

게임 엔진 (1)

설계 요소

김성익(noerror@hitel.net)
2006.02.05

개요

- 게임 엔진 개념 정리
- 엔진 구성 예제
- 엔진 설계 포인트
- 엔진 인터페이스 디자인 요소와 사례

Note. 이번 발표 내용은 개인적인 의견 중심으로 정리한 내용으로 일반적으로 인정되는 내용과 다를 수 있습니다.

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

엔진이란 ?

- **기능적인 관점**
특정 기능을 구현
- **목적**
특별히 어플리케이션에서 기능 구현 없이 사용하도록 설계된 것
- *단순히 wrap한 인터페이스를 제공하는 것도 엔진 범주에 속한다고 봐야*

게임엔진

- 게임 개발에 특화된 형태의 엔진

그래픽 엔진 *GameBryo, RenderWare, OGRE, Jupiter*
2d엔진, 장면 관리 엔진 *dPVS*

캐릭터 엔진 *CAL3D, Granny3D, EMotionFX*

사운드 엔진 *Miles, fmod*, 물리 엔진 *NovodeX, Havok*

인풋 엔진, 네트워크 엔진 *RakNet*, 길 찾기 엔진

스크립트 엔진, 인공지능 엔진, 기타 등등등

(다수의 기능을 혼재하는 경우도 *Torque*, 게임의 로직
까지도 일부 구현하는 경우도 *Unreal, SeriousSam*)

게임 엔진 지향점

- 게임 구현에 특화 된 기능
- 필요에 따라 기능을 확장가능 (*그렇지 않아도 상관은 없어보임*)
- 다양한 게임, 다양한 플랫폼에 적용 가능 (*그렇지 않아도 상관은 없어보임*)
- 엔진의 기능 구현 알고리즘을 전혀(?) 모르더라도 쉽게 개발 가능

파일 IO 엔진 구성 예

- 목표: 간단한 파일 로더를 엔진으로 구성
- 아래 예제처럼 사용할 목적의 엔진 구성

```
#include <stdio.h>

void main()
{
    FILE *fp;
    int readbytes;
    char buffer[128];

    fp = fopen("test.txt", "rb");
    if (fp != NULL)
    {
        do {
            readbytes = fread(buffer, 1, sizeof(buffer)-1, fp);
            buffer[readbytes] = '#0';
            printf("%s", buffer);
        } while(readbytes > 0);
        fclose(fp);
    }
}
```

파일 로더 구현부를 분리

- 파일로더의 구현부를 분리하고 구현한 클래스를 활용하여 알고리즘을 구현

```
#include <stdio.h>

class CFileIO
{
public :
    CFileIO() { m_fp = NULL; }
    ~CFileIO() { if (m_fp) fclose(m_fp); }

    bool Open(const char *fname)
    {
        m_fp = fopen(fname, "rb");
        return m_fp != NULL ? true : false;
    }
    int Read(void * buffer, int len)
    {
        return fread(buffer, 1, len, m_fp);
    }
private :
    FILE * m_fp;
};
```

리소스 추상화, 팩토리

- 파일 객체를 추상화 (인터페이스 중요!!)
구현부를 숨겨서 구현 가능, 쉽게 기능 확장 가능
- 사용하는 쪽에서 리소스를 얻기 위한 팩토리 추가

```
class CFileIO
{
public :
    virtual ~CFileIO() {}
    virtual int Read(void * buffer, int len) = 0;
    virtual int GetLength() { return 0; }
};
```

```
class CEngine
{
public :
    CFileIO * LoadFile(const char *fname, const char *type=0);
};
```

of Flame
Graphics Programming Study

Kasa

파일IO 엔진 사용

- 인터페이스가 정해지면 엔진을 사용하는 쪽의 변화는 거의 없다

```
void main()
{
    int readbytes;
    char buffer[128];
    CEngine engine;
    CFileIO * fp = engine.LoadFile("test.txt");
    if (fp != NULL)
    {
        do {
            readbytes = fp->Read(buffer, sizeof(buffer)-1);
            buffer[readbytes] = '#0';
            printf("%s", buffer);
        } while(readbytes > 0);
        delete fp;
    }
}
```

```

#include <stdio.h>

class CStandardFileIO : public CFileIO
{
public :
    CStandardFileIO() { m_fp = NULL; }
    ~CStandardFileIO() { if (m_fp) fclose(m_fp); }
    bool Open(const char *fname) {
        m_fp = fopen(fname, "rb");
        fseek(m_fp, 0, SEEK_END);
        m_len = ftell(m_fp);
        fseek(m_fp, 0, SEEK_SET);
        return m_fp != NULL ? true : false;
    }
    int Read(void * buffer, int len) {
        return fread(buffer, 1, len, m_fp);
    }
    int GetLength() { return m_len; }
private :
    FILE * m_fp;
    int m_len;
};

CFileIO * CEngine::LoadFile(const char *fname, const char *type)
{
    CStandardFileIO * f = new CStandardFileIO;
    if (f->Open(fname)== true)
        return f;
    delete f;
    return NULL;
}

