Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #10

Resource and object sharing

Lecture #10

Part I: Resource

Introduction

- When you distribute your game, you do not want the user to have access to the data
  - audio, video, textures, 3D models, fonts, *etc.*
  - for copyright or distribution policy
  - to avoid huge folders
- You want to hide and combine them into one (or few) file(s): the resource file(s)
  - conventionally named resources.dat
  - often one file per type (and/or per game level)
- A resource file is a binary file containing data that you can distribute along with the executable

Custom resource format

- The resource manager is in charge of
  - creating the files during development
  - loading the files at execution time
- You can create your own resource manager, the format used for the resource files is then up to you
  - *e.g.* byte-by-byte format to import and export textures, meshes and sound files
  - encryption and compression algorithms can also be incorporated at the same time

File format

- The resource file is composed of
  - A header (resource file structure)
  - A body (data information and content)

File format

- The header
  - contains information describing the content of the resource, for example:
  - First 4 bytes
    - an integer value n indicating how many data are stored in the resource file
  - Next 4 x n bytes
    - an integer value pointing to the storage location of the data within the resource
    - *e.g.* value 2341 indicates that data starts at the resource's 2341 byte

File format

- The body
  - contains the name of each resource stored and the actual data
  - for each data
    - First 4 bytes
      - an integer value d indicating how many bytes of data
    - Next 4 bytes
      - an integer value c indicating how many characters are in the resource name
    - Next c bytes
      - each byte contains a name character
    - Next d bytes
      - the stored data

File format

- Example

BYTELOC	DATA	EXPLANATION
0-3	3	(Integer indicating that 3 data are stored in this resource file)
4-7	16	(Integer indicating that the first data is stored from the 16th byte)
8-11	41	(Integer indicating that the second data is stored from the 41st byte)
12-15	10058	(Integer indicating that the third data is stored from the 10058th byte)
16-19	9	(Integer indicating that the first stored data contains 9 bytes)
20-23	8	(Integer indicating that the first stored data's name is 8 characters in length)
24-31	TEST.TXT	(8 bytes, each encoding one character of the first stored data's name)
32-40	Testing12	(9 bytes, containing the first stored data, which happens to be some text)
41-44	10000	(Integer indicating that the second stored data contains 10000 bytes)
45-48	9	(Integer indicating that the second stored data's name is 9 characters in length)
49-57	TEST2.BMP	(9 bytes, each encoding one character of the second stored data's name)
58-10057	...	(10000 bytes, representing the data stored within TEST2.BMP. Data not shown here.)
10058-10061	20000	(Integer indicating that the third stored data contains 20000 bytes)
10062-10065	9	(Integer indicating that the third stored data's name is 9 characters in length)
10066-10074	TEST3.WAV	(9 bytes, each encoding one character of the third stored data's name)
10075-30074	...	(20000 bytes, representing the data stored within TEST3.BMP. Data not shown here.)

  - the resource file is 30074 bytes in size (approx. 29.4 KB) and contains the data represented by TEST.TXT, TEST2.BMP and TEST3.WAV

Resource manager

- We need a component that can store and read files in this format: the resource manager

```
class ResourceManager {
 public:
   struct ResourceInfo {
         int size;
         int namesize;
         string name;
   }; // structure to store file information within the resource

   void createResourceFile(string resFolder, string resFile);
   // create the resource file resFile from all files in resFolder

   vector<ResourceInfo> infoFiles(string resFolder);
   // get the file information from all files in resFolder

   char * getResourceByName(string resFile, string resName, int &sizeData);
   // get the data from a resource file (update sizeData)
};
```

Resource manager

- To get the information structures (Windows)

```
#include <windows.h>
#include <sys/stat.h>

vector<ResourceInfo> infoFiles (string resFolder) {

    vector<ResourceInfo> res;

    bool success = SetCurrentDirectory(resFolder.c_str()); // change directory
    if (!success) {
        cout << "Error directory not found:" << resFolder;
        return res;
    }

    WIN32_FIND_DATA FindFileData;
    HANDLE hFind = FindFirstFile("*",&FindFileData); // find 1st element in folder
    if (hFind == INVALID_HANDLE_VALUE) return res;

    // ...
```

