

게임 엔진(2) 활용팁

김성익(noerror@hitel.net)
2006.4.30

Spirit of Flame
3D RealTime Graphics Programming Study

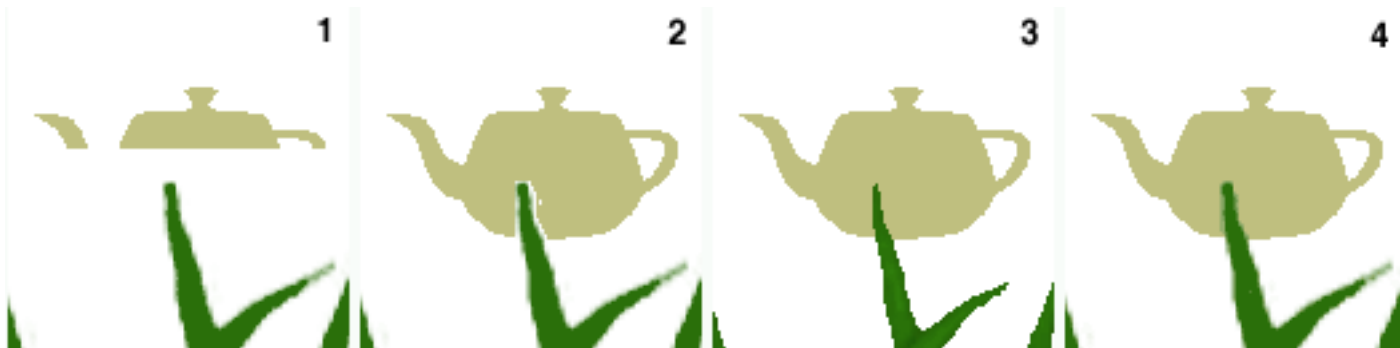
Kasa

개요

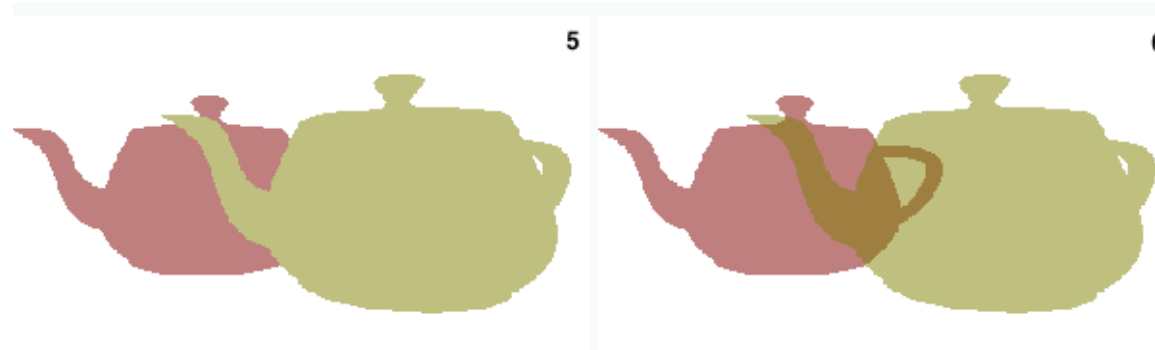
- 게임 엔진을 구성하는 데 고려해 볼 것들
- 몇 가지 처리 방식 제안

알파 메시 렌더링 정책(1)

- 알파 텍스처를 쓴 메시나, 버텍스 알파를 사용하는 메시는 나중에 그린다.



- 알파 메시에 대해서는 멀리 있는 것부터 가까운 메시 순으로 렌더링한다. .



알파 메시 렌더링 정책(2)



- 투명한 메시(혹은 오브젝트)에 대해선 겹치지 않도록 2pass 렌더링한다.

DrawPrimitive 정책(1)

- **최적화 전략**
DrawPrimitive 를 적게 함
속도상의 이득이 있는 순서대로 그림
- **제안 : DrawPrimitive 시 실제로**
DrawPrimitive 하지 않고, 리스트에 추가함,
EndScene 호출시나, 순서를 맞춰야 하는
경우 Flush 함

DrawPrimitive 정책(2)

- Flush 시 소팅해서 찍음
동일한 텍스처끼리
동일한 렌더링 스테이트끼리
렌더링 스테이트가 적게 바뀌도록
- 실제 DrawPrimitive 시
찍은 메시와 다음에 찍을 메시가 동일하고 렌더링스태이트, 텍스처까지 동일하며, 찍을 메시의 버텍스 수와 인덱스 수가 적다면 머지해서 찍는다. (버텍스 수가 적어서 머지할만한 메시는 버텍스 정보과 인덱스 정보를 저장해둠)

DrawPrimitive 정책(3)