```

- 구현부는 완전 분리된다

파일IO 엔진 기능 확장(1)

- 만약 메모리에 미리 읽어두도록 확장한다면

```
class CFileIOMemBuff : public CFileIO
{
public :
    CFileIOMemBuff(CFileIO * io) {
        m_len = io->GetLength();
        m_ptr = new char [m_len];
        m_offset = 0;
        io->Read(m_ptr, m_len);
    }
    ~CFileIOMemBuff() { delete m_ptr; }
    int Read(void * buffer, int len) {
        if (m_len - m_offset < len)
            return Read(buffer, m_len - m_offset);
        if (len > 0) {
            memcpy(buffer, &m_ptr[m_offset], len);
            m_offset += len;
        }
        return len;
    }
    int GetLength() { return m_len; }
private :
    char * m_ptr;
    int m_len, m_offset;
};
```

파일IO 엔진 기능 확장(2)

```
CFileIO * CEngine::LoadFile(const char *fname, const char *type)
{
    if (type != NULL && strstr(type, "membuf") != NULL)
    {
        CFileIO * f = LoadFile(fname);
        if (f != NULL)
        {
            CFileIO * fbuf = new CFileIOMemBuff(f);
            delete f;
            return fbuf;
        }
        return NULL;
    }

    CStandardFileIO * f = new CStandardFileIO;
    if (f->Open(fname)== true)
        return f;
    delete f;
    return NULL;
}
```

- 추가된 기능은 리소스를 생성할 때 type에 membuf 가 담긴 문자열만 보내면 된다

파일IO 엔진 기능 확장(3)

- Http를 이용해 파일을 읽어오는 기능을 확장한다면

```
class CFileIOMemPtr : public CFileIO
{
public :
    CFileIOMemPtr() { m_ptr = NULL; }
    ~CFileIOMemPtr() { delete m_ptr; }

    int Read(void * buffer, int len) {
        if (m_len - m_offset < len)
            return Read(buffer, m_len - m_offset);
        if (len > 0) {
            memcpy(buffer, &m_ptr[m_offset], len);
            m_offset += len;
        }
        return len;
    }
    int GetLength() { return m_len; }

protected :
    void SetPointer(char *ptr, int len) {
        m_ptr = ptr;
        m_len = len;
        m_offset = 0;
    }

private :
    char * m_ptr;
    int m_len, m_offset;
};
```

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa

```

#include <windows.h>
#include <wininet.h>

class CFileIOHTTP : public CFileIOMemPtr
{
public :
    bool Open(const char *url)
    {
        HINTERNET h, h2;
        char * ptr, * old;
        unsigned long tlen, readbytes, i = 0;

        h = InternetOpen("Microsoft Internet Explorer",
            INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, NULL);
        if (h)
        {
            h2 = InternetOpenUrl(h, url, NULL, 0,
                0, 0);
                //INTERNET_FLAG_DONT_CACHE|INTERNET_FLAG_NO_CACHE_WRITE, 0);

            if (h2)
            {
                ptr = new char [256];
                tlen = 256;
                do {
                    if (tlen == i) {
                        old = ptr;
                        ptr = new char [tlen + 256];
                        memcpy(ptr, old, tlen);
                        tlen += 256;
                        delete old;
                    }

                    InternetReadFile(h2, &ptr[i], tlen - i, &readbytes);
                    i += readbytes;
                } while(readbytes > 0);

                InternetCloseHandle(h2);
                SetPointer(ptr, i);
                return true;
            }

            InternetCloseHandle(h);
        }
        return false;
    }
};

```

```

CFileIO * CEngine::LoadFile(const char *fname, const char *type)
{
    if (type != NULL && strstr(type, "membuf") != NULL)
    {
        CFileIO * f = LoadFile(fname);
        if (f != NULL)
        {
            CFileIOMemBuff * fbuf = new CFileIOMemBuff;
            fbuf->Open(f);
            delete f;
            return fbuf;
        }
        return NULL;
    }

    if (!memcmp(fname, "http://", 7))
    {
        CFileIOHTTP * f = new CFileIOHTTP;
        if (f->Open(fname) == true)
            return f;
        delete f;
    }
    else
    {
        CStandardFileIO * f = new CStandardFileIO;
        if (f->Open(fname) == true)
            return f;
        delete f;
    }
    return NULL;
}