Resource manager

- To get the information structures (Windows)

```
    // ...

    do {
         string filename (FindFileData.cFileName);
         if (filename.compare(".") && filename.compare("..")) {
          // not self or parent, could also check for sub-directory
             ResourceInfo resinfo;
             resinfo.size = (FindFileData.nFileSizeHigh * (MAXDWORD+1))  +
                             FindFileData.nFileSizeLow; // set file size
             resinfo.namesize = filename.length(); // set file name size
             resinfo.name = filename; // set file name
             res.push_back(resinfo); // store file info
         }
    }
    while (FindNextFile(hFind,&FindFileData) != 0);

    FindClose(hFind);

    return res;
}
```

Resource manager

- Creation of the resource file

```
void createResourceFile(string resFolder, string resFile) {

    ofstream outfile; // output resource file
    outfile.open(resFile,ios::binary);
    if (!outfile.is_open()) {
         cout << "Unable to create file: " << resFile;
         system("pause");
         return;
    }

    char * buffer; // buffer to save

    // get all files information
    vector<ResourceInfo> fileinfo = infoFiles(resFolder);
    int numberOfFiles = (int) fileinfo.size();
    buffer = (char *) &numberOfFiles;
    outfile.write(buffer, sizeof(int)); // write the number of files

    // ...
```

Resource manager

- Creation of the resource file

```
// ...

// resource header:
int offset = (numberOfFiles+1) * sizeof(int);
// header offset is +1 because of the first "number of files" integer

for (int f = 0; f < numberOfFiles; f++) {

     // location of the data file within resource
     buffer = (char *) &offset;
     outfile.write(buffer, sizeof(int));

     // update offset: file size + name size + name + data
     offset += sizeof(int) + sizeof(int) + fileinfo[f].namesize +
                fileinfo[f].size;
}

// ...
```

Resource manager

- Creation of the resource file

```
    // ..

    // resource body:
    for (int f = 0; f < numberOfFiles; f++) {
          int datasize = fileinfo[f].size;
          buffer = (char *) &datasize;
          outfile.write(buffer, sizeof(int)); // size of the file
          int namesize = fileinfo[f].namesize;
          buffer = (char *) &namesize;
          outfile.write(buffer, sizeof(int)); // size of the file name
          const char * filename = fileinfo[f].name.c_str();
          outfile.write(filename, namesize);  // name of the file
          ifstream datafile;
          datafile.open(filename,ios::binary);
          char * readData = new char [datasize];
          datafile.read(readData,datasize);
          outfile.write(readData,datasize); // copy all data at once
          datafile.close();
          delete [] readData;
    }

    // close resource file
    outfile.close();
}
```

Resource manager

- Reading of the resource file

```
char * getResourceByName (string resFile, string resName, int &sizeData) {

   ifstream infile; // input resource file
   infile.open(resFile,ios::binary);
   if (!infile.is_open()) {
        cout << "Unable to open file: " << resFile;
        system("pause");
        return NULL;
   }

   // buffer to load data
   char * buffer = new char [sizeof(int)];

   // number of files
   infile.read(buffer, sizeof(int));
   int numberOfFiles = *((int *) buffer);

   // ...
```

Resource manager

- Reading of the resource file

```
// ...

// vector of offset in header
vector<int> resourceAddress;
for (int f = 0; f < numberOfFiles; f++) {
     // read each file location
     infile.read(buffer, sizeof(int));
     int address = *((int *) buffer);
     // store them in vector
     resourceAddress.push_back(address);
}

// ...
```

Resource manager

- Reading of the resource file

```
    // ...
    // resource body
    for (int f = 0; f < numberOfFiles; f++) {
          int location = resourceAddress[f];
          infile.seekg(location);
          infile.read(buffer, sizeof(int)); // read file data size
          int size = *((int *) buffer);
          infile.read(buffer, sizeof(int)); // read file name size
          int namesize = *((int *) buffer);
          char * name = new char [namesize+1];
          infile.read(name, namesize); // read file name
          name[namesize] = '\0';
          string sname (name);
          if (!resName.compare(sname)) { // resource found!
                    char * data = new char [size];
                    infile.read(data, size); // read the data
                    delete name;
                    delete [] buffer;
                    infile.close();
                    sizeData = size; // update sizeData
                    return data;
          }
          delete name;
    }
    // ...
```

Resource manager

- Reading of the resource file

```
    // ...

    // exit properly
    delete [] buffer;
    infile.close();
    return NULL;
}
```

