Tutorial – Resource Management

In this tutorial we will be creating a simple resource manager.

There are many different ways we could create a resource manager, so as you complete this tutorial remember that this is only a guide. If you feel a different design would work better for your game, feel free to design a resource manager that better matches your needs.

As you learn more data structures and game design patterns, you may want to revisit this resource manager, integrating your newfound knowledge and optimizing the way this resource manager works.

Design Goals:

There are a few things we need our resource manager to do. It should:

* Allow us to specify the file path of the resource we want to load
* Load the resource if it has not already been loaded
* Return a reference to a loaded resource if the resource has already been loaded
* Automatically track how many references to the resource exist
* Automatically track when a resource is not being used
* Allow us to unload all unused resources

Those are some lofty goals.

Tracking whether or not a resource has been loaded is relatively simple. All we need to do is keep a big list (in our case, a dynamic array) of resources that have been loaded. When we want to load a new resource, we simply check the list to see if it already exists. If it does, then we return an instance to that object.

Automatically tracking when an object is no longer being used is a little tricker, although well worth the effort. Implementing this means we will no longer need to worry (so much) about memory management and unloading resources once we’ve finished with them.

To get this working, we’re going to use shared pointers.

If you’re not familiar with the *std::shared\_ptr* class, you should take a moment now to review the lecture on Shared Pointers, or review the relevant documentation (the Microsoft Developer Network has some very good pages on this).

Here is the class diagram for our design:



The Resource Class:

There are possible implementations we could write that would negate the need for a *Resource* class, although these use the associative container classes that we won’t learn about until much later.

Our Resource Manger needs to store a list of resources (be they textures, sounds or fonts) and be able to match them with their associated file paths.

To accomplish this using what we know so far, we’re going to encapsulate the resource pointer and filename in the *Resource* class. Our *ResourceManager* will then use the *std::vector* class to store our Resource class instances (well, technically its storing shared\_ptr instances to Resource objects).

Take a closer look at the *Resource* class.

You’ll notice that the *m\_pData* variable is a *std::unique\_ptr* instance. This allows us to loads the resource using the *new* keyword (for example *new Texture(…)*), but we don’t need to worry about cleaning up. The smart pointer will automatically destroy the texture when the *Resource* object goes out of scope.

The definition for the *Resource* class is as follows:

template<class T>

class Resource

{

public:

Resource(const std::string& filename) : m\_filename(filename) {

m\_data = std::unique\_ptr<T>(new T(filename.c\_str()));

};

~Resource() {};

std::string getFilename() { return m\_filename; }

private:

std::unique\_ptr<T> m\_data;

std::string m\_filename;

};

Notice that this is a template class.

Using templates this way makes it easy for us to create the kind of resource we want. If we pass in the *aie::Texture* to the template, then the constructor creates a new Texture.

What this means in practice is that we’ll have a resource manager for each *type* of resource. One for fonts, one for textures, one for sounds, etc.

This may or may not be a bad thing, depending on the needs of your game.

An alternative to using templates may be to create a *ResourceType enum*, and pass this as an argument when calling the *Resource* class constructor. This way you could create the type of resource (texture, font, etc.) specified by the argument.

Also note that with template classes, the complete class definition must reside in the header file. (You can go ahead and delete the .cpp file if you already created it). Do you know why this is the case? Look it up now if you’ve forgotten.

The ResourceManager Class:

This class is not difficult to write, but the templates and smart pointers can trip us up if we’re not clear on the functionality we’re writing.

Let’s start with the basic class declaration (we’ll fill in each function in turn):

template< class T >

class ResourceManager

{

std::vector< std::shared\_ptr<Resource<T>> > m\_resources;

// keep the copy constructor and assignment allocator private

ResourceManager(const ResourceManager&) {};

ResourceManager& operator=(const ResourceManager&) {};

public:

ResourceManager() {};

~ResourceManager() {};

std::shared\_ptr<Resource<T>> get(const std::string filename) { };

int getCount();

void collectGarbage() {};

};

The first thing you’ll probably notice is all of those templated classes we’re using to declare the *m\_resources* variable.

Simply put, the *m\_resources* variable is a dynamic array of shared pointers that refer to instance of the *Resource* class. (Perhaps that wasn’t so simply put)

When we create an instance of the Resource class, we put it on the heap. The shared pointer takes ownership of this instance and wraps it in an object that is stored on the stack (in our case, in the *vector*).

You may be thinking, why don’t we just store *Resource* objects in the vector and make the *m\_pData* variable of the *Resource* class the shared pointer?

