Angel 3.2

Generated by Doxygen 1.8.3

Tue Oct 8 2013 02:26:27

Contents

1 The Angel Engine **2**
1.1 ReadMe . . . 2
1.2 Introduction . . . 2
1.3 A Brief Tour of the Code . . . 2
1.4 The World . . . 4
1.4.1 Logging . . . 5
1.4.2 Preferences . . . 5
1.4.3 Text Rendering . . . 6
1.5 Actors . . . 6
1.5.1 Appearance . . . 6
1.5.2 Names and Tags . . . 7
1.5.3 Intervals . . . 8
1.5.4 Subclasses . . . 8
1.5.5 Camera . . . 8
1.5.6 GridActors . . . 9
1.5.7 FullScreenActors . . . 9
1.5.8 HUDActors . . . 9
1.5.9 ParticleActors . . . 9
1.5.10 TextActors . . . 9
1.6 Messaging . . . 9
1.7 Physics . . . 10
1.7.1 Side-Blockers . . . 10
1.8 Input . . . 11
1.8.1 Mouse . . . 11
1.8.2 Keyboard . . . 11
1.8.3 Controller . . . 11
1.8.4 MultiTouch . . . 12
1.8.5 MobileSimulator . . . 12
1.9 Sound . . . 12
1.10 AI . . . 13
1.10.1 Pathfinding . . . 13
1.10.2 Other AI . . . 13
1.11 Archetypes and Level Files . . . 14
1.11.1 Archetypes . . . 14
1.11.2 Level Files . . . 14
1.12 Lua . . . 15
1.12.1 In-Game Console . . . 15
1.13 Tuning . . . 15

1.14 Working on iOS 16
1.15 Niceties and Handy Doodads 16
1.15.1 Publish Script 16
1.16 Final Thoughts 17
1.17 Bug Reports and Feedback 17

2 Class Documentation **17**
2.1 Accelerometer Class Reference 17
2.1.1 Detailed Description 17
2.1.2 Member Function Documentation 18
2.2 Actor Class Reference 18
2.2.1 Detailed Description 20
2.2.2 Constructor & Destructor Documentation 20
2.2.3 Member Function Documentation 21
2.2.4 Member Data Documentation 32
2.3 AIBrain Class Reference 32
2.3.1 Detailed Description 33
2.4 AIBrainState Class Reference 33
2.4.1 Detailed Description 33
2.5 AIEvent Class Reference 34
2.5.1 Detailed Description 34
2.6 AStarSearch< UserState > Class Template Reference 34
2.6.1 Detailed Description 35
2.7 BoundingBox Struct Reference 35
2.7.1 Detailed Description 36
2.8 Camera Class Reference 36
2.8.1 Detailed Description 37
2.8.2 Member Function Documentation 37
2.9 Color Class Reference 43
2.9.1 Detailed Description 43
2.9.2 Constructor & Destructor Documentation 43
2.9.3 Member Function Documentation 44
2.9.4 Member Data Documentation 44
2.10 CompoundLog Class Reference 45
2.10.1 Detailed Description 45
2.10.2 Member Function Documentation 45
2.11 ConfigUpdater Class Reference 46
2.11.1 Detailed Description 46
2.11.2 Member Function Documentation 47
2.12 Console Class Reference 47

2.12.1 Detailed Description 48
2.12.2 Constructor & Destructor Documentation 48
2.12.3 Member Function Documentation 48
2.13 ConsoleLog Class Reference 51
2.13.1 Detailed Description 52
2.13.2 Member Function Documentation 52
2.14 Controller Class Reference 52
2.14.1 Detailed Description 53
2.14.2 Member Function Documentation 53
2.15 ControllerInput Struct Reference 58
2.15.1 Detailed Description 59
2.16 ControllerManager Class Reference 59
2.16.1 Detailed Description 59
2.16.2 Constructor & Destructor Documentation 60
2.16.3 Member Function Documentation 60
2.17 DebugDrawBase Class Reference 60
2.17.1 Detailed Description 61
2.18 DebugLine Class Reference 61
2.18.1 Detailed Description 62
2.19 DeveloperLog Class Reference 62
2.19.1 Detailed Description 62
2.19.2 Constructor & Destructor Documentation 62
2.19.3 Member Function Documentation 62
2.20 EventHandler Class Reference 63
2.20.1 Detailed Description 63
2.21 FileLog Class Reference 63
2.21.1 Detailed Description 64
2.21.2 Constructor & Destructor Documentation 64
2.21.3 Member Function Documentation 64
2.22 FindNextMoveState Class Reference 65
2.22.1 Detailed Description 65
2.23 FollowMoveState Class Reference 65
2.23.1 Detailed Description 66
2.24 FullScreenActor Class Reference 66
2.24.1 Detailed Description 66
2.24.2 Constructor & Destructor Documentation 66
2.24.3 Member Function Documentation 67
2.25 GameManager Class Reference 67
2.25.1 Detailed Description 68
2.25.2 Constructor & Destructor Documentation 68

2.25.3 Member Function Documentation 68
2.26 GestureData Struct Reference 69
2.26.1 Detailed Description 70
2.26.2 Member Data Documentation 70
2.27 GotoAIEvent Class Reference 70
2.27.1 Detailed Description 71
2.28 GotoTargetAIEvent Class Reference 71
2.28.1 Detailed Description 71
2.29 GridActor Class Reference 71
2.29.1 Detailed Description 72
2.29.2 Constructor & Destructor Documentation 72
2.29.3 Member Function Documentation 72
2.30 GwenRenderer Class Reference 74
2.30.1 Detailed Description 75
2.31 AStarSearch< UserState >::HeapCompare_f Class Reference 75
2.31.1 Detailed Description 75
2.32 HUDActor Class Reference 76
2.32.1 Detailed Description 76
2.32.2 Member Function Documentation 76
2.33 InputBinding Class Reference 77
2.33.1 Detailed Description 77
2.34 InputManager Class Reference 77
2.34.1 Detailed Description 78
2.34.2 Member Function Documentation 78
2.35 Interval< T > Class Template Reference 78
2.35.1 Detailed Description 78
2.35.2 Constructor & Destructor Documentation 79
2.35.3 Member Function Documentation 79
2.36 LoadedVariable Struct Reference 80
2.36.1 Detailed Description 80
2.37 LuaConsole Class Reference 80
2.37.1 Detailed Description 81
2.37.2 Constructor & Destructor Documentation 81
2.37.3 Member Function Documentation 81
2.38 LuaScriptingModule Class Reference 82
2.38.1 Detailed Description 82
2.38.2 Member Function Documentation 82
2.39 MathUtil Class Reference 83
2.39.1 Detailed Description 84
2.39.2 Member Function Documentation 84

2.39.3 Member Data Documentation 94
2.40 Message Class Reference 95
2.40.1 Detailed Description 96
2.40.2 Constructor & Destructor Documentation 96
2.40.3 Member Function Documentation 97
2.41 MessageListener Class Reference 97
2.41.1 Detailed Description 97
2.41.2 Member Function Documentation 98
2.42 MobileSimulator Class Reference 98
2.42.1 Detailed Description 98
2.42.2 Member Function Documentation 98
2.43 MouseListener Class Reference 100
2.43.1 Detailed Description 100
2.43.2 Constructor & Destructor Documentation 100
2.43.3 Member Function Documentation 100
2.44 NamedEventAIEvent Class Reference 101
2.44.1 Detailed Description 102
2.45 AStarSearch< UserState >::Node Class Reference 102
2.45.1 Detailed Description 102
2.46 ParticleActor::Particle Struct Reference 102
2.46.1 Detailed Description 103
2.47 ParticleActor Class Reference 103
2.47.1 Detailed Description 104
2.47.2 Constructor & Destructor Documentation 104
2.47.3 Member Function Documentation 104
2.48 PathFinder Class Reference 107
2.48.1 Detailed Description 108
2.49 PathFinderMove Struct Reference 108
2.49.1 Detailed Description 108
2.50 PhysicsActor Class Reference 108
2.50.1 Detailed Description 110
2.50.2 Member Enumeration Documentation 110
2.50.3 Constructor & Destructor Documentation 110
2.50.4 Member Function Documentation 110
2.51 Preferences Class Reference 115
2.51.1 Detailed Description 115
2.51.2 Member Function Documentation 115
2.52 Ray2 Struct Reference 119
2.52.1 Detailed Description 119
2.53 RecoverMoveState Class Reference 119