```

파일IO 엔진 정리

- 간단한 외부 인터페이스를 통해서 쉽게 사용 가능
- 외부에는 추상화 된 객체만 노출하여 구현부와 사용부를 완전 분리 (캡슐화)
- 추상화 특성을 이용하여 여러 가지 기능을 확장
- 팩토리를 통해서 쉽게 엔진의 추가된 기능을 사용

엔진 설계 포인트

- 탄탄하고 확장 가능한 내부 구조
- 외부 인터페이스 디자인

인터페이스에 따라 기능의 범위가 정해지고, 제약이 생기기도 하며, 확장을 가능하기도 하다

외부 인터페이스 디자인

- 1. 엔진의 기능을 외부에 노출하는 요소
- 2. 손쉽게 접근하는 디자인 요소

- 외부 인터페이스의 편의성, 단순성, 확장성, 융통성

사실상 기능을 외부에 노출시키는 것이 제일 중요한 요소라고 보면 필수 요소는 아니라고 할 수는 있음. (다만 게임 엔진은 어플리케이션 개발자를 타겟으로 함)

엔진 구성 요소

- 리소스, 디바이스
실제 게임에서 사용하는 객체
- 팩토리
리소스를 생성하고 관리하는 객체

리소스 객체 추상화(1)

- 추상화는 별도의 인터페이스 학습 없이 기능에 접근 가능하게 함
- udp, tcp, pipe 등의 제약 없이 쉽게 사용 가능한 네트워크 객체 ?
- d3d, opengl 을 지원하는 그래픽 디바이스?
- octree, bsp, quadtree, pvs를 지원하는 씬 매니징 객체 ?

리소스 객체 추상화(2)

- 네비게이션 메쉬, 웨이 포인트를 지원하는 길 찾기 객체 ?
- pvs, octree, view frustum culling, portal을 지원하는 culling 객체 ?
- 메모리db, oracle, mysql을 지원하는 db 객체 ?

리소스 객체 추상화(3)

- 인터페이스 변화 없이 추가적인 기능을 부여할 수 있음
- packet 압축을 하는 네트워크 엔진, packet을 저장해서 모니터링 정보를 생성하는 네트워크 객체 ?
- zip파일을 액세스 하는 파일 객체 ?

리소스 객체 설계 사례

- 리소스 인터페이스 구성 사례 (Cmodel)
- 객체 내부 인터페이스 구성 사례
- 디바이스 인터페이스 구성 팁

설계 사례 - 렌더링객체

- 화면상에 그려지는 렌더링 객체(Cmodel)의 추상 인터페이스 구성

```
class CModel {  
    virtual void LoadMesh(const char *fname);  
  
    virtual void SetTransform(const Matrix & mat);  
    virtual void Render();  
};
```

- 적절한 추상화는 일반 캐릭터, 정적 메시, Octree등으로 관리되는 대형 메시, 높이맵, 빌보드도 일괄적으로 처리 가능

Model 추상화

- 물리적인 천이나 수면도 Model 인터페이스로 구현할 수 있다면 어플리케이션 레벨에서는 똑같이 Model로 리스트로 관리해서 찍을 수도
- 만약 Model 에 GetBound 등의 메소드를 추가해서 Model을 장면 관리를 할 수 있게 한다면, 모든 렌더링 모델을 구현할 때는 컬링 걱정을 하지 않아도 될지도

Model 확장 - 애니메이션

- 애니메이션 기능을 추가한다면

```
class CModel {  
    virtual void LoadMesh(const char *fname);  
  
    virtual void LoadAnimation(const char *fname);  
    virtual void SetAnimation(const char *animname, float time);  
  
    virtual void SetTransform(const Matrix & mat);  
    virtual void Render();  
};
```

- 모션 블렌딩 기능까지 추가한다면

```
class CModel {  
    virtual void LoadMesh(const char *fname);  
  
    virtual void LoadAnimation(const char *fname);  
    virtual void SetAnimation(const char *animname, float time, float weight=1.0f);  
  
    virtual void SetTransform(const Matrix & mat);  
    virtual void Render();  
};
```

Model 복잡도 증가요소

- 만약 애니메이션을 공유한다면
- IK를 적용한다고 하면
- *나쁘다고 볼 수는 없지만 기능 추가에 따라 추상화 객체 CModel의 Method가 증가하며 복잡도가 늘어난다*

외부 제어 객체(1)

