SDL:Tutorials:Complete 2D Engine System Core

From GPWiki

Contents

The System Core

What exactly does a system core do?

The system core handles the low down details of creating a window, managing what state the game is in and the game processes. For each part I have created a class, cApp, cStateManager and cProcessManager. I will get into more details of what each class really does later on in this page. The system core could handle more or less than what I am but it all depends on the design of your game.

cStateManager

What is a state manager and why do I need it?

A state manager, like its name implies, keeps track of what state the game is in. To make this more clear, let's break down a game. First you enter the game and the creator's logo or a movie or something is shown. Let's call this the intro state. Next you get to a main menu where you chose if you want to start a new game, save/load a game, enter configuration or quit. We can call this main menu state. Now, after we start a new game, we enter the main game state. As you can see, we need something to tell the game what state we are in. This is what the state manager does.

The cStateManager Class

Here is the class structure for the statemanager I use.

```
class cStateManager
{
    public:
        cStateManager();
        ~cStateManager();
        bool Push( void (*Function)(void* CallerPtr, Purpose Purp),  void* CallerPtr = 
NULL);
        bool Pop( void* CallerPtr = NULL);
        bool PopAll( void* CallerPtr = NULL);
        bool Process( void* CallerPtr = NULL);

    private:
        sState* m_CurrentState;
};
```

The state manager is designed like a stack. You can push a state on to the stack and that becomes the current state when you goto call process. You can Pop the current state to revert to the previous state or you can clear all states with PopAll. The last function is the Process function which calls the function of the current state. After looking at this class your probably thinking well what does this state structure look like? What is this CallerPtr? and if your not familiar with function pointers the Push function has to look a little odd.

The sState structure

To answer your first question the sState is defined as follows.

```
struct sState
{
    sState* Prev;
    void (*Function)(void* CallerPtr, Purpose Purp);

    sState()
    {
        Prev = NULL;
        Function = NULL;
    }

    ~sState()
    {
        delete Prev;
    }
};
```

Basically it has a pointer to the previous state and a function pointer. A function pointer is like a regular pointer but it calls a function when it is dereferenced instead of a value. This particular function takes two arguments, the CallerPtr and a Purpose. First let me explain the CallerPtr. If you haven't figured it out it is just a pointer to what called the function.

The ePurpose enum

The purpose tells the function why it was called so it can perform accordingly. Here is the purpose class.

```
enum Purpose
{
    STOP_PURPOSE = 0,
    INIT_PURPOSE,
    FRAME_PURPOSE,
    NO_PURPOSE
};
```

This is pretty self explanatory, it has four states so the function know what to do. STOP_PURPOSE to tell the function it is being stopped, INIT_PURPOSE to tell the function it is being initialized, FRAME_PURPOSE to tell the function that it is being called inside the main game loop (frame) and NO_PURPOSE for anything that doesn't fit in those categories.

Example in action

Now that you have seen the code for the state manager you might be wondering how to use it. Here is a really simple example

```
void fun1(void* CallerPtr, Purpose Purp)
{
    switch(Purp)
    {
        case STOP_PURPOSE:
            cout << "fun1 Stopping" << endl;
            break;
        case INIT_PURPOSE:
            cout << "fun1 Starting" << endl;
            break;
        case FRAME_PURPOSE:
            cout << "fun1 processing" << endl;
            break;
        default:
            cout << "fun1 no purpose" << endl;
            break;
    }
}

void fun2(void* CallerPtr, Purpose Purp)
{

    switch(Purp)
    {
        case STOP_PURPOSE:
            cout << "fun2 Stopping" << endl;
            break;
        case INIT_PURPOSE:
            cout << "fun2 Starting" << endl;
            break;
        case FRAME_PURPOSE:
            cout << "fun2 processing" << endl;
            break;
        default:
            cout << "fun2 no purpose" << endl;
            break;
    }

}

void fun3(void* CallerPtr, Purpose Purp)
{

    switch(Purp)
    {
        case STOP_PURPOSE:
            cout << "fun3 Stopping" << endl;
            break;
        case INIT_PURPOSE:
            cout << "fun3 Starting" << endl;
            break;
        case FRAME_PURPOSE:
            cout << "fun3 processing" << endl;
            break;
        default:
            cout << "fun3 no purpose" << endl;
            break;
    }

}

cStateManager StateMan;

cout << "  <process>" << endl;
StateMan.Process(NULL);

cout << "  <change to fun1>" << endl;
StateMan.Push(fun1, NULL);

cout << "  <change to fun2>" << endl;
StateMan.Push(fun2, NULL);

cout << "  <process>" << endl;
StateMan.Process(NULL);

cout << "  <kill fun2>" << endl;
StateMan.Pop();

cout << "  <change to fun3>" << endl;
StateMan.Push(fun3);

cout << "  <process>" << endl;
StateMan.Process(NULL);
```

```
cout << "  <kill all>" << endl;
StateMan.PopAll(NULL);

cout << "  <process>" << endl;
StateMan.Process(NULL);
```