Resource manager

- Usage

```
// Manager creation
ResourceManager mgr;

// Create the resource file
mgr.createResourceFile(“GameResourceFolder”,“myResourceFile.dat”);

// Read from the resource file
int sizeData;
char * data = mgr.getResourceByName(“../myResourceFile.dat” ,
   “myFile.ext” , sizeData);

// Use of the data (example)
for (int d=0; d < sizeData; d++) {
   // ... code using data[d] ...
}
```

Resource manager

- Then, some tools (libraries) with help you to convert the char * data to a usable image, sound, text *etc.* in your game
  - Such as the Simple DirectMedia Layer (SDL) library
- You can create your own conversion routines that depends on the graphics engine, audio manager *etc.*
- You can also physically re-create a temporary file to load in your game and delete it when done (much slower)

Resource manager

- The manager should not load twice the same resource
  - waste of time and memory
- The manager keeps track of the loaded resources
  - usually one map per type of resource

```
map<string, Texture2D *> _sprites;
map<string, SoundEffect *> _sounds;
map<string, 3DMesh *> _meshes;
// ...
```

Resource manager

- The manager checks the loaded assets before reading the resource file again
- Or every asset is loaded at start-up to avoid lag at run-time (but potential useless memory allocation)

```
Texture2D * getSprite(string assetName) {
    Texture2D * theTexture = NULL;
    map<string, Texture2D *>::iterator it = _sprites.find(assetName);
    if (it == _sprites.end()) { // asset not found
        int sizeData;
        char * data = mgr.getResourceByName(“resources.dat”,assetName,sizeData);
        theTexture = new Texture2D(data,sizedata);
        _sprites[assetName] = theTexture; // add resource to map
    }
    else theTexture = it->second; // asset already loaded
    return theTexture;
}
```

Visual Studio resources

- Visual Studio has its own resource manager
- You can directly import some file formats
  - Bitmap (bmp, dib, gif, jpg, jpe, jpeg, png)
  - Icon (ico)
  - Cursor (cur)
  - Audio (wav)
  - Web page (html, htm)
- You can create custom import procedures for other formats

Lecture #10

Part II: Object sharing

Object sharing

- Consider the following code

```
// Create a new enemy and point the player to it
Enemy* enemy = new Enemy();
player.setTarget(enemy);
// ...
// Some time later, the enemy dies
delete enemy;
```

- What may happen here?
  - the player object does not know that the enemy object is deleted, creating a dangling pointer
- There are several solutions for solving this object sharing problem

Object sharing

- Solution 1: Do not allow object sharing in your game
  - Unfortunately not always possible or desirable
  - It also means you have sometimes to keep duplicate copies (textures, sounds, …)
  - In the case of the enemy, we do not want to pass a copy to it instead of the original, as its state will change

Object sharing

- Solution 2: Ignore the problem
  - Potentially this would lead to problems
    - if the player want to access the enemy state
  - For small project, it might work
  - Probably only acceptable for a prototype or a tech demo
  - If everything is statically allocated, then you could also get away with it
  - However, no easy fixes when a bug does occur

Object sharing

- Solution 3: Leave it up to the owner
  - Every shared object is assigned an owner
  - The owner is the only responsible for creating, managing and deleting the object
  - Not possible to enforce on users, but if dealt with carefully it could work
  - What happens when an object changes owner?
  - What to do with pointers from non-owner objects?
    - Add notifying behavior (Listener DP)
    - And extra performance cost

Object sharing

- Solution 4: Reference counting
  - No need for an owner
  - Object is kept around as long as it is needed
  - As soon as the last reference goes away, we delete the object

Object sharing

- Solution 4: Reference counting

```
class RefCounted {
   public:
         virtual ~RefCounted() {};

         int addRef() { return ++refCount_; }

         int release(){
                 --refCount_;
                 int tmpRefCount = refCount_;
                 if (refCount_ <= 0) delete this;
                 return tmpRefCount; // copy of deleted refCount_
                 // ok as function call on stack (return value and @)
         }

         int getRefCount() const { return refCount_; }

   protected:
         int refCount_;
};
```

