Profile Class

Intro

Casual game titles frequently use player profiles to allow more than one person to play a game on each install. This is commonly referred to as 'Player Profiles'.

Profiles can contain all sorts of information, such as current level, player achievements, other game progress extras, etc... The type of information is limitless. Basically, any information that can vary from player to player.

The problem is storing and retrieving this information while developing a game. The information format, types or variables can change during game development. Each time a new setting is added, previous file formats become useless.

The information stored in the user profile is typically one of the 4 default types {int, bool, float, string}. Using this insight, we can develop a universal format for storing profile settings. Each setting will have a string identifier key and a value. Using the XML format, it is easy to store associative key values and their data.

Basic Use

The Profile class is a Singleton instance. This is so that you can access it from anywhere in the program without having to pass the pointer around.

```
	Profile::GetProfile()
```

Load Previous

The first thing you need to do is load the user. It's common to load the last user from the previous game session. This can easily be done by using a 'LastUser' Registry key:

```
	SexyString aPreviousUser;
	gSexyAppBase->RegistryReadString("LastUser", &aPreviousUser);
	if(aPreviousUser != "")
		GameState::Profile::GetProfile()->LoadUser(aPreviousUser);
```

If the "LastUser" registry key doesn't exist, then you should run a dialog box to get a new user name.

Save to Registry

During shutdown, it's easy to save the "LastUser" key:

```
// Store the last user playing the game
gSexyAppBase->RegistryWriteString("LastUser", Profile::GetProfile()->GetUserName());
Profile::GetProfile()->SaveUser();
```

Get Values

To retrieve a profile variable, you simple need to know the string identifier for the variable and the variable type. You can also supply a default value for the Profile to return in case the value doesn't exist or hasn't been set yet:

```
int aLevel = Profile::GetProfile()->GetIntegerValue(_S("User_Current_Level"), 0);
MasterGameLogic::GetLogicController()->StartLevel(aLevel);
```

In the previous example, the current level will return '0' if the 'User_Current_Level' doesn't exist.

Set Values

You can set a profile value like so:

```
        Profile::GetProfile()->SetIntegerValue(_S("User_Current_Level"), 0);
```

Advanced Use

When trying to save a game state or a type other than {int, bool float, string} it is possible to save the data directly to a file. The profile class provides a simple mechanism for organizing the state files.

```
        std::string aSaveFile =
                Profile::GetProfile()->GetStateFileName(_S("SaveState"));
        Buffer aBuffer;

//      aBuffer.WriteLong(HINTS_META_TAG);
//      aBuffer.WriteLong(mHints);
//
//      TODO: Save Master Game State Data

        if(gSexyAppBase->WriteBufferToFile(aSaveFile, &aBuffer))
        {
                Profile::GetProfile()->SetBoolValue(_S("USER_HAS_SAVE"), true);
                Profile::GetProfile()->SetStringValue(_S("USER_SAVE_PATH"), (aSaveFile));
                Profile::GetProfile()->SaveUser();
        }
        else
        {
                Profile::GetProfile()->SetBoolValue(_S("USER_HAS_SAVE"), false);
                Profile::GetProfile()->SetStringValue(_S("USER_SAVE_PATH"), _S(""));
                Profile::GetProfile()->SaveUser();
        }
```

The most important part of this example is '`GetStateFileName`'. This method returns the file name of a state file for the current user. It is composed of

"%APP_DATA_FOLDER% / users / %PROFILE_USERNAME% / %STATE_NAME% .sav"

The users directory will look something like this:

```
  |
  |- Users /
      |-Bob.xml
      |   |- Bob /
      |       |- BoardState.sav
      |       |- MasterState.sav
```

Non-XML Serialization

The user profile variables are stored in XML format. Values that are not set do not have XML entries in this file. This makes it harder for users to Hack: let's pretend that you have a super secret achievement named 'HAS_SUPER_SECRET'. If the value existed in the XML, then the user could hack the file and turn on the achievement.

However, some developers will forgo an XML format altogether. It is possibl e to use a buffer to save the profile:

```
bool Profile::Save(SexyString theFileName)
{
        Buffer aBuffer;
        aBuffer.WriteLong(CFG_FILE_MAX_VERSION - 1); // Write Latest Version
        // Write Version 3
        {
        }
        // Write Version 2
        {
        }
        // Write Version 1
        {
                // First: Write Booleans
                // Format: SIZE : { (STRING, VALUE), (STRING, VALUE) , ... , SIZE - 1}
                aBuffer.WriteLong((long)mBoolMap.size());
                std::map<SexyString, bool>::iterator bool_itr = mBoolMap.begin();
                for (; bool_itr != mBoolMap.end(); ++bool_itr)
                {
                        aBuffer.WriteString(bool_itr->first);
                        aBuffer.WriteBoolean(bool_itr->second);
                }

                // Then : Write Integers
                // Format: SIZE : { (STRING, VALUE), (STRING, VALUE) , ... , SIZE - 1}
                aBuffer.WriteLong((long)mIntegerMap.size());
                std::map<SexyString, int>::iterator int_itr = mIntegerMap.begin();
                for (; int_itr != mIntegerMap.end(); ++int_itr)
                {
                        aBuffer.WriteString(int_itr->first);
                        aBuffer.WriteLong((long)int_itr->second);
                }

                // Then : Write Floats
                // Format: SIZE : { (STRING, VALUE), (STRING, VALUE) , ... , SIZE - 1}
                aBuffer.WriteLong((long)mFloatMap.size());
                std::map<SexyString, double>::iterator float_itr = mFloatMap.begin();
                for (; float_itr != mFloatMap.end(); ++float_itr)
                {
                        aBuffer.WriteString(float_itr->first);
                        aBuffer.WriteString(StrFormat("%f", float_itr->second));
                }

                // Finally : Write Strings
                // Format: SIZE : { (STRING, VALUE), (STRING, VALUE) , ... , SIZE - 1}
                aBuffer.WriteLong((long)mStringMap.size());
                std::map<SexyString, SexyString>::iterator str_itr = mStringMap.begin();
                for (; str_itr != mStringMap.end(); ++str_itr)
                {
                        aBuffer.WriteString(str_itr->first);
                        aBuffer.WriteString(str_itr->second);
                }
        }
        if (gSexyAppBase->WriteBufferToFile(theFileName, &aBuffer))
                return true;
        return false;
}
```

```
bool  Profile::Load(SexyString theFileName)
{
        if (FileExists(theFileName))
        {
                Buffer aBuffer;
                if (gSexyAppBase->ReadBufferFromFile(theFileName, &aBuffer))
                {
                        long aFileVersion = aBuffer.ReadLong();
                        switch(aFileVersion)
                        {
                        //case CFG_FILE_VERSION_3: // Do Not break, Fall through will read
                        //case CFG_FILE_VERSION_2: // Older File Versions
                        case CFG_FILE_VERSION_1:
                        {
                                long aSize = 0;
                                // First read bools
                                aSize = aBuffer.ReadLong();
                                for (long i = 0; i < aSize; ++i)
                                {
                                        SexyString aKey = aBuffer.ReadString();
                                        bool aValue = aBuffer.ReadBoolean();
                                mBoolMap.insert(std::pair<SexyString, bool>(aKey, aValue));
                                }
                                // Then read Integers
                                aSize = aBuffer.ReadLong();
                                for (long i = 0; i < aSize; ++i)
                                {
                                        SexyString aKey = aBuffer.ReadString();
                                        int aValue = (int)aBuffer.ReadLong();
                                mIntegerMap.insert(std::pair<SexyString, int>(aKey, aValue));
                                }
                                // then read Floats
                                aSize = aBuffer.ReadLong();
                                for (long i = 0; i < aSize; ++i)
                                {
                                        SexyString aKey = aBuffer.ReadString();
                                        double aValue = atof(aBuffer.ReadString().c_str());
                                mFloatMap.insert(std::pair<SexyString, double>(aKey, aValue));
                                }
                                // then read Floats
                                aSize = aBuffer.ReadLong();
                                for (long i = 0; i < aSize; ++i)
                                {
                                        SexyString aKey = aBuffer.ReadString();
                                        SexyString aValue = aBuffer.ReadString();
                                mStringMap.insert(std::pair<SexyString, SexyString>(aKey, aValue));
                                }
                                break;
                        }
                }
                return true;
                }
        }
        return false;
}
```

This approach uses a fall-through approach for extending the file format. Newer versions of the file will be added to the beginning of the stream during 'Save' so that the switch in the 'Load' method can use the 'fall through' mechanism to read the format.

Older versions of the program will not be able to read newer versions of the file, but all old saves will be compatible with future modifications.