- 머지해서 찍을 때는 가능한 상황이라면 D3DLOCK_DISCARD 없이 D3DLOCK_NOOVERWRITE로 LOCK

```
int UploadVertexBuffer32(void * buffer, int len, IDirect3DVertexBuffer9 ** vptr, int align)
{
    HRESULT hr;

    unsigned long dwLockFlags = D3DLOCK_NOOVERWRITE;
    unsigned char * pVptr;
    int offset;

    if (len + m_iVOffset + align > m_iVLength)
    {
        dwLockFlags = D3DLOCK_DISCARD;
        m_iVOffset = 0;
        m_pCurVBufLockable = m_pVBufLockable[m_pCurVBufLockable == m_pVBufLockable[0] ? 1 : 0];
    }

    int paddbytes = (align - (m_iVOffset%align)) % align;

    offset = m_iVOffset + paddbytes;

    hr = m_pCurVBufLockable->Lock(m_iVOffset, len + paddbytes, (void**)&pVptr, dwLockFlags);
    _ASSERT(hr == D3D_OK);

    memcpy(pVptr + paddbytes, buffer, len);
    m_pCurVBufLockable->Unlock();

    *vptr = m_pCurVBufLockable;
    m_iVOffset = offset + len;

    return offset / align;
}
```

DrawPrimitive 정책(4)

- 엔진 사용하는 레벨에서도 어느 정도 조절을 해준다면 별다른 처리 없이 성능 향상이 가능한 부분도 존재
- 예를 들어 UI를 찍을 경우 동일한 텍스처 위주로 DrawPrimitive를 한다면, 실제 머지 되기 때문에 적은 횟수로 DrawPrimitive 가 됨 - 내부적으로 적은 수의 DrawPrimitive를 위해서 최적의 UI나 파티클 출력 내용물의 버텍스 리스트와 인덱스 리스트를 만들 필요가 적어짐
- 셰이더 상수 관련

렌더스테이트관리(1)

- 렌더링시 큰 영향을 미침
- 상태 변수가 많아서 관리가 어려움
- 관리가 안될 경우 오브젝트 렌더링 순서에 따라 렌더링 상태가 바뀌는 괴 현상 생김

렌더스테이트관리(2)

```
void DrawCharacterDecal(obj)
{
    g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
    g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
    g_pd3dDevice->SetRenderState(D3DRS_ZFUNC, D3DCMP_EQUAL);
    obj->DrawSubset(0);
}

void DrawCharacter(obj)
{
    g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
    obj->DrawSubset(0);

    for(i=0; i<visible.size(); i++)
    {
        DrawCharacter(visible[i]);
        DrawCharacterDecal(visible[i])
    }
}
```

- DrawCharacter 에 SetRenderState(D3DRS_ZFUNC, D3DCMP_LESSEQUAL); 를 추가하면 문제가 안 생기지만, 추후에 도 단 하나의 렌더링 루틴에서 새로운 State 를 사용할 경우 모든 렌더링 루틴에서 그 상태에 대한 처리를 해주어야 함

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

렌더스테이트관리(3)

- 제안 : 비트 연산을 통해서 하나의 프래그에 해당 정보를 저장한다.

```
#define _ALPHABLEND          0 // 기본값:생략가능
#define _NOALPHABLEND      1
#define _ZFUNC_LESSQUAL    0
#define _ZFUNC_EQUAL       2

void SetRenderState(int flags)
{
    if ((flags & (_ALPHABLEND | _NOALPHABLEND)) == _ALPHABLEND)
    {
        g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
    }
    else
    {
        g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
        g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
    }

    if ((flags & (_ZFUNC_LESSQUAL | _ZFUNC_EQUAL)) == _ZFUNC_EQUAL)
    {
        g_pd3dDevice->SetRenderState(D3DRS_ZFUNC, D3DCMP_EQUAL);
    }
    else
    {
        g_pd3dDevice->SetRenderState(D3DRS_ZFUNC, D3DCMP_LESSEQUAL);
    }
}
```

렌더스태이트관리(4)

```
void DrawCharacterDecal(obj)
{
    SetRenderState(_NOALPHABLEND|_ZFUNC_EQUAL);
    obj->DrawSubset(0);
}

void DrawCharacter(obj)
{
    SetRenderState(0);
    obj->DrawSubset(0);
}
```

- 렌더링 스테이트를 하나의 데이터로 관리가 가능해지므로 관리가 쉽고, 직관적으로 한 눈에 상태를 알 수 있음.
- 상태의 데이터가 적기 때문에 오브젝트를 렌더링 스테이트 별로 소팅시 비교가 용이함

렌더스테이트관리(5)

- 특별히 세팅을 전혀 안할 경우 0가 되기 때문에 나중에 하나의 오브젝트에서만 렌더링시 CULL 방향을 수정한다고 해도 문제 없이 추가 가능함

```
#define _CULL_CCW 0
#define _CULL_NONE 4

void SetRenderState(int flags)
{
    ...
    if ((flags & (_CULL_CCW | _CULL_NONE)) == _CULL_CCW)
    {
        g_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    }
    else
    {
        g_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
    }
}
```