Object sharing

- To use the reference counting functionality, a class just inherits from *RefCounted*
- For additional security, we might declare the destructor of *RefCounted* as protected
  - – called only from release function

```
class Enemy : public RefCounted {
   public:
         // ...
   protected:
         // ...
         virtual ~Enemy ();
};
```

Object sharing

- Drawbacks of reference counting
  - – You have to remember to call addRef() and release() ‘everywhere’ (and correctly)
    - if not, either object never deleted (memory leak) or deleted too early (run-time crash during further access)
    - quite difficult to maintain, and easily unstable
  - – Objects could get destroyed a bit too easily
    - example: re-use of the same object few lines later
    - we could add an object manager (mostly for resources) that keeps always 1 reference to them
  - – More (awkward) code

Object sharing

- Solution 5: Handles
  – Shared object problems are due to the existence of multiple pointers to the same object
  – Handles prevent that situation from happening
  – Instead of using pointers to a shared object, we are using an identifier (the handle)
  – When we want to use the object, we ask the owner for a pointer that corresponds to the handle
  – After usage, we throw the pointer away

Object sharing

- One pointer per shared object exists: the one from the owner
- Users of the object pass by the handle first
- If the object does not exist anymore, a NULL pointer is returned
- Handles can be a plain integer
- In order to ensure a unique identifier for each entity, 32-bit number should be enough

Object sharing

- Examples
  - Enemy object

```
typedef unsigned int Handle;
Handle hEnemy = CreateEnemy();
// ...
Enemy * pEnemy = GetEnemy(hEnemy);
```

  - Handles for textures

```
typedef unsigned int Handle;
Handle hTexture = CreateTexture(“texture.tif”);
// ...
Texture* pTexture = GetTexture(hTexture);
if (pTexture != NULL) {
   // texture still exists, we can do something with it
}
```

Object sharing

- Handles can be cumbersome because we need the translation step to get the pointer
  - Generally implemented using a map, or a hash table
  - Main performance hit is caused by the indirection level
- Again, think of at which level handles are useful
  - Do not try a “one-handle-per-polygon” approach

Object sharing

- Solution 6: Smart pointers
  - Smart pointers know what is happening to the objects that they refer to
  - C++ flexibility allows us to create objects that look and feel like pointers + do some extra work for us such as:
    - Check that memory is valid
    - Keep reference counts and statistics
    - Apply different pointer copying policies
    - Delete object they are pointing to if the pointer itself is destroyed

Object sharing

- Smart pointers generally behave like real pointers
  - -> and * operators implemented, fast copy, type-safe, memory efficient
- Under the hood, smart pointers
  - implement handles
  - or do reference counting

Object sharing

- A handle-based smart pointer is simply a wrapper around a handle

```
class EnemyPtr {
 public:
  EnemyPtr(Handle h) : hEnemy_(h) {}

  bool operator == (int n) { return n == (int)getEnemy(hEnemy_); }

  bool operator != (int n) { return !operator==(n); }

  Enemy * operator -> () { return getEnemy(hEnemy_); }

  Enemy & operator * () { return *getEnemy(hEnemy_); }

 private:
  Handle hEnemy_;
};
```

Object sharing

- We can treat it as a real pointer

```
EnemyPtr ptr(enemyHandle);
if (ptr != NULL) {
   cout << ptr->getName();
   const Point3D& pos = ptr->getPosition();
}
```

- This EnemyPtr class works only for pointers to Enemy objects
- We can use template to create smart pointers of any type

Object sharing

- Template smart pointers

```
template <class T>
class HandlePtr {
 public:
  HandlePtr(Handle h) : hObject_(h) {}

  bool operator == (int n) { return n == (int)getPtr(hObject_); }

  bool operator != (int n) { return !operator==(n); }

  T * operator -> () { return getPtr(hObject_); }

  T & operator * () { return *getPtr(hObject_); }

 private:
  Handle hObject_;
};
```

Object sharing

- Template smart pointers

```
typedef HandlePtr<Texture> TexturePtr;
typedef HandlePtr<Enemy> EnemyPtr;
// ...

EnemyPtr pEnemy = CreateEnemy();
// ...
if (pEnemy != NULL) Game::Instance()->addEnemy(*pEnemy);

TexturePtr pTexture = CreateTexture(“wall.png”);
// ...
if (pTexture != NULL) pTexture->draw();
```