Basically all the 3 functions do is say how it is called. Also you will notice I put in NULL for the CallerPtr on all functions, for now I am putting NULL but later on I will show you how this is used (in cApp). Ok so what does this code produce?

```

  <process>
  <change to fun1>
fun1 Starting
  <change to fun2>
fun2 Starting
  <process>
fun2 processing
  <kill fun2>
fun2 Stopping
  <change to fun3>
fun3 Starting
  <process>
fun3 processing
  <kill all>
fun3 Stopping
fun1 Stopping
  <process>
```

For clarity I will explain what is going on. First we call process but there are no states so nothing happens, next we push on fun1 which starts the initialization of that function. Same happens with fun2 as fun1, then we call process again. Notice how it only runs fun2 as that is the current state. Next we stop fun2 and it makes a call telling the function so. Now we add fun3 to the stack, the function is initialized then we PopAll which calls the shutdown of fun3 and fun1. Notice that this shutdown sequence is in reverse order of how it was called. Finally process is called again and of course nothing is there so nothing happens.

cProcessManager

What exactly is a process manager, is it useful?

In this tutorial a process manager is a collection of processes. When a process manager runs it runs all the processes in the set. Ok so why is this useful? Sometimes within a state you wish to have a dynamic list of processes each frame. For an example when you are running in your "main game state" you may wish to draw something extra such as the fps if it is in debug mode or some other information that you find useful. A process manager handles adding functions to your game without having to have ugly 'switch' or 'for' statements all over your code. It also helps make using your state manager easier to use, many of times in a state you want to run more than one function each frame, a process manager handles this. To make this clearer lets say in a certain state you want to handle input, rendering, updating your players and playing sounds. A process manager would combine all these separate functions into one so all of them are called when you tell the process manager to do its job.

The cProcessManager class

Here is my implementation of the cProcessManager class.

```
class cProcessManager
{
    public:
        cProcessManager();
        ~cProcessManager();
        int Push( void (*Function)(void* CallerPtr, Purpose Purp),  void* CallerPtr = NULL);
        bool Pop( unsigned int id, void* CallerPtr = NULL);
        bool PopAll( void* CallerPtr = NULL);
        bool Process( void* CallerPtr = NULL);

    private:
        sProcess* m_FirstProcess;
        sProcess* m_LastProcess;
};
```

As you can see this is very similar to the state manager class. You can push and pop processes just like a state manager but wait is this 'id' variable? Simply put the id is a unique identifier for each process in the manager. When you push a process it either returns 0 for failure or a positive number indicating what id was assigned to that process. Similarly the pop takes an id to remove from the manager. The last function Process goes through all processes in the order that they were added. Simple enough right?

The sProcess structure

```
struct sProcess
{
    sProcess* Next;
    sProcess* Prev;
    void (*Function)(void* CallerPtr, Purpose Purp);

    unsigned int Id;

    sProcess()
    {
        Prev = NULL;
        Next = NULL;
        Function = NULL;
        Id = 0;
    }

    ~sProcess()
    {
        delete Prev;
        delete Next;
    }
};
```

Like before this structure is similar to the 'sState' structure. There really is little difference, I have added 'Id' to the mix for process identification and a pointer to the previous process to make my life easier when coding the process manager (see the code and you can see why).

An Example

Since I have explained the functionality of the state manager and what the purposes are for this example should be straight forward.

```
cProcessManager ProcMan;

    cout << "  <process>" << endl;
    ProcMan.Process(NULL);

    cout << "  <create fun1>" << endl;
    int id1 = ProcMan.Push(fun1, NULL);

    cout << "  <create fun2>" << endl;
    ProcMan.Push(fun2, NULL);

    cout << "  <process>" << endl;
    ProcMan.Process(NULL);
    cout << "  <kill fun1>" << endl;
    ProcMan.Pop(id1);

    cout << "  <create fun 3>" << endl;
    ProcMan.Push(fun3);

    cout << "  <process>" << endl;
    ProcMan.Process(NULL);

    cout << "  <kill all>" << endl;
    ProcMan.PopAll(NULL);

    cout << "  <process>" << endl;
    ProcMan.Process(NULL);
```

