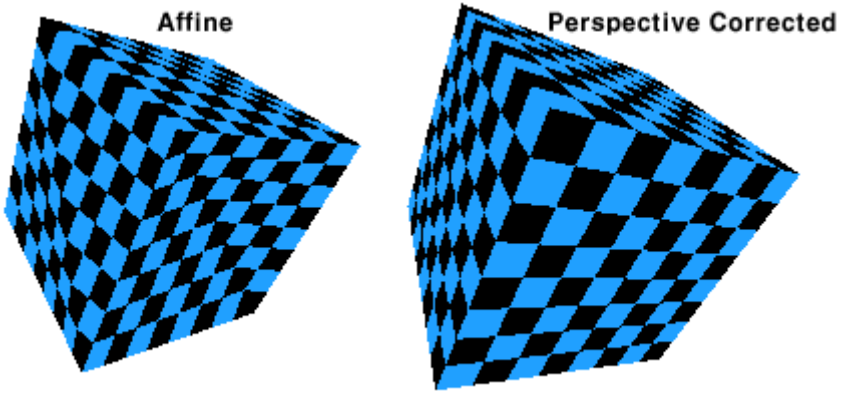


• 버텍스에 텍스처의 u, v 좌표가 저장되어 있다.

• 이미지 종속적이지 않기 위해 u, v 는 각 0 에서 1의 크기로 매핑한다.

• 칼라와 마찬가지로 각 엣지와 픽셀의 u, v를 보간하며, 픽셀 렌더링 시에는 u, v 좌표를 이용해서 텍스처의 칼라값을 읽어온다

• x, y는 z에 의해 원근 투영 되었기 때문에 u, v를 원근보정 (Perspective Correction)해주지 않으면 텍스처가 외곡 되어 보인다.



http://www.lysator.liu.se/~mikaelk/doc/perspectivetexture/

• FOV가 클 수록 영향을 많이 받는다.

• 원근 보정을 하기 위해 u, v에 대해서는 아래와 같은 보간 공식을 사용한다

• $U(t) = (U1 / Z1 * (1-t) + U2 / Z2 * t) / ((1 / Z1) * (1-t) + (1 / Z2) * t)$

• 버텍스의 1/Z 인 RHW 값을 활용한다

9. 텍스처 블랜딩

• 텍스처의 u, v 로 읽을 때

• Point 셈플링 : 가장 근접한 칼라

• T(u, v) = GetTexel(U * W, V * H)

```
unsigned long ReadTexel(_TEXTURE * tex, int x, int y)
{
    unsigned long c = 0xFF000000;
    x %= tex->w;
    y %= tex->h;
    memcpy(&c, tex->image + y * tex->pitch + x * 3, 3);
    return c;
}
```


• Bilinear Filtering : 인접한 텍셀과 가중치(접한 면적)을 고려한 방식

• u, v 가 걸친 4개의 각 텍셀의 값과 걸친 면적을 곱한 값을 렌더링한다

```
unsigned long ReadTexel(_TEXTURE * tex, float u, float v)
{
    int itu, itv;

    itu = (int)(u * tex->w);
    itv = (int)(v * tex->h);

#ifndef _BILINEAR_FILTERING
    return ReadTexel(tex, itu, itv);
#else
    float wu, wv, w[2][2], r, g, b;
    unsigned long c[2][2];

    wu = (float) fmod(u * tex->w, 1.0f);
    wv = (float) fmod(v * tex->h, 1.0f);

    w[0][0] = (1.0f - wu) * (1.0f - wv);
    w[0][1] = wu * (1.0f - wv);
    w[1][0] = (1.0f - wu) * wv;
    w[1][1] = wu * wv;

    c[0][0] = ReadTexel(tex, itu, itv);
    c[0][1] = ReadTexel(tex, itu + 1, itv);
    c[1][0] = ReadTexel(tex, itu, itv + 1);
    c[1][1] = ReadTexel(tex, itu + 1, itv + 1);

    r = _UNPACK_R(c[0][0]) * w[0][0] +
        _UNPACK_R(c[0][1]) * w[0][1] +
        _UNPACK_R(c[1][0]) * w[1][0] +
        _UNPACK_R(c[1][1]) * w[1][1];
    g = _UNPACK_G(c[0][0]) * w[0][0] +
        _UNPACK_G(c[0][1]) * w[0][1] +
        _UNPACK_G(c[1][0]) * w[1][0] +
        _UNPACK_G(c[1][1]) * w[1][1];
    b = _UNPACK_B(c[0][0]) * w[0][0] +
        _UNPACK_B(c[0][1]) * w[0][1] +
        _UNPACK_B(c[1][0]) * w[1][0] +
        _UNPACK_B(c[1][1]) * w[1][1];

    return _PACK((int)r, (int)g, (int)b, 0xFF);
#endif
}
```