2.53.1 Detailed Description 119
2.54 Renderable Class Reference 120
2.54.1 Detailed Description 120
2.54.2 Constructor & Destructor Documentation 120
2.54.3 Member Function Documentation 121
2.55 RenderableIterator Class Reference 122
2.55.1 Detailed Description 122
2.56 SearchInterface Struct Reference 122
2.56.1 Detailed Description 123
2.57 Sentient Class Reference 123
2.57.1 Detailed Description 123
2.57.2 Member Function Documentation 124
2.58 SoundDevice Class Reference 124
2.58.1 Detailed Description 125
2.58.2 Member Function Documentation 125
2.59 SpatialGraph Class Reference 128
2.59.1 Detailed Description 128
2.60 SpatialGraphKDNode Class Reference 128
2.60.1 Detailed Description 128
2.61 SpatialGraphManager Class Reference 129
2.61.1 Detailed Description 130
2.62 StartMoveState Class Reference 130
2.62.1 Detailed Description 130
2.63 StartRecoverMoveState Class Reference 130
2.63.1 Detailed Description 131
2.64 Switchboard Class Reference 131
2.64.1 Detailed Description 131
2.64.2 Member Function Documentation 131
2.65 SystemLog Class Reference 134
2.65.1 Detailed Description 134
2.65.2 Member Function Documentation 134
2.66 TagCollection Class Reference 134
2.66.1 Detailed Description 135
2.66.2 Member Function Documentation 135
2.67 TestConsole Class Reference 136
2.67.1 Detailed Description 136
2.67.2 Constructor & Destructor Documentation 137
2.67.3 Member Function Documentation 137
2.68 TextActor Class Reference 137
2.68.1 Detailed Description 138

2.68.2 Constructor & Destructor Documentation . . . 138
2.68.3 Member Function Documentation . . . 138
2.69 TextureCacheEntry Struct Reference . . . 141
2.69.1 Detailed Description . . . 142
2.70 TGenericCallback< ClassInstance, ParamType > Class Template Reference . . . 142
2.70.1 Detailed Description . . . 142
2.71 TimerAIEvent Class Reference . . . 142
2.71.1 Detailed Description . . . 143
2.72 Touch Struct Reference . . . 143
2.72.1 Detailed Description . . . 143
2.73 TouchListener Class Reference . . . 143
2.73.1 Detailed Description . . . 144
2.73.2 Constructor & Destructor Documentation . . . 144
2.73.3 Member Function Documentation . . . 144
2.74 Traversal Class Reference . . . 145
2.74.1 Detailed Description . . . 145
2.75 TraversalAIEvent Class Reference . . . 145
2.75.1 Detailed Description . . . 146
2.76 Tuning Class Reference . . . 146
2.76.1 Detailed Description . . . 146
2.76.2 Member Function Documentation . . . 146
2.77 TypedMessage< T > Class Template Reference . . . 149
2.77.1 Detailed Description . . . 150
2.77.2 Constructor & Destructor Documentation . . . 150
2.77.3 Member Function Documentation . . . 150
2.78 UserInterface Class Reference . . . 150
2.78.1 Detailed Description . . . 151
2.78.2 Member Function Documentation . . . 151
2.79 ValidateMoveState Class Reference . . . 154
2.79.1 Detailed Description . . . 154
2.80 Vec2i Struct Reference . . . 154
2.80.1 Detailed Description . . . 154
2.81 Vec2ui Struct Reference . . . 155
2.81.1 Detailed Description . . . 155
2.82 Vec3i Struct Reference . . . 155
2.82.1 Detailed Description . . . 155
2.83 Vec3ui Struct Reference . . . 156
2.83.1 Detailed Description . . . 156
2.84 Vector2 Struct Reference . . . 156
2.84.1 Detailed Description . . . 157

2.84.2 Constructor & Destructor Documentation 157
2.84.3 Member Function Documentation 158
2.84.4 Member Data Documentation 161
2.85 Vector3 Struct Reference 162
2.85.1 Detailed Description 163
2.85.2 Constructor & Destructor Documentation 163
2.85.3 Member Function Documentation 164
2.85.4 Member Data Documentation 167
2.86 World Class Reference 168
2.86.1 Detailed Description 169
2.86.2 Member Function Documentation 170
2.87 XboxButtonBindRecord Struct Reference 179
2.87.1 Detailed Description 179

3 File Documentation 179

3.1 Actors/Actor.h File Reference 179
3.1.1 Enumeration Type Documentation 180
3.2 Actors/ParticleActor.h File Reference 180
3.3 Actors/TextActor.h File Reference 180
3.3.1 Enumeration Type Documentation 180
3.4 Infrastructure/Common.h File Reference 181
3.4.1 Detailed Description 181
3.5 Infrastructure/TextRendering.h File Reference 181
3.5.1 Detailed Description 181
3.5.2 Function Documentation 181
3.6 Infrastructure/Textures.h File Reference 183
3.6.1 Detailed Description 184
3.6.2 Function Documentation 184
3.7 Input/Input.h File Reference 186
3.7.1 Detailed Description 186
3.8 Util/DrawUtil.h File Reference 186
3.8.1 Detailed Description 186
3.8.2 Function Documentation 186
3.9 Util/FileUtil.h File Reference 187
3.9.1 Detailed Description 187
3.9.2 Function Documentation 188
3.10 Util/StringUtil.h File Reference 190
3.10.1 Detailed Description 190
3.10.2 Function Documentation 190

Index **194**

1 The Angel Engine

1.1 ReadMe

Make sure to first read the overview information at http://angel2d.com – it contains a good rundown of what Angel is and, more importantly, what it is not. Calibrating your expectations can save a lot of heartache later. :-)

1.2 Introduction

This documentation is meant as an introduction to Angel, but also serves as a small intro into basic game architecture. If you're a beginner, don't be off-put by the advanced stuff. If you're experienced, don't be patronized by the beginner stuff. This documentation is probably best skimmed or searched for data rather than read straight through, but hey, it's a free country.

This file really only documents the engine itself – for information on how to get started building, the history of Angel, support, etc., please visit the website. Note that this documentation is focused on the desktop builds of Angel. The differences that exist in the iOS build are documented in their own section.

Whether you're a beginner or advanced, please drop a line if you think something in here is wrong or could be explained better. Always looking to improve.

1.3 A Brief Tour of the Code

The root directory has just one other directory: `Code`. You may ask why we bother having this single-nested hierarchy. The idea is that when you're working on a game, you may wish to have your raw assets in version control as well, instead of just the final exported files you place in your game's `Resources` directory. So at this root level you can add another directory called `Assets` to keep track of those, if you wish.

If you open the Code directory, you should see the following files and subdirectories [not an exhaustive list, these are just the ones that are interesting or may warrant explanation]:

- **GameJam.sln:** The Visual Studio 2008 solution file (Windows)
- **GameJam-Mac.xcodeproj**: The Xcode 3 build and project information (Mac OS X)
- **GameJam-iOS.xcodeproj**: The Xcode 3 build and project information (iOS)
- **README.Linux:** Information about how to get Angel up and running on Linux, along with data about what should and shouldn't work at present.
- **Angel:** where the core prototyping framework gets built. Doesn't do anything by itself, but requires client code to load it up and get things going.
  - *Angel.h:* the master header file that includes all other header files you should need for your game. The included starter projects have this set up as part of their pre-compiled header.
  - *AngelConfig.h:* A set of global defines that are used to disable large scale libraries like FMOD and DevIL.
  - *Libraries:* all the third-party libraries that Angel works to do its magic.
    They all include their distributed documentation and appropriate website links.
    * Box2D: Physics
      (http://box2d.org)
    * DevIL: Image loading
      (http://openil.sourceforge.net)
    * FMOD: Sound
      (http://www.fmod.org)

        * FreeType: Font rasterization
          (http://www.freetype.org)
        * FTGL: Makes it easy to use FreeType in OpenGL
          (http://homepages.paradise.net.nz/henryj/code/#FTGL)
        * GLFW: A cross-platform framework to handle basic windowing and input events
          (http://www.glfw.org/)
        * gwen: A simple GUI toolkit designed for game engines
          (http://gwen.googlecode.com)
        * HID Utilities: USB Controller support for Mac OS X
          (http://developer.apple.com/samplecode/HID_Utilities_Source/index.html)
        * LibOgg and LibVorbis: Open source alternative to FMOD
          (http://www.vorbis.com)
        * LibPNG: Library for loading PNG files (when DevIL is disabled)
          (http://www.libpng.org/pub/png/libpng.html)
        * Lua: Scripting
          (http://www.lua.org)

- **Tools:** a collection of scripts and tools that are used as part of the build process or may be useful to developers.
    - *BuildScripts:* the Lua files which get called during the build process
        * `angel_build.lua`: utility functions and classes for the build/publish
        * `publish.lua`: On Windows, this gets called whenever you make a release build.
          It produces a new directory called `Published` within your project directory – the `Published` directory should contain everything your game needs to run on any (Windows) system and is suitable for zipping up and sharing.
        * `publish_mac.lua`: The publish script for the Mac build. It produces a new directory in your directory. The application there is renamed to what you've defined in `build.lua`.
        * `swig_wrap.lua`: part of the build process to determine if the SWIG scripting bridge needs to be regenerated. It can be time consuming to regenerate and recompile, so it's worth checking first. (See SWIG, below.)
    - *Mac360:* The binary and source versions of a Mac OS X kernel extension that makes it possible for Angel to interface with the Xbox 360 controller. You may have trouble compiling if this isn't installed. (A pain, but thems the breaks.)
    - *swigwin:* The Windows distribution of SWIG – a tool for generating script interfaces to C and C++ code. The `swig_wrap.lua` build script calls this executable (on Windows) and processes `Angel/Scripting/Interfaces/angel.i` to generate all the script hooks into the engine. For the most part, this should just work, but please see the SWIG documentation if you're curious as to how. (Warning: SWIG can be a deep rabbit hole, so don't go diving into it unless you already know what what you're doing or are not using Angel in a GameJam context.)
- **IntroGame:** A modular example project that shows off a lot of the features of Angel using thoroughly commented code. We recommend you start here and explore the indicated files when you see a feature you like. Its directory structure is very similar to that of ClientGame, described in detail, below.
- **ClientGame:** Where you should build your actual game.
    - *GameInfo.txt:* a ReadMe file that will be included in your final `Published` directory by the publish build script (see above). Make sure to edit this file appropriately, or you'll have the sample text as the first thing people see about your game! (Because everybody reads the ReadMe, of course...)
    - *Attributions.txt:* Angel uses a lot of open source components, and the license requirements typically require a shoutout at some point in your documentation. This file will also get included in your `Published` directory, and already contains the acknowledgements for all of Angel's components. Make sure, if you bring in any additional open source components, that you add the appropriate license information. This is also a good place to throw in asset acknowledgements if you didn't create them yourself – though make sure you're ok copyright-wise!