- 외부에서 Model의 본을 제어할 수 있는 MotionController 객체 추가하면??

```
class CMotionController
{
    virtual void EnumBone();
    virtual Matrix & GetBasicMatrix(const char * name);
    virtual void SetMatrix(const char * name, const Matrix &mat);
    ....
};

class CModel {
    virtual void LoadMesh(const char *fname);

    virtual CMotionController * GetModelController();

    virtual void SetTransform(const Matrix & mat);
    virtual void Render();
};
```

외부 제어 객체(2)

- 이 방법으로도 애니메이션과 IK를 구현 가능

```
class CAnimationController
{
    virtual void Load(const char *fname);
    virtual void SetAnimation(CModelController * ctl, const char *animname, float t);
};

class CIKController
{
    virtual void SetModelController(CModelController * ctl);
    virtual void SetConstrain(...);
    virtual void Solve(const char *root, const char *endeff, Vector3 & vec);
};
```

- 별도의 장치 없이 IK와 애니메이션을 혼합 가능
- 게임 어플리케이션에서 외부 제어 객체에 접근 안 한다고 가정하면 객체 정의를 숨가는 것도 가능

외부 제어 객체(3)

- *검기를 구현한다면 ??*
- *특정 Model의 본에 다른 Model을 attach한다면??*
- *추가적인 외부 제어 객체를 통해서 다른 기능을 추가 애니메이션 외에도 텍스처 애니메이션이 필요하다면??*

쉬어가기 (Tip)

인덱스 활용

```
class LMotionController
{
    virtual void EnumBone();
    virtual Matrix & GetBasicMatrix(const char * name);
    virtual void SetMatrix(const char * name, const Matrix &mat);
    ....
};
```

- 문자열을 이용하면 개발 효율이 좋지만 검색 속도상 불이익이 있을 수 있다
- 테이블 등을 활용한 인덱스를 활용하면 속도상의 불이익을 크게 줄일 수 있다.

```
class CMotionController
{
    virtual void EnumBone(...);
    virtual int FindBone(const char *name);
    virtual Matrix & GetBasicMatrix(int index);
    virtual void SetMatrix(int index, const Matrix &mat);
    ....
};
```

인스턴스 객체

- ***Proxy 패턴을 적극 활용하면 인스턴스 구현상 고민이 줄어든다***

```
class CD3DModel : public CModel {
    ...
    CMotionController * GetMotionController() { return &m_motionctl; }

    void SetTransform(const Matrix & mat);
    void Render();
private :
    Matrix m_matrix;
    CD3DMotionController m_motionctl;
};

class CD3DModelProxy : public CModel {
    CD3DModelProxy(CD3DModel * body) { m_body = body; m_body->AddRef(); }
    ~CD3DModelProxy() { m_body->SubRef(); }

    CMotionController * GetMotionController() { return &m_motionctl; }

    void SetTransform(const Matrix & mat) { m_matrix = mat; }
    void Render() {
        CopyController(body->GetMotionController(), &m_motionctl);
        body->SetTransform(m_matrix);
        body->Render();
    }
private :
    Matrix m_matrix;
    CD3DMotionController m_motionctl;
};
```

디바이스

- 싱글톤으로 구성 가능
- 어플리케이션 레벨에 노출되는 외부 인터페이스는 적음
- 실제 구현 객체와 연동되어 사용됨
- *역시 단순화는 구현에 도움이 됨*
- *디바이스 관련 메소드가 리소스에 위치하는 것보다는 디바이스 객체에 있는 것이 유리*

리소스 팩토리

- **Abstract Factory**
- 어플리케이션 레벨에선 추상화 된 객체만을 받는다
- 팩토리의 인터페이스에 따라 쉽게 추가된 기능 사용가능
- 독립되어 내부 구현의 자유도는 높다

인스턴스 매니징

- 하나의 자원을 여러 개의 리소스 객체에서 공유해서 사용할 수 있다.
- 만약 자원 공유가 되지 않으면 중복되는 자원 낭비가 심해진다

리소스 매니징

- 게임에서 사용할 수 있는 시스템의 자원은 제한됨
- 쾌적한 게임을 위해서 사용하지 않는 자원을 관리하는 정책이 필요

- 불필요한 자원을 검출하는 알고리즘이 필요
사용중인 인스턴스의 수를 검사
마지막 사용한 시간으로 검사

- *신경 쓰지 않으면 관리가 힘든 요소가 많음*

기타

- DLL 사용하는 경우 자원 관리
자원 해제문제

질문

Spirit of Flame
3D RealTime Graphics Programming Study

Kasa