That’s a valid point, but it means we’d be calling the copy constructor or assignment operator on the *Resource* class a lot. Did you notice that big messy string hiding in the *Resource* class?

This design means we can pass around shared pointers to *Resources* (instead of copying *Resources* containing shared pointers). Conceptually it’s a little easier to understand, and it’ll be a little faster as well!

We’ve made the copy constructor and the assignment operator of the *ResourceManager* class private, which means no copying a *ResourceManager* from one instance to another. This ensures we don’t end up with two resource managers managing the same resource.

So that means the first function we need to implement is the *.get()* function. This function accepts the filename to load, creates a *Resource* instance, wraps it in a shared pointer, inserts it in our vector, and finally returns the shared pointer. If the resource is already loaded, we just return the shared pointer to that instance.

std::shared\_ptr<Resource<T>> get(const std::string filename)

{

std::vector< std::shared\_ptr<Resource<T>> >::iterator it;

for (it = m\_resources.begin(); it != m\_resources.end(); it++)

{

if (filename.compare((\*it)->getFilename()) == 0)

{

return (\*it);

}

}

std::shared\_ptr<Resource<T>> resource( new Resource<T>(filename) );

m\_resources.push\_back(resource);

return resource;

}

If we wanted to know how many resources the manager currently had loaded (say, for debugging purposes), all we need to do is return the size of the *m\_resources* vector.

int getCount()

{

return m\_resources.size();

}

That leaves the *collectGarbage()* function.

This function will destroy all resources where the reference count stored in the shared pointer equals 1.

The reference count will never be 0, because the reference to the instance in the *ResourceManager*’s vector counts as 1.

This means that if a resource ever has a reference count of exactly 1, then only the resource manager holds a reference to it and it can safely be deleted.

void collectGarbage()

{

for (std::vector< std::shared\_ptr<Resource<T>> >::iterator it = m\_resources.begin(); it != m\_resources.end(); )

{

if ((\*it).use\_count() == 1)

it = m\_resources.erase(it);

else

++it;

}

}

Notice the lack of the increment in the post condition of the for loop. This is because when we erase, the iterator becomes invalidated and we obtain a new iterator from the *erase()* function that points to the next element in the list.

If we iterated in the for loop, we’d skip an element in the vector. So instead, we only want to increment the iterator if we haven’t erased an element.

That’s completes all we need for our simple resource manager.

To use the resource manager, create a simple Player class as follows:

typedef std::shared\_ptr<Resource<aie::Texture>> TexturePtr;

class Player

{

public:

Player(TexturePtr texture) : m\_ship(texture) {};

~Player() {};

private:

TexturePtr m\_ship;

};

(Notice how I’ve added a typedef to make using the resources a little easier. If you do this, don’t forget you’ll need a typedef for each type of ResourceManager you have)

Now, in your application class create an instance of the *ResourceManager*, and a Player:

class MyApp : public aie::Application {

public:

...

protected:

...

ResourceManager<aie::Texture> m\_images;

Player\* m\_player;

};

Although, now that we know the benefits of smart pointers we’d ideally want to wrap that Player pointer in a unique\_ptr instance.

Now let’s put our resource manager to use and test it with a bit of debugging code:

bool MyApp::startup() {

...

TexturePtr pShip = m\_images.get("./textures/ship.png");

m\_player = new Player(pShip);

std::cout << "images loaded: " << m\_images.getCount() << std::endl;

return true;

}

void MyApp::shutdown() {

std::cout << "images loaded: " << m\_images.getCount() << std::endl;

delete m\_player;

m\_images.collectGarbage();

std::cout << "images loaded: " << m\_images.getCount() << std::endl;

...

delete m\_2dRenderer;

}

In the *startup* function we use the *.get()* function of our resource manager to get a new shared pointer *Resource*, which we pass to the constructor of our Player class. (Did you notice how we could have done this in one line of code?)

In the *shutdown* function we delete the player instance and call *collectGarbage()* to clean up any unused resources.

I’ve added some debugging code that will output the number of textures currently loaded by the resource manager at various times in the game’s life.

Challenge:

1. As we mentioned before, we could remove the *Resource* class altogether and use an associative container instead of a *vector*.

Replace the *std::vector* in your *ResourceManager* with the *std::map*.

(You may want to leave this until you cover the associative containers session during class)

1. How do you feel about using templates for the resource manager? Would you rather have one manager class handle all the resources? Create a *ResoruceType* enumeration and use this when loading a resource to specify how the resource should be loaded.