- *build.lua:* a set of values used by the build and publish scripts to customize your game. At the moment, it only contains a value for giving your game a name – we strongly recommend naming your game here as opposed to renaming the project itself so merging will be easier later.
- *Config:* configuration files for your game – this entire directory gets copied to your distribution, so don't leave anything in here you don't want to become public somehow.
  - `input_bindings.ini`: maps keyboard and controller button presses to Messages that will be sent in the game (see the Messages section)
  - `ActorDef`: definition files for your actor archetypes (see the Archetypes section)
  - `Level`: definitions files for your levels (see the Levels section)
- *Logs:* Where any logs you generate in your game will be stored for later viewing. (Assuming you use the Angel built-in logging. Obviously you could write your own files wherever the heck you want.) Gets copied to the Published directory.
- *Resources:* All non-code bits of your game should go in here. This will also be copied to the Published directory.
  - `Fonts`: hopefully this is self-explanatory
  - `Images`: hopefully this is self-explanatory
  - `Sounds`: hopefully this is self-explanatory
  - `Scripts`: Any Lua that you want to use in your game should get put in here. The included `client_start.lua` gets imported directly into the console namespace, so any functions you define here will be available in the console. If you want to do any game setup from Lua, `client_start.lua` is where it should go. As you build you'll notice other files get copied to this directory – the scripts that help make Angel run. These files will be frequently updated from the `Angel/Scripts/EngineScripts` directory every time you build, so making changes to them here will leave you disappointed when they get overwritten.

1.4 The World

Angel's entire simulation is based around the World class. It represents, unsurprisingly, the world of your game. Nothing will take place in your game without adding things to the world.

Like many other classes in Angel, World uses the singleton pattern. What this means is that there can only ever be one instance of this class in the program – if you think about it, this makes a lot of sense. So you'll never actually create the world directly. Instead, you call the static function World::GetInstance(), which returns a reference to the single world that has been created for your game. (Instead of using World::GetInstance, you'll find that we use a defined shortcut called theWorld. Since getting the world instance is such a common operation, this shortcut is handy.) For more information on the singleton pattern, this paper from Microsoft is a good starting point:

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

To do the initial setup of the world, we call the World::Initialize function.

```
theWorld.Initialize(1024, 768, "My Game", false);
```

This call opens a window to view the world – it's 1024x768 with "My Game" in the title bar and anti-aliasing disabled. Obviously you can swap out your own values and change the properties of the window.

After you call World::Initialize, you'll want to do all your setup (adding Actors to the world), but nothing will actually happen until you call World::StartGame(). This actually begins the game loop, and the function will never return until you close down the application by calling World::StopGame().

The game loop is an important concept. There are two parts to it (if you look at the World::TickAndRender function you'll see there's actually a bit more, but conceptually, two parts): Update and Render. Every object that you add to the world implements these two functions and will have them called once every time we go through the game loop. The Update parts just deal with game logic – handling collisions, answering messages, updating positions, changing colors, incrementing scores, etc. The Render parts are strictly for drawing the individual element to the screen. We separate the two because what happens in the Update part of the loop can affect how things should be drawn in the Render part.

You can also pause and unpause your game by calling World::StopSimulation or World::StartSimulation. While the sim is paused, certain things will still go on (like sound and input), but normal objects won't get their update calls. (Note that they'll still be drawn!)

The world has a good deal of other functionality, but it's mostly related to other parts of the game and so it'll be covered in other sections.

1.4.1 Logging

Logging is one of those things (like a lot of Angel's functionality) that isn't really hard to get working, but is kind of annoying and not the fun part of game development. So we provide some simple classes to make logging easier, and if you need something more advanced, you can always just hack it in yourself. :-)

There are a few types of logs. All of them are derived classes of DeveloperLog, and implement DeveloperLog::Log and DeveloperLog::Printf. When you call DeveloperLog::Log, you just pass it a string and it gets written in the log of your choice. When you call DeveloperLog::Printf, you can pass it a format string and list of parameters, just like you would do with the `normal printf function`. For all cases where you write to a log, the system will automatically put a newline character at the end.

The first type of log we care about is a SystemLog. This log just spews right to the OS's console. You may ask why not use the normal `std::cout` or `printf` functions – by using a SystemLog, you integrate with the Angel logging system, which lets you easily mirror output to a file, for instance. Also, when debugging in Visual Studio, the SystemLog prints to the Output pane instead of the system console, which makes it a lot easier to review messages.

Then there's a ConsoleLog, which writes its output to the in-game Console (invoked with '∼'; see below). This can be handy for getting debug values without having to even leave your game.

We also have FileLog, which writes its output to a file of your choosing. The class provides a static function FileLog::MakeLogFileName, which accepts a string and returns another string representing the file path where that log should reside. On Windows, this is in the Logs directory right next to your executable. On the Mac, though, the logs will go in `∼/Library/Application Support/Angel Games/Logs`, in order to be a good OS X citizen.

Finally, we have a CompoundLog, which lets you combine several different logs together and write to them all at once. The best use case for this is the static "sysLog," which Angel uses internally to write all its messages and errors. By default, it just contains a SystemLog, but you can add a FileLog to it if you want to record the Angel messages in a more persistent fashion.

That's pretty much it for logging – the system is simple, but really flexible. You could create a separate log for warnings or errors, put some into one file, some into another, some only to the console, etc.

```
sysLog.Log("This message will go to the system console.");
ConsoleLog *c = new ConsoleLog();
c->Printf("This will print pi to the in-game console: %1.4f. ",
      MathUtil::Pi);
FileLog *f = new FileLog(FileLog::MakeLogFileName("warn_log"));
f->Log("This will write to a file in the Logs directory called \"warn_log.log\".");
sysLog.AddLog(f);
sysLog.Log("This will log to both the system console and the log file we created.");
```

1.4.2 Preferences

Angel contains a simple mechanism for storing persistent values so your players can store data across sessions. It's only good for simple sets of values (floats, ints, strings), but you can do quite a bit with just those.

To set a preference, we call the appropriate function for the desired type.

```
thePrefs.SetInt("GameStartupSettings", "NumberOfEnemies", 15);
thePrefs.SetFloat("GameStartupSettings", "JumpHeight", 3.14159);
thePrefs.SetString("GameStartupSettings", "HeroName", "SuperDude");
```

Note that just setting the preference does not make it persist. If you want to hang on to the current values of the preference system, call `thePrefs.SavePreferences()` at some point. This will write out a file that the game will look for at next startup.

There's a file in the Config directory of your game called `defaults.ini`. Any values you put in here will automatically be loaded into the Preferences structure, where tables will become categories. The values will be overridden if you have called `thePrefs.SavePreferences()` to store them off.

1.4.3 Text Rendering

There are two steps you have to take before drawing text to the screen. First off, you have to register the font with our text system. You can do this by calling RegisterFont:

```
RegisterFont("Resources/Fonts/Helvetica.ttf", 12, "Helvetica12");
```

This example will register the font file at the path we referenced (in our `Resources/Fonts` directory), at a size of 12 points, with the nickname "Helvetica12" that we will use to reference this font when we actually want to draw text.

By default Angel includes a mono-spaced font called `Inconsolata`, which it registers at 24 points as "Console" and 18 points as "ConsoleSmall". You can use this for your own text if you like, or use any font which is readable by FreeType. Be careful when distributing your game, though, since most fonts are not free to distribute. Here are some resources for fonts which may be distributed (but check their licenses):

- OFL Licensed Fonts: `http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item-_id=OFL_fonts`
- Bitstream Vera Fonts: `http://www.gnome.org/fonts/`
- Open Font Library: `http://openfontlibrary.org/`
- Google Web Fonts: `http://www.google.com/webfonts`

When you actually want to render some text on the screen, you simply call:

```
DrawGameText("Draw Me", "Helvetica12", 50, 50, 45.0f);
```

This will render the text "Draw Me", in the font referenced by "Helvetica12", starting at 50 pixels from the top of the window and 50 pixels from the left of the window. It will be rendered at a 45 degree angle. Text will also accept any color values that you set using a glColor$*$ call, including transparency.