렌더스테이트관리(6)

- 해당 값이 비트 단위로 저장되기 때문에 간단한 비트연산으로 RenderState 함수 호출을 줄일 수 있음 (중복된 상태 변환이 없어짐)

```
void SetRenderState(int flags)
{
    int changebit = flags ^ g_iCurRenderState;

    if (changebit == 0)
        return ;

    g_iCurRenderState = flags;

    ...

    if (changebit & (_CULL_CCW | _CULL_NONE))
    {
        if ((flags & (_CULL_CCW | _CULL_NONE)) == _CULL_CCW)
        {
            g_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
        }
        else
        {
            g_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
        }
    }
}
```

- SetRenderState 함수 호출의 최소화 : 렌더스테이트 별로 소팅 후 렌더링

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

부동 소수점 수 압축

- 애니메이션의 키 프레임 값 등 정밀도가 조금 떨어져도 되며 메모리를 많이 차지하는 float을 압축해서 메모리 사용 최적화로 성능 향상

```
// SEEE EMMM MMMM MMMM
unsigned short fp32_16(float t)
{
    int s = (((unsigned long*)&t) & 0x80000000) >> 31;
    int e = (((unsigned long*)&t) & 0x7F800000) >> 23 - 127;
    int m = (((unsigned long*)&t) & 0x007FFFFF);

    if (e < -7)
        return 0;

    return (s << 15) | ((e + 7) << 11) | (m >> (23 - 11));
}

float fp16_32(unsigned short t)
{
    int s = ((t & 0x8000) >> 15);
    int e = ((t & 0x7800) >> 11) - 7;
    int m = (t & 0x007FFF);

    if (t == 0)
        return 0.0f;

    unsigned long f = (s << 31) | ((e + 127) << 23) | (m << (23 - 11));
    return *((float*)&f);
}
```

of Flame
ime Graphics Programming Study

Kasa

SSE를 활용

- **SSE의 특징**
부동 소수점 곱하기를 동시에 여러 개를 한다
연산 속도는 증가되지만 레지스터간 이동(형 변환)의 부담을 고려하지 않으면 속도 향상이 적거나 없다
메모리는 16 바이트 align 되어야 한다
- 데이터 형식 변환 등의 적극적인 최적화가 필요하다

이벤트 객체를 이용한 동기화 (1)

- **Mutex, CriticalSection 등과 Event 객체의 차이점 : Event는 수동으로도 동기 상태를 제어할 수 있다.**

```
HANDLE m_hLoading = CreateEvent(NULL, TRUE, FALSE, NULL);
```

```
Thread1:  
void Render()  
{  
    WaitForSingleObject(m_hLoading, INFINITE);  
    m_pModel->Render();  
}
```

```
Thread2:  
void LoadThread()  
{  
    m_pModel = LoadModel(m_szFilename);  
    SetEvent(m_hLoading);  
}
```

- **백그라운드로 로딩하도록 처리 때 미처 로딩이 마치기도 전에 사용을 요청한다면 그 동작을 우선 로딩하여 해당 이벤트를 Set 해주면 사용 스레드는 우아하게 하던 일 계속함 (단, 로딩이 긴 것보다는 게임 중간에 약간 밀리는 일이 있더라도 로딩이 짧은 것이 좋다는 가정.)**

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

이벤트 객체를 이용한 동기화 (2)

- 엔진 레벨에서 멀티 쓰레드를 잘 활용한다면 쾌적한 게임 구현에 도움이 됨

- 만약 로딩만 하는 쓰레드를 운영한다면

```
void LoadThread()  
{  
    while(1)  
    {  
        if (HasItem())  
            LoadItem();  
    }  
}
```

- 이처럼 구성하면 로딩 안 할 때도 과부하가 생김

이벤트 객체를 이용한 동기화 (3)

- 강제로 Sleep 루틴을 넣어서 로딩이 없을 때는 100ms 에 한번씩 루프에 오도록 수정됨. 하지만 로딩시 Sleep 중이면 최대 100ms 의 지연이 생김

```
void LoadThread()  
{  
    while(1)  
    {  
        if (HasItem())  
            LoadItem();  
        else  
            Sleep(100);  
    }  
}
```

- 아이템 추가하고 SetEvent 해주면 LoadThread 가 바로 깨어나서 우아하게 로딩을 시작함. (ResetEvent(m>Loading) 이벤트 부분과 아이템 추가시 SetEvent 부분은 동기화가 고려 안된 예임.)

```
void LoadThread()  
{  
    while(1)  
    {  
        WaitForSingleObject(m_hLoading, INFINITE);  
  
        while(HasItem())  
            LoadItem();  
  
        ResetEvent(m_hLoading);  
    }  
}
```

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

질문/의견

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa