Ogre Wiki

Support and community documentation for Ogre3D

I needed a GameStateManager for my current project and I got most of the information for it off of Managing Game States with OGRE; however, I believe the improvements I made may be of benefit to all.

gamestate.h - Inherit class GameState to create a manageable GameState class.
Don't forget the DECLARE_GAMESTATE_CLASS(yourclassname).

```
#ifndef __gamestate_h__
#define __gamestate_h__

#include "global.h"

class GameState;

/** \class GameStateListener
    Allows a GameStateManager to recieve callbacks
    from a GameState. */
class GameStateListener
{
public:
    /** Constructor */
    GameStateListener(void) {};

    /** Virtual Deconstructor */
    virtual ~GameStateListener(void) {};

    /** Store a game state to manage. */
    virtual void ManageGameState(Ogre::String state_name,GameState*state)=0;

    /** Find a state by name. */
    virtual GameState *findByName(Ogre::String state_name)=0;

    /** Request a change to state. */
    virtual void changeGameState(GameState *state)=0;

    /** Push state onto the stack. */
    virtual bool pushGameState(GameState* state)=0;

    /** Pop a game state off the stack. */
    virtual void popGameState()=0;

    /** Cause a shutdown. */
    virtual void Shutdown()=0;

};

/** \class GameState
    Inherit this class to make a game state capable
    of being mananged by the game state manager.
    Be sure to use DECLARE_GAMESTATE_CLASS(class)
    in your public section.
*/
class GameState: public Ogre::FrameListener,
```

```
    public OIS::KeyListener,
    public OIS::MouseListener
{
public:
    /** Do not inherit this directly! Use DECLARE_GAMESTATE_CLASS (class) to
do it for you. */
    static void Create(GameStateListener *parent, const Ogre::String name) {}
;

    /** Destroy self. */
    void destroy(void)
    { delete this; }

    /** Initialize the game state with device information. */
    void init(device_info *devices)
    { mDevice=devices; }

    /** Inherit to supply game state enter code. */
    virtual void enter(void)=0;
    /** Inherit to supply state exit code. */
    virtual void exit(void)=0;
    /** Inherit to supply pause code. Inherit only if this game state can be
paused.
        Return true for successful pause, or false to deny pause. */
    virtual bool pause(void)
    { return false; }
    /** Inherit to supply resume code. Inherit only if this game state can be
paused. */
    virtual void resume(void) {};

protected:
    /** Constructor: This should be a private member of an inherited class. *
/
    GameState(void) {};
    /** Destructor: This should be a private member of an inherited class. */
    virtual ~GameState(void) {};

    /** Find a state by its name. */
    GameState *findByName(Ogre::String state_name)
    { return parent->findByName(state_name); }

    /** Request a change to game state. */
    void changeGameState(GameState *state)
    { parent->changeGameState(state); }

    /** Push game state onto the stack. */
```

```
    bool pushGameState(GameState* state)
    { return parent->pushGameState(state); }

    /** Pop a game state off the stack. */
    void popGameState(void)
    { parent->popGameState(); }

    /** Cause a shutdown. */
    void Shutdown(void)
    { parent->Shutdown(); }

    /** Stores the GameStateManager which is managing this state. */
    GameStateListener *parent;
    /** Keeps a method of device interaction. */
    device_info *mDevice;
};

/** Create the game state. Inherit, Create your class, and have it managed.
    Example:
    \code
    static void MyGameStateClass::Create(GameStateListener *parent,
            const Ogre::String name)
    {
        myGameStateClass myGameState=new myGameStateClass();
        myGameState->parent=parent;
        parent->ManageGameState(name,myGameState);
    }
    \endcode
*/
#define DECLARE_GAMESTATE_CLASS(T) static void Create(GameStateListener *pare
nt,const Ogre::String name)    {                       T *myGameState=new T(
);                    myGameState->parent=parent;               parent->Mana
geGameState(name,myGameState);        }

#endif
```