Of particular note is that the text rendering system operates using pixel coordinates – this is different from the rest of Angel, which uses the coordinates of the world for placing items.

Most of the time, though, you won't need to manipulate the text system directly. It's typically easier to use a TextActor, which we'll cover in a bit.

1.5 Actors

Actors are the central unit of simulation in Angel. Anything you want to place in the world – your hero character, enemies, the ground, the walls, a spaceship – is likely going to be an Actor.

1.5.1 Appearance

The base Actor provides all the functionality you need to draw something on the screen. It has a position, a size, a color, a texture (or series of textures), and a shape.

```
Actor *a = new Actor();
a->SetPosition(-3.0f, 4.0f);
a->SetColor(1.0f, 0.0f, 0.0f);
a->SetDrawShape(ADS_Circle);
a->SetSize(3.0f);
```

This code will create an Actor who lives at (-3, 4) on the XY coordinate plane, is pure red, and drawn as a circle with a radius of 3.

NB: The Actor won't show up in your world until you add it. Don't forget to call:

```
theWorld.Add(a);
```

This inserts your Actor into the world so it will start receiving Update and Render calls when the game loop runs. You can also pass a second parameter to the World::Add function that indicates the layer you want to put this Actor on. This only affects the order in which an Actor is rendered – Actors on lower layers will always appear behind Actors on higher ones. Within a layer, Actors are drawn in the order they were added. The layer numbers are arbi[illegible], so use whatever system makes sense to you.

You can also assign a texture to an Actor. If we wanted the previous actor to have a texture, we would have called:

```
a->SetDrawShape(ADS_Square);
a->SetSprite("Resources/Images/superdude.png");
```

Images can be in any format supported by `DevIL`, and any transparency on the image will show up in Angel. (Note that if you disable DevIL in AngelConfig.h, only PNG images will be supported.)

You can also assign multiple sprites to an Actor, either for use in an animation or for switching between various visual states. To load up a series of images, call:

```
a->LoadSpriteFrames("Resources/Images/superdude_001.png");
```

This will load up the `superdude_001.png` file, and all other files that start with "superdude_" and end in a number. Internally we're limited to 64 frames, but if you need to get more, simply change the MAX_SPRITE_FRAMES value in Actor.h.

Once you've loaded up your sprites, you can swap between them manually:

```
a->SetSpriteFrame(7);
```

Or play a series of them:

```
a->PlaySpriteAnimation(0.1f, SAT_OneShot, 0, 10, "JumpingAnim");
```

That call will play frames 0-10, waiting 0.1 seconds between each frame, and ending when it finishes (other options are SAT_PingPong and SAT_Loop). The final parameter, "JumpingAnim," is an optional name assigned to this animation. When it finishes, the animation system will call Actor::AnimCallback on this Actor, passing in that name. If you want to receive notifications about animations finishing, you can implement this in a subclass.

1.5.2 Names and Tags

Once you've added a bunch of Actors to your world, it can become difficult to manage them. So we've added the ability to add metadata to Actors to make it easier to work with individuals and groups.

The first bit of metadata is the Actor's name. Every Actor in the world has a name that is guaranteed to be unique. By default this name is simply "Actor" with numbers appended to make it unique, but you can assign whatever names you want to your actors.

```
a->SetName("hank");
b->SetName("hank");
theWorld.Add(a);
theWorld.Add(b);
```

Actor names are always upper-cased, so after this code runs, `a` will have the name "Hank." Since Actor names are also guaranteed unique, `b` will have the name "Hank1." (The Actor::SetName function returns the **actual** name that was assigned to the Actor, so you know what it is.)

NB: Names are only guaranteed unique when the Actor has been added to the World. Until then, if you haven't assigned a name, the Actor::GetName function will return an empty string.

Later, if you want to access one of these objects, you can call:

```
Actor *retrieved = Actor::GetNamed("Hank");
```

The Actor::GetNamed function will return `NULL` if there's no Actor with the given name, so be sure to check your pointers before dereferencing.

Oftentimes, though, you want to keep track of a group of Actors with similar properties. That's where tags come in. You can throw a set of freeform string data on an Actor and retrieve groups that match specific sets. For example:

```
//a, b, and c are Actors
a->Tag("blue, person");
b->Tag("blue, person, tall");
c->Tag("green, person");

ActorSet people = theTagList.GetObjectsTagged("person");
ActorSet bluePeople = theTagList.GetObjectsTagged("person, blue");
ActorSet tallPeople = theTagList.GetObjectsTagged("tall, person");
```

After calling this, `people` will be an ActorSet containing `a`, `b`, and `c`. `bluePeople` will just contain `a` and `b`, while `tallPeople` will only contain `b`. If you decide that `c` is now sub-human, you can call:

```
c->Untag("person");
```

Tags and names give you a lot of flexibility in how you manage groups of Actors. Depending on your specific need, it can be a lot easier to just use tags and names instead of managing your own pointers and groups of Actors.

1.5.3 Intervals

Usually, when you change an Actor's property, it immediately takes effect. Setting a new position teleports the Actor there, changing a size happens immediately, etc. A lot of times, though, you want an Actor to visibly transition to a new state. You could set up a timer and do all your updates in the Actor::Update function to accomplish this, but if your transition is straightforward, we've provided some utility functions to make it simpler.

```
a->MoveTo(Vector2(-5.0f, 0.0f), 3.0f, true, "MovementFinished");
a->RotateTo(0.0f, 3.0f, true);
a->ChangeColorTo(Color(0.0f, 0.0f, 1.0f), 3.0f, true);
a->ChangeSizeTo(3.0f, 3.0f, true);
```

The Actor `a` will now begin moving from its current position to (-5, 0), while rotating to 0 degrees, turning blue, and changing its size to 3. All of this will happen over the course of the 3 seconds (the second parameter of each interval function is the duration).

The third parameter indicates whether the Actor should move linearly or use the MathUtil::SmoothStep function. If the Actor is transitioning smoothly, you'll see it ease out of the old state and into the new state; otherwise it will follow a straight trajectory. (See MathUtil::SmoothStep for more information.)

The final optional parameter to an interval function is a Message name that will get broadcast when the transition is done. Because all the changes in our example will all finish at the same time, we only send one Message. More on messaging in a bit; for now just know that it's a way for your Actors to get notifications of specific events.

1.5.4 Subclasses

The base Actor class contains a lot of functionality, but oftentimes you want to do more or different things with it. We expect that most developers will subclass Actor to add their own functionality to it. We also provide some useful subclasses, described below.

1.5.5 Camera

A fairly unique subclass of Actor, the Camera is what controls how you view the world. By default, it's set up at (0, 0, 10), and looks at the origin. Note that the Camera is the only Actor in Angel which accepts three-dimensional coordinates. You can zoom in and out by changing its Z value. Like the world, it uses the singleton pattern; its instance is accessed via `theCamera`. See the Camera class documentation for more information.

1.5.6 GridActors

By default, a GridActor is the first thing you see when you start up Angel. Not too surprisingly, it draws a grid of lines to the screen. Our default grid matches the OpenGL coordinate system, so it's useful for placing objects in your world or setting up your camera positioning. You can change the color of the lines, spacing, draw extents, etc. of the grid. See the GridActor class documentation for more information.

1.5.7 FullScreenActors

A FullScreenActor is useful for splash screens or backdrops. No matter where you move the Camera, this Actor is guaranteed to take up the entire window. If there's a texture assigned, it will be stretched to fit the current window parameters. See the FullScreenActor class documentation for more information.

1.5.8 HUDActors

HUD Actors are used when you want to create some element of user interface that should appear on the screen at the same position regardless of Camera movements. Essentially, all the HUDActor's functions regarding things like size and position operate in pixel units and screenspace. Otherwise, they are normal Actors, and you can do animations, transitions, etc. with them. See the HUDActor class documentation for more information.

1.5.9 ParticleActors

A game isn't a game until it has particle systems! If you're unfamiliar with what particles are in the context of a game, `this article by Jeff Lander` is a good introduction. Don't worry about the implementation, though, since we've done that for you. :-)

The ParticleActor has a lot of properties to affect its appearance, but don't be scared. The important functions to look at are things like ParticleActor::SetParticleLifetime, which says how long each individual particle can stay on the screen, ParticleActor::SetEndColor, which sets the color that the particle will transition to over its lifetime (use one with an alpha of 0 to have the particles fade out).

You can set textures and animations on particles like you would with any other Actor – animated particles are good for making things like smoke and fire.

See the ParticleActor class documentation for more information.

1.5.10 TextActors

The raw text rendering system described earlier is pretty limited. You have to draw the text every frame, it doesn't handle newlines, it works in screen coordinates, you can't align the text, etc. The TextActor class mitigates these pains by bundling up a bunch of convenient text functionality into an Actor which draws itself at a position in the World (as opposed to on the screen).

```
TextActor *t = new TextActor("Console", "Here I am \nRock you like a hurricane",
      TXT_Center);
theWorld.Add(t);
```

This code will create a TextActor that is center-aligned, uses the default console font, and prints the lyrics to a totally sweet rock song. The newline character gets interpreted properly, so the second line of the song will be below the first one.

See the TextActor class documentation for more information.

1.6 Messaging

Angel supports sending messages through a centralized Switchboard class. Any class which implements the MessageListener interface can send or subscribe to Messages.

Each Message has a name that it's given, which should tell what event it's signaling or what type of Message it is. MessageListeners who wish to subscribe to that Message pass it to the Switchboard to receive newly broadcast Messages every frame.

```
theSwitchboard.SubscribeTo(a, "MeteorHit");
theSwitchboard.UnsubscribeFrom(a, "GameStarted");
theSwitchboard.Broadcast(new Message("ReadyToGo"));
TypedMessage<Vector2> loc = new TypedMessage("StartingSpot", a->
      GetPosition(), a);
theSwitchboard.Broadcast(loc);
```

This code does a few things. First of all, it subscribes `a` to any Messages named "MeteorHit." So when someone broadcasts a MeteorHit message, the MessageListener::ReceiveMessage function will be called on `a`. Then it unsubscribes `a` from the "GameStarted" Message, so it won't receive notifications anymore. Finally, it creates a new TypedMessage that carries a bit of position data, and broadcasts it to anyone who cares about Messages with the name "StartingSpot." (TypedMessages are a templated class, so you can put anything you need to broadcast into them.)

NB: When you pass a Message pointer to the Switchboard::Broadcast function, that's it, you're done with it. The Switchboard will delete the memory once it's delivered the Message to all subscribers, so don't expect to do anything with it afterwards. It's not a good idea to hang on to Message pointers for this reason.

Controller and keyboard input is performed using Messages – by making changes to the `input_bindings.-ini` file in your directory, you designate what Messages get sent when keys or buttons are pressed. (See the `input_bindings.ini` file included with IntroGame for an example.)

By using Messages to signal game events, you make it easier to extend your game – anybody can listen for any message, so you don't have to keep making changes to the class sending the signals to have new objects respond to it. You can also kind of "sketch out" your game flow in Message form before implementing game logic, or experiment with an individual Actor by sending it the Messages it will receive when it's a part of the full game.

1.7 Physics

Angel supports simple rigid body physics by incorporating the `Box2D` library. Box2D has **a lot** of functionality, so check out the documentation available on their website and wiki for details.

If all you want to do is play with some simple physics, though, we've wrapped up a lot of Box2D into a very *angelic* interface. :-)

The first thing you need to do if you want to use physics is call World::SetupPhysics. By default, this function sets up physics with a standard gravity pointing down the Y axis and extending from (-100, -100) to (100, 100). (Outside of that region, physics don't exist. It is the netherrealm.) You can override these defaults if you want a different (or no) gravity, or a differently sized simulation space.

Afterwords, you can create a PhysicsActor and put them in the world.

```
p1 = new PhysicsActor();
p1->SetDensity(0.8f);
p1->SetFriction(0.5f);
p1->SetRestitution(0.7f);
p1->SetShapeType(PhysicsActor::SHAPETYPE_BOX);
p1->InitPhysics();
```

This creates a new PhysicsActor, sets up its physical properties, and kicks it into motion. If you're not familiar with the physics terms, see the individual function documentation for information on the effects they have on your object. Note that the PhysicsActor will not start being physically simulated until you call PhysicsActor::InitPhysics. That function also locks in the physical properties you set, and they can no longer be changed. (Also things like position and rotation, since you've now turned those over to Sir Isaac Newton.)

1.7.1 Side-Blockers

Oftentimes you want to keep your PhysicsActors from flying off the edge of the screen. We've added a function to the World that makes it easy to set up blockers at the edge of the screen.

```
theWorld.SetSideBlockers(true, 0.7f);
```

That code will turn the blockers on and make them fairly bouncy (restitution = 0.7). You can turn them off or change their bounciness at any time. Note that the side blockers move when the Camera moves, but otherwise are static in the world.

1.8 Input

A game isn't much without being able to accept input from the player. We've provided three different ways to get that input, which should cover a pretty good variety of use cases.

1.8.1 Mouse

There are two things that you care about with regard to the mouse: when the player moves it, and when they click it. To get information on either of these events, have a class implement the MouseListener interface. Any instance of MouseListener will get MouseListener::MouseMotionEvent, MouseListener::MouseDownEvent, and MouseListener::MouseUpEvent called when these happenings take place. All three functions pass in the current mouse coordinates (in screen space), and the button functions pass in an enum to tell you which button was affected (if you care about left-click vs. right-click, for instance).

It's a pretty bare-bones mouse notification system, but it should be enough to build whatever mouse input functionality you need.

1.8.2 Keyboard

The keyboard contains many more buttons than the mouse (at least on most machines). Keyboard input is handled through the Messaging system and the `input_bindings.ini` file located in the Config directory. Using that .ini file, you can map keypresses to Messages that will be sent at the appropriate time. For instance:

```
SPACE= Jump
G= ToggleGrid
A= +SomeonePressedA
A= -SomeoneReleasedA
```

Maps the space bar to a "Jump" Message and the *G* button to a "ToggleGrid" Message. The *A* button mappings here are a bit more interesting. By prepending a Message with +, you indicate that Message should only be sent when the key is pressed down. If you prepend it with a - (minus sign), the Message will be sent when the key is released. If you have no symbol in front of a Message, the assumption is that you want it sent when the key is pressed.

Once you have your input bindings set up, you just need to set up objects in your world to listen for those Messages and respond appropriately.

For a more immediate check without having to set up Messages and bindings, you can call InputManager::IsKeyDown, passing it either a plain `char` or any of the defined values from `GL/glfw.h`.

```
bool d = theInput.IsKeyDown('d'); // Will return true if the D button is pressed
```

1.8.3 Controller

Angel supports getting input from an Xbox 360 controller connected over USB. On Windows, developers will need the DirectX SDK and players will need the DirectX runtime. Both are freely available from Microsoft. On the Mac, developers and players both need to install a kernel extension that exposes the controller as a HID device. It's available in our Tools directory, or from `http://tattiebogle.net/index.php/ProjectRoot/Xbox360Controller/OsxDriver`

To get input from the controller, you have a few options. For button presses, you can use the input_bindings.ini file to map things like `P1BUTTON_A` for the *A* button on controller one. You can also directly query the controller during gameplay, which gives you access to the analog sticks and triggers. You can also use the vibration function of the controller (currently only on Windows).

See the Controller class documentation for more information.

1.8.4 MultiTouch

When building for iOS devices, Angel will wrap the unique hardware input for these devices into a more angelic, C++ interface so you don't have to learn Objective-C and/or scatter your code with strange references to a foreign SDK.

You can have any of your classes inherit from TouchListener to get notifications when each touch starts, moves, or ends. (The interface is very similar to the MouseListener setup.)

In addition, you can query to get a list of all touches the devices is currently registering with the static function TouchListener::GetTouchList, which returns a list of pointers to Touch structures.

Finally, there's the Accelerometer class, which just provides access to the current tilt data as a Vector3. Note that it does some simple buffering and smoothing of the values to prevent jitter – if you want to tweak the lag/smoothness tradeoff, edit the value for ANGEL_ACCEL_BUFFER_SIZE, found in AngelAppDelegate.m

1.8.5 MobileSimulator

Angel builds and runs on iOS devices and the simulators that Apple provides with their SDK. Maybe not everyone on your team has a Mac, though, or you just want to play with an idea and don't want to bother setting up the extra machine. The MobileSimulator class will help you out here. When you add one of these objects for the world, it *pretends* to be the hardware of an iPhone, iPod Touch, or iPad. Mouse clicks are treated as touches, holding down the Ctrl key will let you play with multi-touch gestures, and the Xbox 360 controller's thumbsticks are read as the accelerometer input.

The important thing here is that this simulator fills all the same data structures that the real hardware would, so as long as your code is working through the provided TouchListener and Accelerometer interfaces, it will work the same as it would on the device itself.

Obviously you would eventually want to build with the real Apple SDK, but this can get you a lot of the way there. Also, since the iOS build doesn't support the Console, using the MobileSimulator can be useful to let you run the same code, but have more control over your world while you're prototyping.

1.9 Sound

The sound system in Angel is a wrapper around `FMOD`. It can play most sound formats, including WAV, MP3, and Ogg Vorbis. We recommend Ogg for your sound distribution needs because of the following properties:

- It's free to distribute Ogg files, unlike MP3s, which require a license. (We're just talking about the file formats themselves here; copyrighted content is still legally thorny to distribute.)
- They have good compression to save you space (unlike WAVs)
- They have more precise timing than MP3s, which can be important if you're trying to sync up events in your game with a music track.

If you don't currently have a sound program capable of dealing with Ogg, there's `Audacity`, which is free, and `Reaper`, which has a free trial period.

There are two steps to playing sound in your game.

```
SAMPLE_HANDLE mySound = theSound.LoadSample("Resources/Sounds/bonk.ogg", false);
SOUND_HANDLE soundPlaying = theSound.PlaySample(mySound);
```

The first call loads up a sound from disk and prepares it for playback. It returns a sound handle that you'll want to hold on to so you can use it for triggering the sound later. The `bool` parameter says whether or not the sound should stream or get loaded all at once (longer sounds should be streamed).

After a sound has been loaded, you play it with the second call, passing the handle you got from the first. There are additional parameters you can set when calling SoundDevice::PlaySample, to set the volume, the stereo positioning, and whether it should loop. This function returns a SOUND_HANDLE that you can use to manipulate the sound **during** its playback.

See the SoundDevice class documentation for more information.

FMOD is the default sound system for Angel, but it is **not** free to distribute if you are charging for your game. Their `rates` are very reasonable, but if you would prefer a free alternative, you can disable FMOD in AngelConfig.h. This will cause Angel to fall back to OpenAL support only.

NB: OpenAL, as implemented in Angel, does not produce sound at the same quality level as FMOD. In particular, you're likely to hear skips/pops at the boundaries of looping audio. In addition, we only support Ogg Vorbis playback if you've chosen to use OpenAL. Them's the breaks.

1.10 AI

1.10.1 Pathfinding

One of the trickier problems of low-level AI is getting characters to move around the environment without running into things. Real games use big expensive solutions like `Kynapse`, because generating pathfinding data in 3D is hard. Luckily, we're only 2D, so we whipped up a path generation system just for Angel. The paths it produces aren't the prettiest, but it sure beats authoring by hand (and then having to change it if you change the layout of your space).

The path generation tool will generate paths around PhysicsActors located in the world. The best idea is probably to add your static geometry (the things you want to path around), generate your pathfinding data, then add all your other PhysicsActors.

Here's how you actually generate path data:

```
BoundingBox bounds(Vector2(-20, -20), Vector2(20, 20));
theSpatialGraph.CreateGraph(0.75f, bounds);
```

First we create a bounding box to let the spatial graph know what area it needs to explore (the smaller the area, the quicker the paths will generate). The second function takes a radius and the bounding box we made. The radius is how wide the Actor who will be trying to pathfind is. This way the generator knows how small a space is too small for it to fit.

Once you've generated the graph, you're ready to get path information from it.

```
Vector2 start(0, 0);
Vector2 end(5, 5);
VectorList path;
theSpatialGraph.GetPath(start, end, path);
```

SpatialGraph::GetPath will fill up the `path` parameter with a list of points which make up the most efficient path from `start` to `end`. If there is no path available, `path` will be empty.

Once you have this data, your actors can do with it whatever they want. See the IntroGame's Pathfinding screen for an example of how to build an Actor that follows a path to a given point.

(Note that even in a large space, generating the spatial graph is fairly quick. While you definitely shouldn't call it every frame, calling it when something major has changed in your world is not inappropriate.)

1.10.2 Other AI

The good news is that there are other AI functions in Angel, like a simple state machine that can let you give different goals to an Actor and have him switch between them (including automatically traversing a path).

The bad news is that it was written by someone who's no longer affiliated with Angel, and the code is mysterious. Some day it will be investigated, grokked, and documented. But not today. (If somebody would like to take up this task, we wouldn't stop them.)

1.11 Archetypes and Level Files

1.11.1 Archetypes

When you want to iterate on an Actor's properties (getting it to be just the right size or starting position), it can be very annoying to have to keep recompiling your whole game just to check on it. We've provided a way to make actor definitions data-driven so you can just work with a set of text files.

In the `Config` directory of your game, you'll find another directory called `ActorDef`. Any .lua file you throw in here will be assumed to be describing a set of Actor archetypes. The format is fairly straightforward.

```
simple_actor = {
  color = {1, 0, 1},
  position = {-3, -2},
  alpha = 0.5,
  size = 5,
  tag = "simple, small, purple",
  name = "SimpleActor"
}
```

If you're familiar with Lua, you may notice that this is a straight-up table declaration. The name of the table represents the name of the archetype – what you'll use to actually instantiate it.

The properties that follow will directly affect the Actor after it's been created. Any Actor function that takes the form "SetX" can be used here as simply "x," and other functions can simply be lowercased to be invoked (see how Actor::Tag became simply "tag"). Vectors and colors are enclosed in braces. The properties in an Actor defintion file will only work for functions that take one parameter – if you try and call something with multiple parameters, you'll get scripting errors.

Once we've defined `simple_actor` in our file, we can create one in code very simply.

```
Actor *a = Actor::Create("simple_actor");
```

Note that the Actor::Create function doesn't add the Actor to the World, so you'll still want to call World::Add.

1.11.2 Level Files

Once we've made our actor definitions, we may want to iterate on their placement in the world. To do that, we create another .lua file in `Config/Level`.

```
LeftActor = {
  type = "simple_actor",
  size = 3,
  position = {-5, 0},
  tag = "spawned"
}

RightActor = {
  type = "simple_actor",
  size = 5,
  position = {5, 0},
  color = {1, 0, 0},
  alpha = 0.8,
  tag = "spawned"
}
```

Each section in a level file describes an Actor to be placed in the world. The first property in a section is `type`, which indicates which archetype to use for this Actor. All the properties after that will be called on the newly created Actor, **potentially overriding** properties set in the archetype.

To load a level in code:

```
theWorld.LoadLevel("my_level");
```

This example will process `Config/Level/my_level.lua`, create all the Actors from designated archetypes, apply the additional properties, and add them to the world.

1.12 Lua

Angel includes Lua scripting, for those who prefer to work in something other than C++. Internal functions are exposed to the scripting layer using `SWIG`. Within Angal, Lua is intended less as a full runtime (though you can receive messages, create Actors, and call almost all engine functions from Lua), but more as a configuration language for use in ActorDefs and Level files.

In the `Resources/Scripts` directory, you'll find a file called `client_start.lua`. This will get executed at startup, so any Lua setup you want to do can go in here.

Most internal functionality is exposed in Lua. So, for example, you could have this in your `client_start.lua`.

```
a = Actor_Create("simple_actor")
theWorld:Add(a)
```

Yeah, it's not too exciting, but you don't have to recompile the engine when you change it!

If you're not familiar with Lua, Roberto Ierusalimschy's `Programming in Lua` is a good start. (The online edition is for version 5.0, whereas we ship with 5.2. Most functionality is the same, but there are some differences, so if you want to dive further into Lua, be sure to track down a print copy.)

1.12.1 In-Game Console

When you press the "∼" button, the in-game console appears. This is, for most intents and purposes, a functional Lua console. From here you can manipulate your world, create new Actors, modify existing ones, etc.

It even has auto-complete. Pretty snazzy.

1.13 Tuning

Very frequently while developing a game, you'll want the ability to change values at runtime and watch their results. We provide a few methods of doing this; hopefully one of them is a good fit to your workflow.

If you look in your `Config` directory, you'll see a file called `tuning.lua`. You can declare variables in here that will be available for easy tuning when the game runs.

As an example, if you wanted to tune how high a character jumped, you could put this in your tuning file:

```
JumpHeight = {
  type = "float",
  value = 10.0
}
```

Then, in your C++ code, in the function that handles jumping, you get access to it like so:

```
float jumpHeight = theTuning.GetFloat("JumpHeight");
```

Lua usage would looks nearly identical:

```
jumpHeight = theTuning:GetFloat("JumpHeight")
```

Now this doesn't seem like much of a win, since you could have just `#defined` it at the top of your source file. But, if you edit this `tuning.lua` file while the game is running, Angel will detect the changes and alter the variable's value.

If you prefer to stay in one program while tuning, we also provide some easy console handles for playing with variables. Pull up the console with (∼), and run:

```
tune("JumpHeight", 3)
```

And you'll see the effects immediately in-game.

Once you have the values where you want them, you can call `SaveTuningVariables()` from the console and the values you've set will be written back out into the file so you don't have to try and remember them.

1.14 Working on iOS

There are a few differences to be aware of when building a game for iOS, or when porting an existing game to the platform.

- iOS apps are structured a little differently than traditional desktop programs. In particular, the `main()` function is used to set up the OpenGL context and input hooks, so the spot to do any game initialization is now in `iPhoneMain.cpp`.
- You can't set the name of your came in the `build.lua` file anymore, because Xcode needs to know the final program name so it can do code signing. To set your game's name, open the target properties for ClientGame and set the "Product Name" value to whatever you want.
- If you are disabling FMOD, in addition to setting the value in AngelConfig.h, you **also** need to remove the "Other Linker Flags" values in the Target properties. Otherwise the app will try to link against FMOD anyway, and then segfault when you try to load audio.
- In the `ClientGame/ios` folder, in addition to the `iPhoneMain.*` and .plist files, you'll also find a series of images.
  - The ones named as `Icon*`.png will, not surprisingly, get used as icons in various places in iOS. Don't change the names, dimensions, or format of any of these images, but put your own content into each image file. The Xcode target has already been set up to pull these images in as part of the packaging process.
  - The ones called `Default*`.png will be used as splash screens on various devices while the game is loading. Same principles apply – keep the names, dimensions, and PNG format, but put whatever you want into the files.
  - Finally, there's the `iTunesArtwork` file. Though it doesn't have a file extension, it's a PNG file at 512x512 pixels. It's for that glorious day in the far-flung future when your game is making a hojillion dollars on the App Store. This is the image that will be displayed on the store page.