- 밉맵(Mip Map)
  화면의 텍셀 비율로 텍스처 레벨 결정
- 트라이 리니어 필터링 (Trilinear Filtering)
  바이리니어 필터링 + 밉맵

10. Z 버퍼링

• 프레임 버퍼와 1:1 매칭되는 Z 버퍼를 가지고 있음

• 찍으려고 하는 위치의 z 값과 찍으려는 z 값을 비교해서 연산 조건에 만족하는 경우에만 렌더링

• 가장 최종적인 컬링 기법

• Z 버퍼링 연산 조건, Z값을 쓰는 지 세팅

```
void CRasterizer :: RasterLine(int y, _EDGE * l, _EDGE * r, ...)
{
    if (l->x > r->x)
        RasterLine(y, r, l, tex);
    else {
        int x, ix1, ix2;
        float t;
        ix1 = (int)(l->x);
        ix2 = (int)(r->x);
        for(x=ix1; x<=ix2; x++) {
            t = (x - ix1 + 0.5f) / (ix2 - ix1 + 1);
            z = Lerp(l->z, r->z, t);
            if (m_Zbuffer.pBuffer[y * m_Zbuffer.w + x] >= z) {
                m_Zbuffer.pBuffer[y * m_Zbuffer.w + x] = z;
                ...
            }
        }
    }
```

11. 알파 블랜딩 / 테스팅

• 찍고자 하는 칼라값와 프레임 버퍼의 칼라값을 블랜딩함

• Cnew = Ctag * (1 – alpha) + Csrc * alpha

• 전형적인 알파 블랜딩 공식

• 알파 테스팅 - 알파 값으로 화면에 실제로 찍는 지 안 찍는지를 결정

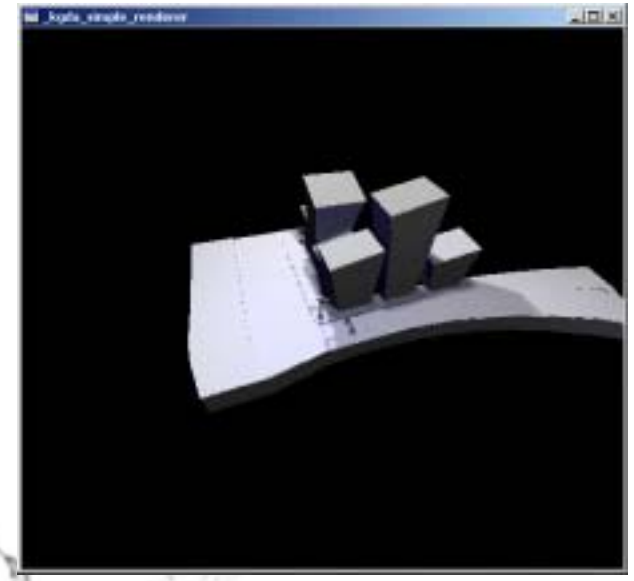
• 기타 프레임 버퍼와의 연산 공식을 이용해서 가산, 감사, 곱등의 원하는 방식으로 블랜딩

```
int rgb[4];

if (rgb[3] > _ALPHATEST) {
    if (rgb[3] < 0xFF) {
        unsigned long framc = m_DIB.GetPixel(x, y);
        t = (float)rgb[3] / 0xFF;
        rgb[0] = (int) Lerp(rgb[0], _UNPACK_R(framc), t);
        rgb[1] = (int) Lerp(rgb[1], _UNPACK_G(framc), t);
        rgb[2] = (int) Lerp(rgb[2], _UNPACK_B(framc), t);
    }

    m_DIB.PutPixel(x, y, _PACK(rgb[0], rgb[1], rgb[2], 0));
}
```

F. 기타

• 소프트 렌더러 제작시 최적화 포인트
  고정 소수점
  연산 최적화
  루프 최적화
  메모리 최적화

• 셈플 코드
http://digibath.com/noerror/download/kgds2004_renderpipe.zip (156k)



참고 도서

• RealTime Rendering 2nd ed (토마스 뮐러, 역, 정보문화사)

• Direct 3d Shader X (울프강 엥겔 외, 역, 정보문화사)

• Mathematics for 3d Game Programming & Computer Graphics (Erin Lengyel, 역, Charles River Media)

• Linear Algebra with applications (Gareth Williams, 역, 사이텍미디어)

• The Art of 3-d Computer Animation And Imaging (Issac Victor Kerlow, 역, Wiley)

G. 질문 / 답