Object sharing

- A reference-counting based smart pointer is simply a wrapper around a reference counting
  - Every time a smart pointer is created, the reference count is incremented
  - Wherever the object is deleted, the reference count is decremented
- Reference counting (addRef, release) can be moved from the shared object class to the smart pointer class
- Template approach can also be implemented for type-safe use with any pointer type

Resource maintenance

- Resources are usually shared objects
- RAII: resource acquisition is initialization
- Example
  - Reading a file

```
void World::LoadMap (const string& fileName) {
   FILE * file = fopen(fileName.c_str(),”r”);
   // read the file and do something with it
   // that might goes wrong
   fclose(file);
}
```

  - In case of critical error (*e.g.* exception thrown) the file would not be closed

Resource maintenance

- Adding try/catch for exception safety

```
void World::LoadMap (const string& fileName) {
    FILE * file = NULL;
    try {
        file = fopen(fileName.c_str(),”r”);
        // read the file and do something with it
        // that might generate an exception
    }
    catch (...) {
        fclose(file);
        throw;
    }
    // ...
    fclose(file);
}
```

Resource maintenance

- All exceptions are caught and the file is closed, *i.e.* the resource is released in the catch block
  - Error-prone, because it can get rather complicated if numerous resources are acquired and released
  - C++ does not have a finally keyword
  - Code duplication for delete/cleanup operations
- A more elegant solution
  - Wrap resources into classes, and use constructors for acquisition and destructors for release
    - Destructors are called even when exceptions appear and this way release is guaranteed

Resource maintenance

- A handler based file pointer class

```
class FilePtr {
   public:
        FilePtr(const std::string& fileName);
        ~FilePtr();
        FILE * getFileHandler();
   private:
        FILE * handler_;
}
```

```
FilePtr::FilePtr(const std::string& fileName) {
   handler_ = fopen(fileName.c_str(),”r”);
}
FilePtr::~FilePtr(){ fclose(handler_); }

FILE * FilePtr::getFileHandler(){ return handler_; }
```

Resource maintenance

- Using the file pointer class

```
void World::LoadMap (const string& fileName) {
   FilePtr file (fileName);
   // read the file and do something with it
   // that might generate an exception
}
```

  – FilePtr object is automatically destroyed by the destructor and the resource is released (either by exception throwing or function termination)

Using auto_ptr

```
class Player {
   /* ... */
};

void Run () {
   Player * p = new Player();
   // <- throws exception
   delete p;
}
```

- In case of an exception, the object p is not deleted

Using auto_ptr

- Use auto_ptr for dynamically allocate local objects (on the heap)
  - to store a pointer to an object obtained via new
  - to delete that object when it itself is destroyed (such as when leaving block scope)

```
void Run () {
   auto_ptr<Player> p (new Player());
   // ...
}
```

- auto_ptr takes care of deleting p when leaving the scope
  - either on normal return or when an exception appears

Using auto_ptr

- An auto_ptr owns the object it holds a pointer to
- Copying an auto_ptr copies the pointer and transfers ownership to the destination
- If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined.

Using auto_ptr

- You can do

```
auto_ptr<Player> p1 (new Player());
auto_ptr<Player> p2 = p1;
```

  - p2 will own the object, p1 is set to NULL
  - deleting p1 does not delete Player object

- You cannot do (should not do)

```
Player* player = new Player();
auto_ptr<Player> p1 (player);
auto_ptr<Player> p2 (player);
```

  - more than one auto_ptr owns the Player object

Using auto_ptr

- Conventional pointer vs. auto_ptr

```
class Player {
public:
   Player();
   ~Player();
private:
   State * pS_;
};
```

```
class Player {
public:
   Player();
   ~Player();
private:
   auto_ptr<State> apS_;
};
```

```
Player::Player() : ps_(new State())
{ }

Player::~Player() {
   delete ps_;
}
```

```
Player::Player() : aps_(new State())
{ }

Player::~Player() { }
```

Using auto_ptr

- The auto_ptr public members

```
(constructor)     // Construct auto_ptr object
(destructor)      // Destruct auto_ptr

get               // Get the pointer

operator*         // Dereference object
operator->        // Dereference object member
operator==        // Release and copy auto_ptr

release           // Release pointer (set to NULL)
reset             // Deallocate object pointed and set new value
```

End of lecture #10

Next lecture

Optimization and Advanced STL