See the MultiTouch section for information on how to access the hardware input of iOS devices.

1.15 Niceties and Handy Doodads

Finally, we provide a number of utility classes and functions to handle common game tasks. In particular, the MathUtil class contains declarations of constants and wrappers for common (annoying) math that game developers find useful. The StringUtil.h file contains a number of simple functions for processing strings. Check them out before re-inventing the wheel.

1.15.1 Publish Script

Modern software distribution can be a pain, especially ensuring that what runs on the developer's machine will run on the player's. Making sure all the resources are included, the right libraries, etc. We've taken some steps to make this easier on you in the form of the publish scripts.

On Windows, the publish script runs whenever you build your program in Release mode. (If you find this to be a buzzkill, you can disable it by going to Project -> Properties -> Configuration Properties -> Build Events -> Post-Build Event and editing the command line there. Make sure you're editing the Release configuration.) After the script runs you'll find a `Published` directory sitting in your `Release` directory. It should contain everything you need to run your game. Let us know if you find that something is missing from the `Published` directory so the publish script can be updated.

On the Mac, the publish script also runs when you build in Release, which you can disable by commenting out the appropriate lines in the "Run Script" phase of your game's target. It produces a `Published` directory in your `build` directory that should also contain everything you need to distribute your game.

Both publish scripts also put the `Attributions.txt` and `GameInfo.txt` alongside the executable for your game. The attributions file is important for licensing purposes (Angel uses **a lot** of open source libraries), and the GameInfo file is good for giving your players whatever additional information you want them to have about your game.

1.16 Final Thoughts

This was an overview of what Angel can do, but there's a lot more to be found by exploring the classes and their interactions. The full API documentation is also here, so poke around if you're curious as to exactly what a certain function or class does.

Or, try it out in your game and see what it does. :-)

Good luck!

1.17 Bug Reports and Feedback

Please use the angel-engine Google Group to discuss any problems, thoughts, ideas for new features, bugs, complaints about the documentation, etc.

`http://groups.google.com/group/angel-engine`

We'd especially like to see any games you make with Angel! For serious.

2 Class Documentation

2.1 Accelerometer Class Reference

A class to read the data from a hardware accelerometer.

`#include <MultiTouch.h>`

Public Member Functions

- const Vector3 GetData ()
- void NewAcceleration (Vector3 data)

Static Public Member Functions

- static Accelerometer & GetInstance ()

2.1.1 Detailed Description

This class facilitates reading data from the hardware accelerometer on iOS devices. Note that it performs some simple averaging of values over time – if you want to access the data more immediately, you can tune the defined values in AngelAppDelegate.m

Definition at line 167 of file MultiTouch.h.

2.1.2 Member Function Documentation

2.1.2.1 Accelerometer & Accelerometer::GetInstance ( ) [static]

Used to access the singleton instance of this class.

Returns

The singleton

Definition at line 107 of file MultiTouch.cpp.

2.1.2.2 const Vector3 Accelerometer::GetData ( )

Read the current value or the detected accelerometer values.

Returns

The X, Y, and Z parameters being detected. Values are in Gs.

Definition at line 116 of file MultiTouch.cpp.

2.1.2.3 void Accelerometer::NewAcceleration (Vector3 *data*)

Push a new accelerometer value into the object. No need to call this directly – it's either called by the iOS framework or the MobileSimulator class.

Parameters

data	A 3D vector representing the current acceleration

Definition at line 134 of file MultiTouch.cpp.

The documentation for this class was generated from the following files:

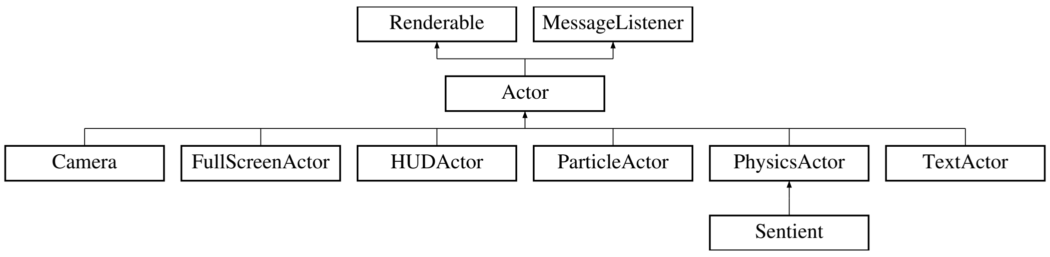
- Input/MultiTouch.h
- Input/MultiTouch.cpp

2.2 Actor Class Reference

Basic simulation element for Angel.

```
#include <Actor.h>
```

Inheritance diagram for Actor:



Public Member Functions

- Actor ()
- virtual ∼Actor ()

- void SetSize (float x, float y=-1.f)
- void SetSize (const Vector2 &newSize)
- const Vector2 & GetSize () const
- const BoundingBox GetBoundingBox () const
- virtual void SetPosition (float x, float y)
- virtual void SetPosition (const Vector2 &pos)
- const Vector2 & GetPosition () const
- virtual void SetRotation (float rotation)
- const float GetRotation () const
- void SetColor (float r, float g, float b, float a=1.0f)
- void SetColor (const Color &color)
- const Color & GetColor () const
- void SetAlpha (float newAlpha)
- const float GetAlpha () const
- virtual void SetDrawShape (actorDrawShape drawShape)
- const actorDrawShape & GetDrawShape () const
- void UseDisplayList (int listIndex)
- void MoveTo (const Vector2 &newPosition, float duration, bool smooth=false, String onCompletion-Message="")
- void RotateTo (float newRotation, float duration, bool smooth=false, String onCompletionMessage="")
- void ChangeColorTo (const Color &newColor, float duration, bool smooth=false, String onCompletion-Message="")
- void ChangeSizeTo (const Vector2 &newSize, float duration, bool smooth=false, String onCompletion-Message="")
- void ChangeSizeTo (float newSize, float duration, bool smooth=false, String onCompletionMessage="")
- int GetSpriteTexture (int frame=0) const
- bool SetSprite (const String &filename, int frame=0, GLint clampmode=GL_CLAMP, GLint filtermode=GL_LI-NEAR, bool optional=false)
- void ClearSpriteInfo ()
- void LoadSpriteFrames (const String &firstFilename, GLint clampmode=GL_CLAMP, GLint filtermode=GL_-LINEAR)
- void PlaySpriteAnimation (float delay, spriteAnimationType animType=SAT_Loop, int startFrame=-1, int end-Frame=-1, const char ∗animName=NULL)
- void SetSpriteFrame (int frame)
- int GetSpriteFrame () const
- const bool IsSpriteAnimPlaying () const
- virtual void AnimCallback (String animName)
- void SetUVs (const Vector2 &lowleft, const Vector2 &upright)
- void GetUVs (Vector2 &lowleft, Vector2 &upright) const
- const bool IsTagged (const String &tag)
- void Tag (const String &newTag)
- void Untag (const String &oldTag)
- const StringSet & GetTags () const
- const String & SetName (String newName)
- const String & GetName () const
- virtual void ReceiveMessage (Message ∗message)
- void SetLayer (int layerIndex)
- void SetLayer (const String &layerName)
- virtual void Update (float dt)
- virtual void Render ()
- virtual void LevelUnloaded ()
- Actor ∗ GetSelf ()
- virtual const String GetClassName () const

Static Public Member Functions

- static Actor ∗const GetNamed (const String &nameLookup)
- static Actor ∗ Create (const String &archetype)
- static void SetScriptCreatedActor (Actor ∗a)

Protected Attributes

- Vector2 **_size**
- Vector2 **_position**
- Color **_color**
- float **_rotation**
- float **_UV** [8]
- actorDrawShape **_drawShape**
- int **_spriteCurrentFrame**
- int **_spriteNumFrames**
- float **_spriteFrameDelay**
- float **_spriteCurrentFrameDelay**
- int **_spriteTextureReferences** [MAX_SPRITE_FRAMES]
- spriteAnimationType **_spriteAnimType**
- int **_spriteAnimStartFrame**
- int **_spriteAnimEndFrame**
- int **_spriteAnimDirection**
- int **_displayListIndex**
- StringSet **_tags**
- String **_name**
- String **_currentAnimName**

Static Protected Attributes

- static const float **_squareVertices** [ ]
- static float **_circleVertices** [ ]
- static float **_circleTextureCoords** [ ]
- static std::map< String, Actor ∗ > **_nameList**
- static Actor ∗ **_scriptCreatedActor** = NULL

Additional Inherited Members

2.2.1 Detailed Description

Actor is the basic unit of simulation in Angel. It's not just an abstract base class – you can actually do a good deal of your game just using plain Actors, depending on how you want to structure your logic.

Note that you have to add the Actor to the World if you want to see it on screen.

