Tutorial – Design Patterns

In this tutorial we are going to implement the singleton, prototype and factory patterns. While the sample application we will build is reasonably trivial, it is hoped that the simplicity of the implementation will highlight the implementation details of these patterns and inspire you to use them in your games.

A Texture Manager Using the Singleton Pattern:

We’ll be creating a lot of sprites in this demo. What we want to avoid is reloading the texture for each new sprite. Instead, we want to check if a texture has already been loaded – if it hasn’t then load it, otherwise reuse the previously loaded texture.

To achieve this we’ll need a specialized class responsible for checking if the texture has already been loaded – the Texture Manager.

While we could create the class object in our Application class and pass a pointer around to any class that needs to load a texture, a simpler implementation would be to allow us to access the Texture Manager class object from anywhere in our codebase. To do this we’ll make the Texture Manager class a Singleton.

Using the AIEProject template, make a new application and add the following class:

#include <map>

#include <string>

class Texture;

class TextureManager

{

protected:

static TextureManager\* s\_pInstance;

TextureManager();

public:

~TextureManager();

static TextureManager\* GetInstance();

static void DestroyInstance();

Texture\* LoadTexture(const char\* filename);

protected:

std::map<std::string, Texture\*> m\_textureList;

};

Notice that we’ve made the constructor protected. What this means in practices is that only the TextureManager class can create TextureManager objects.

This may sound like a catch-22 (how do we make TextureManager objects when we need a TextureManager object to create new TextureManager objects?), but it’s not. We’ll also define the static function GetInstance() to create and return the singleton instance.

The definition of the Texture Manager class is as follows:

#include "TextureManager.h"

#include "Texture.h"

TextureManager\* TextureManager::s\_pInstance = nullptr;

TextureManager\* TextureManager::GetInstance(){

if(s\_pInstance == nullptr) {

s\_pInstance = new TextureManager();

}

return s\_pInstance;

}

void TextureManager::DestroyInstance(){

if(s\_pInstance != nullptr) {

delete s\_pInstance;

s\_pInstance = nullptr;

}

}

TextureManager::TextureManager(){}

TextureManager::~TextureManager(){

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

{

delete it->second;

}

}

Texture\* TextureManager::LoadTexture(const char\* filename){

auto it = m\_textureList.find(filename);

if(it != m\_textureList.end())

return it->second;

Texture\* pT = new Texture(filename);

m\_textureList[filename] = pT;

return pT;

}

So GetInstance() first checks to see if the static s\_pInstance variable is null. If it is then a new TextureManager is created and returned, otherwise the previously created object is returned. This function ensures that only 1 instance of the class exists within our program at any point in time.

We’ve also add a static function to destroy the Texture Manager instance. You should call this function from your applications destructor. It will ensure all the textures are unloaded and all memory is deallocated.

To load a texture, we would first get the instance via a call to GetInstance(), and then call the LoadTexture() function.

For example:

Texture\* pTexture = TextureManager::GetInstance()->LoadTexture(“cake.png");

Snacks and Prototypes:

We have to have something to draw to the screen, so we’re going to create a simple class that contains a texture to draw, and a position and velocity vector. T

This class will implement an interface called IPrototype. This interface will have a single abstract virtual function Create(), which will be overloaded by our sub-class Snack.

Here is the declaration for the IPrototype interface, Snack class, and Vector2 struct:

class IPrototype {

public:

IPrototype() {};

virtual ~IPrototype() {};

virtual IPrototype\* Clone() = 0;

};

class Texture;

class SpriteBatch;

struct Vector2 {

float x, y;

};

class Snack : public IPrototype {

public:

Snack(Texture\* pTexture);

Snack(const char\* filename);

Snack(const Snack& other);

~Snack() {};

Snack& operator=(const Snack& other);

virtual IPrototype\* Clone();

bool IsAlive() {return m\_bAlive;}

Vector2 GetPosition() {return m\_position;}

void SetAlive(bool state) {m\_bAlive = state;}

void SetVelocity(float x, float y);

void SetPosition(float x, float y);

void Update(float dt);

void Draw(SpriteBatch\* spritebatch);

private:

Texture\* m\_pTexture;

Vector2 m\_position;

Vector2 m\_velocity;

bool m\_bAlive;

};

We’ve added the copy constructor and assignment operator to make creating the clones of the prototypes easy, but the main function to take note of is the Clone() function. This is what will return a clone of the object, ensuring that we can implement the Prototype design pattern.

Here is the definition of our Snack class. Note the call to the TextureManager in the constructor.

#include "Snack.h"

#include "Texture.h"

#include "TextureManager.h"

#include "SpriteBatch.h"

Snack::Snack(Texture\* pTexture){

m\_position.x = 0; m\_position.y = 0;

m\_velocity.x = 0; m\_velocity.y = 0;

m\_pTexture = pTexture;

}

Snack::Snack(const char\* filename){

m\_position.x = 0; m\_position.y = 0;

m\_velocity.x = 0; m\_velocity.y = 0;

m\_pTexture = TextureManager::GetInstance()->LoadTexture(filename);

}

Snack::Snack(const Snack& other){

m\_pTexture = other.m\_pTexture;

m\_position = other.m\_position;

m\_velocity = other.m\_velocity;

}

Snack& Snack::operator=(const Snack& other){

m\_pTexture = other.m\_pTexture;

m\_position = other.m\_position;

m\_velocity = other.m\_velocity;

return \*this;

}

IPrototype\* Snack::Clone(){

return new Snack(\*this);

}

void Snack::SetPosition(float x, float y){

m\_position.x = x;

m\_position.y = y;

}

void Snack::SetVelocity(float x, float y){

m\_velocity.x = x;

m\_velocity.y = y;

}

void Snack::Update(float dt){

m\_position.x += m\_velocity.x \* dt;

m\_position.y += m\_velocity.y \* dt;

}

void Snack::Draw(SpriteBatch\* spritebatch)

{

spritebatch->DrawSprite(m\_pTexture, m\_position.x, m\_position.y);

}

The Snack Factory:

The final piece of our puzzle is to create a Factory class to create our game objects. This will allow us to pass a string (like “cake”) to the factory’s Create() function and get a Snack class object back that has been initialized with the variables needed to display a cake.