gamestatemanager.h. - Include this where you want to create your state manager.

```
#ifndef __gamestatemanager_h__
#define __gamestatemanager_h__

#include "global.h"
#include "gamestate.h"

/** \class GameStateManager
    The GameStateManager manages changes in the game states
    in the game.  It holds a stack of all open game states
    and maps all events. */
class GameStateManager: public GameStateListener
{
public:
    /** Holds information about the states in order to
        manage them properly and provide access. */
    typedef struct
    {
        Ogre::String name;
        GameState *state;
    } state_info;

    /** Constructs the GameStateManager.  Must have all
        input, output, gui functions in order to manage
        states. */
    GameStateManager(device_info *devices);

    /** Cleans up the states before the instance dies. */
    ~GameStateManager();

    /** Store a game state to manage. */
    void ManageGameState(Ogre::String state_name,GameState *state);

    /** Find a game state by name. */
    GameState *findByName(Ogre::String state_name);

    /** Start game state */
    void start(GameState *state);

    /** Change to a new game state */
    void changeGameState(GameState* state);

    /** Push game state onto the stack. */
    bool pushGameState(GameState* state);

    /** Pop a game state off the stack. */
    void popGameState(void);
```

```
    /** This is a special case function to cause a shutdown. */
    void Shutdown(void);

protected:
    /** This initializes a state to receive the events. */
    void init(GameState *state);

    /** This removes event handling from a previous state. */
    void cleanup(GameState *state);

    /** This is the stack where the active states are stored. */
    std::vector<GameState*> mStateStack;
    /** This holds the states that are being managed. */
    std::vector<state_info> mStates;
    /** System Information. */
    device_info *mDevice;
    /** If this is set to true, the game state manager prepares to exit. */
    bool mShutdown;
};

#endif //__gamestatemanager_h__
```

gamestatemanager.cpp - Implementation of the game state manager.

```
#include "gamestatemanager.h"

/**    Constructs the GameStateManager.  Must have all
    input, output, gui functions in order to manage
    states. */
GameStateManager::GameStateManager(device_info *devices)
{
    mDevice=devices;
}

/** Cleans up the game states before the instance dies. */
GameStateManager::~GameStateManager()
{
    // clean up all the states on the stack
    while (!mStateStack.empty())
    {
        cleanup(mStateStack.back());
        mStateStack.back()->exit();
        mStateStack.pop_back();
    }

    // destroy the game states
    while(!mStates.empty())
    {
        mStates.back().state->destroy();
        mStates.pop_back();
    }
}

/** Store a game state to manage. */
void GameStateManager::ManageGameState(Ogre::String state_name,GameState *sta
te)
{
    state_info new_state_info;
    new_state_info.name=state_name;
    new_state_info.state=state;
    new_state_info.state->init(mDevice);
    mStates.push_back(new_state_info);
}

/** Find a game state by name.
    @Remarks returns 0 on failure.*/
GameState *GameStateManager::findByName(Ogre::String state_name)
{
    std::vector<state_info>::iterator itr;
```

```
    for(itr=mStates.begin();itr!=mStates.end();itr++)
    {
        if(itr->name==state_name)
            return itr->state;
    }

    return 0;
}

/**  Start game state. This is used to start the game state
     manager functioning with a particular state.
     This function also does the main game loop and
     takes care of the Windows message pump.*/
void GameStateManager::start(GameState *state)
{
    changeGameState(state);

    while (!mShutdown)
    {

        mDevice->keyboard->capture();
        mDevice->mouse->capture();

        // run the message pump
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        {
            MSG msg;
            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                 if (msg.message == WM_QUIT)
                     Shutdown();
                 else
                 {
                     TranslateMessage(&msg);
                     DispatchMessage(&msg);
                 }
            }
        }
#endif
        mDevice->ogre->renderOneFrame();
    }
}

/** Change to a game state.  This replaces the current game state
    with a new game state.  The current game state ends before
    the new begins. */
```

```
void GameStateManager::changeGameState(GameState* state)
{
    // cleanup the current game state
    if ( !mStateStack.empty() )
    {
        cleanup(mStateStack.back());
        mStateStack.back()->exit();
        mStateStack.pop_back();
    }

    // store and init the new game state
    mStateStack.push_back(state);
    init(state);
    mStateStack.back()->enter();
}

/** Push a game state onto the stack. This pauses the current game state
    and begins a new game state. If the current game state refuses to
    be paused, this will return false. */
bool GameStateManager::pushGameState(GameState* state)
{
    // pause current game state
    if ( !mStateStack.empty() )
    {
        if(!mStateStack.back()->pause())
            return false;
        cleanup(mStateStack.back());
    }

    // store and init the new state
    mStateStack.push_back(state);
    init(state);
    mStateStack.back()->enter();

    return true;
}

/** Pop a game state off the stack. This destroys the current game state
    and returns control to the previous game state. */
void GameStateManager::popGameState(void)
{
    // cleanup the current game state
    if ( !mStateStack.empty() )
    {
        cleanup(mStateStack.back());
        mStateStack.back()->exit();
```

```
            mStateStack.pop_back();
        }

        // resume previous game state or quit if there isn't one
        if ( !mStateStack.empty() )
        {
            init(mStateStack.back());
            mStateStack.back()->resume();
        }
            else
                    Shutdown();
    }

    /** Special case function to shutdown the system. */
    void GameStateManager::Shutdown()
    {
        mShutdown=true;
    }

    /** This initializes a game state to receive the events. */
    void GameStateManager::init(GameState *state)
    {
        mDevice->ogre->addFrameListener(state);
        mDevice->keyboard->setEventCallback(state);
        mDevice->mouse->setEventCallback(state);
    }

    /** This removes event handling from a previous game state. */
    void GameStateManager::cleanup(GameState *state)
    {
        mDevice->ogre->removeFrameListener(state);
    }
```