Note that I have omitted the fun1, fun2 and fun3 functions but they are the same as I used in the statemanager. Also this code is shockingly similar to the state manager code, in fact all I did was create a 'cProcessManager' instead of a 'cStateManager' and change StateMan to ProcMan. The only big difference if you call it that is how you stop a process. When you stop a process you have to give it an id for which one to stop. The reason I did this is to show you difference between the two managers. So onward with the results.

```
  <process>
  <create fun1>
fun1 Starting
  <create fun2>
fun2 Starting
  <process>
fun1 processing
fun2 processing
  <kill fun1>
fun1 Stopping
  <create fun 3>
fun3 Starting
  <process>
fun2 processing
fun3 processing
  <kill all>
fun2 Stopping
fun3 Stopping
  <process>
```

There are a few subtle difference between the outputs of each manager. The first one is when you call process, all functions are called in the order that they are added. The second difference is when you goto PopAll processes, instead of going in reverse order like a stack you stop the processes in order of how they were added.

cApp

cApp? what's a cApp?

Finally we get to the last class in the system core. The 'cApp' class handles the basic initialization of the window and has a 'Run' function which starts our main game. Without further ado here it is.

```
class cApp
{
    public:
        cApp();
        virtual ~cApp();

        virtual void Init() = 0;
        virtual void Run() = 0;
        virtual void Shutdown() = 0;
        virtual void ResizeWindow(int width, int height) = 0;
        virtual void ToggleFullscreen() = 0;

    protected:
        int m_Width;
        int m_Height;
        int m_Bpp;
        std::string m_WindowName;
        bool m_FullScreen;

        SDL_Surface* m_Surface;
        Uint32 m_SdlFlags;
};
```

First thing I notice when I look at this is that the class actually does nothing, meaning that you will have to implement the 'cApp' class yourself according to how you want the game to run. So if the 'cApp' class was implemented statically what would be in the functions you ask? Well basically the 'Init' function initializes the windowing system and all the libraries that it will be using. In my source code you will see I initialize the window via SDL and then I setup a little bit of OpenGl, however I will probably be moving these OpenGL functions to the graphics core later on. The next function is called 'Run', in this function I don't have much of anything in my code at the moment. What should be there is setting up a default state then starting it up, in otherwords your main game loop. Then we have the 'Shutdown' function, which does what you would think, it shutsdown the game. For now 'Shutdown' is mostly blank in my code. Next there is the 'ResizeWindow' function which takes a width and a height, again this function actually doesn't do anything. Last but not least we have 'ToggleFullscreen', when implemented it does as it says. It switches between fullscreen and windowed mode. Now we have quite a few variables, if you are familiar with SDL then you shouldn't have a problem figuring out what they are for.

cApp doesn't do anything, how does this help?

cApp is just an interface it provides structure on how a window should be initialized, run, shutdown and whatever other functions you want the window to do. The purpose of this tutorial series isn't to teach you about SDL or whatever library you are using, it is about designing the game engine so I won't go into details about how I implemented these functions (you can read my source code if you really want to know). There are plenty of tutorials on this site to help you get started on implementing these functions if you are lost.

Wrapping it all up

If you were able to follow all that code above then good for you. You are on your way to making a great engine. With the system core you can now create a window, tell the game what state it is in and setup processes. In other words you have a nice framework to build on. Keep in mind this is just an example system core, your core could be wildly different, add new features or even take away features you don't need. Everything depends on what your game calls for so design your game first and then make your engine with the features you need.

About My Source Code

Seeing as this is the first tutorial in the series that I have uploaded source code for I need to tell you a few things about it. My code is a complete XCode 2.2 project meaning that you can compile it on any Mac with XCode installed. XCode uses GCC version 3.3 so my code will be portable to any other platform you see fit, the only thing you will have to change is the headers. The point of providing this source code is to show you how I implemented the engine and a real world example of how it works, in most cases you will not need to compile to understand what the code does as I try to comment as much as I can. With that being said here is my code for this tutorial, if you happen to use it in a game I would like to know.

Get The Source-> Source Code

This tutorial was written by Seoushi. Do you like this tutorial? Have any questions or comments? Let me know or you can ask the forums.

Retrieved from "http://gpwiki.org/index.php/SDL:Tutorials:Complete_2D_Engine_System_Core"

Categories: C and SDL | All C articles | All SDL articles