The lecture slides used a series of IF statements with hard-coded strings to implement this. This class will use something slightly different.

The SnackFactory contains a Map of key-value pairs. The string key will be the name (like “cake”), and the Snack value will hold the prototype we’ll use when creating the object (this is where our Prototype pattern comes in to play). We will add new prototypes to the map using the AddPrototype function.

The declaration of the SnackFactory class is as follows:

#include <map>

class IPrototype;

class SnackFactory

{

public:

SnackFactory();

~SnackFactory();

void AddPrototype(const char\* name, IPrototype\* prototype);

IPrototype\* Create(const char\* name);

private:

std::map<std::string, IPrototype\*> m\_prototypes;

};

The definition is similarly quite simple:

#include "SnackFactory.h"

#include "IPrototype.h"

SnackFactory::SnackFactory(){}

SnackFactory::~SnackFactory(){

for(auto it = m\_prototypes.begin(); it != m\_prototypes.end(); ++it) {

delete it->second;

}

}

void SnackFactory::AddPrototype(const char\* name, IPrototype\* prototype){

m\_prototypes[name] = prototype;

}

IPrototype\* SnackFactory::Create(const char\* name){

return m\_prototypes[name]->Clone();

}

The ‘Game’:

Add the following member variables to your Game1 application class:

SnackFactory\* m\_pSnackFactory;

std::list<Snack\*> m\_snacks;

The first thing we need to do in our Game constructor is create the SnackFactory and initialize it with some Snack Prototypes.

Game1::Game1(unsigned int windowWidth, unsigned int windowHeight, bool fullscreen, const char \*title) : Application(windowWidth, windowHeight, fullscreen, title)

{

srand(time(NULL));

m\_spritebatch = SpriteBatch::Factory::Create(this, SpriteBatch::GL3);

m\_pSnackFactory = new SnackFactory();

// create some snacks to store in the factory

Snack\* cake = new Snack("./Images/cake.png");

cake->SetPosition(200, 100);

Snack\* croissant = new Snack("./Images/croissant.png");

croissant->SetPosition(500, 400);

m\_pSnackFactory->AddPrototype( "cake", cake);

m\_pSnackFactory->AddPrototype( "croissant", croissant);

}

And don’t forget the destructor:

Game1::~Game1()

{

delete m\_pSnackFactory;

TextureManager::DestroyInstance();

SpriteBatch::Factory::Destroy(m\_spritebatch);

}

Now, whenever we want to create a new cake or croissant, we can simply call the SnackFactory and pass in the appropriate string.

In fact, that’s what we’ll do in the Update function. So that we can make lots of Snacks we’ll have a timer that will create a random snack every half second. Once we’ve created the snack we’ll assign it a random velocity. And of course we’ll kill the snack once it goes off the screen.

void Game1::Update(float deltaTime){

static float timer = 0;

if(timer <= 0) {

timer = 0.5f;

Snack\* snack;

if(rand()%2 == 0)

snack = (Snack\*) m\_pSnackFactory->Create("cake");

else

snack = (Snack\*) m\_pSnackFactory->Create("croissant");

snack->SetVelocity(50-rand()%100, 50-rand()%100);

m\_snacks.push\_back( snack );

}

timer -= deltaTime;

for(auto it = m\_snacks.begin(); it != m\_snacks.end(); ++it) {

(\*it)->Update(deltaTime);

Vector2 pos = (\*it)->GetPosition();

if(pos.x < 0 || pos.y < 0 ||

pos.x > this->GetViewWidth() || pos.y > this->GetWindowHeight()) {

delete (\*it);

it = m\_snacks.erase(it);

if(it == m\_snacks.end())

break;

}

}

}

You can probably guess what the draw function is going to do… run through the list of snacks and draw them.

void Game1::Draw()

{

// clear the back buffer

ClearScreen();

m\_spritebatch->Begin();

for(Snack\* s : m\_snacks) {

s->Draw(m\_spritebatch);

}

m\_spritebatch->End();

}

That’s it! You should now be able to run your game and see lots of cakes and croissants spawning on the screen.

Exercise:

Currently when a new Snack is created by the factory it will get stored in any random available location on the heap. But creating and destroying so many objects can fragment our memory (which is not good for games)

We could optimize our program by adding an Object Pool.

As explained in the lecture slides an object pool is an array of objects that is allocated when your program launches. Whenever you need to create a new object (like a Snack) you take the first ‘dead’ object out of the pool and reuse it by updating its member variables. When that object dies, rather than delete the object we set its state to ‘dead’ and it goes back in the pool to be reused later.

This has the effect of eliminating memory fragmentation and improving the performance of our game.

Implement an object pool inside the Snack Factory.

Whenever you ask the factory to create a Snack, it will get the first ‘dead’ object from its object pool, reset its member variables according to the prototype, and then return a pointer to the reused object.

The Snack class already has an ‘alive’ member variable, and member functions to modify it, so you should be able to add the object pool to this program by only modifying the SnackFactory class.