The actual program flow is very similar to the GameManager class in the wiki. The major difference here is that the GameStateManager will keep track of your state objects for you and let you look them up by name. This keeps you from having to have a singleton GameStateManager and global states.

In order to use this GameStateManager:

```
//Create the game state manager here.
GameStateManager GameStateMgr(&device);

//Create the game states.
TitleScreen::Create(&GameStateMgr,"TitleScreen");
MenuScreen::Create(&GameStateMgr,"MenuScreen");
PlayScreen::Create(&GameStateMgr,"PlayScreen");

//Start the game state manager running.
GameStateMgr.start(GameStateMgr.findByName("TitleScreen"));
```

When you want to call another game state to run (so that when you exit it you are back to the current game state), you simply call:
pushGameState(findByName("statename"));
from your game state class.

If you want to change game states so that you never come back to the current game state, call:
changeGameState(findByName("statename"));

When a state is done, it exits by calling:
popGameState();

If a game state should not be pausable for whatever reason (perhaps because it must stay up to date with the network) then do not overide the pause() or resume() functions and the game state will refuse to be paused.

I hope someone gets some use out of this.

BTW, I realize that GameState::Create() does not return a pointer to the state, this is because I didn't want to give the false impression that you needed to delete the game state. The game state manager will free them for you. If you are in dire need of a pointer, you can call GameStateManager::findByName("name of game state") as soon as the game state has been created to get the GameState*.

The device_info structure that is passed along is something that I use to send pointers to the game states for global game objects without having to use global variables. My current device_info structure looks like:

```
typedef struct
{
        Orbits::ConfigManager *config;
    Ogre::Root *ogre;
    Ogre::RenderWindow *rwindow;
    OIS::InputManager *InputMgr;
    OIS::Keyboard *keyboard;
    OIS::Mouse *mouse;
    CEGUI::OgreCEGUIRenderer *GUIRenderer;
    CEGUI::System *GUISystem;
} device_info;
```

But this is probably something that will be implementation specific.

Contributors to this page: jacmoe and OgreWikiBot .
Page last modified on Saturday 02 of January, 2010 16:58:21 PST by jacmoe.

Source History

Search by Tags

Search Wiki by Freetags

Latest Changes

1. Ogre 2.1 FAQ
2. ManualObject
3. Advanced Mogre Framework
4. OgreBites
5. Advanced Ogre Framework
6. Tutorials
7. Basic Tutorial 2
8. Basic Tutorial 1
9. Basic Tutorial Introduction
10. CMake Quick Start Guide

...more

Search

Find

Go

Advanced Search Help

Online Users

21 online users