If you want to subclass Actor, the two most important functions are Render and Update, which will get called on each Actor in the World every frame.

Definition at line 75 of file Actor.h.

2.2.2 Constructor & Destructor Documentation

2.2.2.1 Actor::Actor ( )

The constructor doesn't do anything special. It defaults to a square Actor with a side-length of 1.0f, no texture, white, and at the origin.

Definition at line 60 of file Actor.cpp.

2.2.2.2 Actor::∼Actor ( ) [virtual]

The destructor ensures that the Actor has been unsubscribed from all Messages, removed from all tag lists, and purged from the list of unique names.

Definition at line 97 of file Actor.cpp.

2.2.3 Member Function Documentation

2.2.3.1 void Actor::SetSize (float *x*, float *y* = `-1.f`)

Set the size of this Actor.

Parameters

x	Horizontal size in OpenGL units – negative numbers treated as zero
y	Vertical size in OpenGL units – if less than or equal to zero, assumed to be equal to x

Definition at line 326 of file Actor.cpp.

2.2.3.2 void Actor::SetSize (const Vector2 & *newSize*)

Set the size of this Actor.

Parameters

newSize	Desired size of Actor in OpenGL units. If either dimension is negative, it's clamped to zero.

Definition at line 340 of file Actor.cpp.

2.2.3.3 const Vector2 & Actor::GetSize ( ) const

Return the size of this Actor.

Returns

Actor's size as a Vector2

Definition at line 353 of file Actor.cpp.

2.2.3.4 const BoundingBox Actor::GetBoundingBox ( ) const

Return the BoundingBox for this Actor.

Returns

Actor's bounding box

Definition at line 358 of file Actor.cpp.

2.2.3.5 void Actor::SetPosition (float *x*, float *y*) [virtual]

Set the position of the Actor in world coordinates.

Parameters

x	X-coordinate in OpenGL units
y	Y-coordinate in OpenGL units

Reimplemented in PhysicsActor, Camera, and TextActor.

Definition at line 366 of file Actor.cpp.

2.2.3.6 void Actor::SetPosition (const Vector2 & *pos*) `[virtual]`

Set the position of the Actor in world coordinates.

Parameters

pos	Desired X and Y coordinates wrapped up into a Vector2

Reimplemented in PhysicsActor, Camera, and TextActor.

Definition at line 372 of file Actor.cpp.

2.2.3.7 const Vector2 & Actor::GetPosition ( ) const

Return the position of this Actor.

Returns

Actor's position as a Vector2

Definition at line 377 of file Actor.cpp.

2.2.3.8 void Actor::SetRotation (float *rotation*) `[virtual]`

Sets the rotation of the Actor. Positive rotations are counter-clockwise.

Parameters

rotation	Desired rotation in degrees

Reimplemented in PhysicsActor, Camera, and TextActor.

Definition at line 382 of file Actor.cpp.

2.2.3.9 const float Actor::GetRotation ( ) const

Return the rotation of this Actor.

Returns

Actor's rotation in degrees

Definition at line 387 of file Actor.cpp.

2.2.3.10 void Actor::SetColor (float *r*, float *g*, float *b*, float *a* = `1.0f`)

Sets the color of the Actor with individual components.

Parameters

r	Red
g	Green
b	Blue
a	Alpha (1.0f is opaque)

Definition at line 397 of file Actor.cpp.

2.2.3.11 void Actor::SetColor (const Color & *color*)

Sets the color of the Actor from a Color object.

Parameters

color	Desired color

Definition at line 402 of file Actor.cpp.

2.2.3.12 const Color & Actor::GetColor ( ) const

Return the Color of this Actor.

Returns

Actor's current color as a Color object

Definition at line 392 of file Actor.cpp.

2.2.3.13 void Actor::SetAlpha (float *newAlpha*)

Set the transparency of the Actor, independent of the other color components.

Parameters

newAlpha	Desired transparency (1.0f is opaque, 0.0f is invisible)

Definition at line 407 of file Actor.cpp.

2.2.3.14 const float Actor::GetAlpha ( ) const

Get current transparency for this Actor.

Returns

Actor's current transparency value.

Definition at line 412 of file Actor.cpp.

2.2.3.15 void Actor::SetDrawShape (actorDrawShape *drawShape*) [virtual]

Set the shape of the Actor when it's drawn.

See Also

actorDrawShape

Parameters

drawShape	Desired shape

Definition at line 229 of file Actor.cpp.

2.2.3.16 const actorDrawShape & Actor::GetDrawShape ( ) const

Get the current shape of this Actor.

Returns

Actor's shape

Definition at line 234 of file Actor.cpp.

2.2.3.17 void Actor::UseDisplayList (int *listIndex*) [inline]

Use a display list index for drawing rather than a built-in shape.

Parameters

listIndex	The index to use (generate using glGenLists())

Definition at line 221 of file Actor.h.

2.2.3.18 void Actor::MoveTo (const Vector2 & *newPosition*, float *duration*, bool *smooth* = false, String *onCompletionMessage* = "")

A "fire and forget" function that moves an Actor to a new position over a designated amount of time. This lets you handle movements without having to set up your own timers and keep track of them yourself. At the moment, it's limited to purely linear movement.

See Also

RotateTo
ChangeColorTo
ChangeSizeTo

Parameters

newPosition	The target position for the movement
duration	How long it should take to get there
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	If specified, a Message of this type will be sent when the movement is complete, letting you know when it's done. You will have to manually subscribe to this Message, though.

Definition at line 239 of file Actor.cpp.

2.2.3.19 void Actor::RotateTo (float *newRotation*, float *duration*, bool *smooth* = false, String *onCompletionMessage* = "")

A "fire and forget" function that rotates an Actor over a designated amount of time.

See Also

MoveTo

Parameters

newRotation	The target rotation
duration	How long it should take
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	the type of Message to be sent on completion

Definition at line 245 of file Actor.cpp.

2.2.3.20 void Actor::ChangeColorTo (const Color & *newColor*, float *duration*, bool *smooth* = false, String *onCompletionMessage* = "")

A "fire and forget" function that changes an Actor's color over a designated amount of time.

See Also

MoveTo

Parameters

newColor	The target color
duration	How long it should take
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	the type of Message to be sent on completion

Definition at line 251 of file Actor.cpp.

2.2.3.21 void Actor::ChangeSizeTo (const Vector2 & *newSize,* float *duration,* bool *smooth* = `false`, String *onCompletionMessage* = " ")

A "fire and forget" function that changes an Actor's size over a designated amount of time. This version uses a Vector2 if you want to set a non-uniform target size.

See Also

MoveTo

Parameters

newSize	The target size
duration	How long it should take
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	the type of Message to be sent on completion

Definition at line 257 of file Actor.cpp.

2.2.3.22 void Actor::ChangeSizeTo (float *newSize,* float *duration,* bool *smooth* = `false`, String *onCompletionMessage* = " ")

A "fire and forget" function that changes an Actor's size over a designated amount of time.

See Also

MoveTo

Parameters

newSize	The target size
duration	How long it should take
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	the type of Message to be sent on completion

Definition at line 263 of file Actor.cpp.

2.2.3.23 int Actor::GetSpriteTexture (int *frame* = `0`) const

Gets the OpenGL texture reference that the Actor is currently using when it draws itself.

Parameters

frame	If the Actor has an animation, you can retrieve the reference for a specific frame

Returns

The OpenGL texture reference

Definition at line 430 of file Actor.cpp.

2.2.3.24 bool Actor::SetSprite (const String & *filename*, int *frame* = `0`, GLint *clampmode* = `GL_CLAMP`, GLint *filtermode* = `GL_LINEAR`, bool *optional* = `false`)

Apply texture information to an Actor. The file can be any image format supported by DevIL, and transparency in the image will be used when drawing the Actor.

Parameters

filename	The path to an image file
frame	If you're building an animation for this Actor, you can specify the frame to which this texture should be assigned.
clampmode	The OpenGL clamp mode to use when drawing this sprite. can be either GL_CLAMP or GL_-REPEAT.
filtermode	The OpenGL filter mode to use when drawing this sprite. One of GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MI-PMAP_LINEAR, or GL_LINEAR_MIPMAP_LINEAR
optional	If set to true, the engine won't complain if it can't load this texture.

Returns

True if the sprite was successfully set, false otherwise

Definition at line 440 of file Actor.cpp.

2.2.3.25 void Actor::ClearSpriteInfo ( )

Remove all sprite information from an Actor

Definition at line 450 of file Actor.cpp.

2.2.3.26 void Actor::LoadSpriteFrames (const String & *firstFilename*, GLint *clampmode* = `GL_CLAMP`, GLint *filtermode* = `GL_LINEAR`)

A convenience function for loading up a directory of image files as an animation. We expect the name of the first image to end in _###, where

represents a number. The number of digits you put at the end

doesn't matter, but we are internally limited to 64 frames. If you want more, just change MAX_SPRITE_FRAMES in Actor.h.

From the first file that you pass the function, it will iterate through the file's directory and sequentially load up any images that follow the same naming pattern. So if you had a directory with `anim_001.png`, `anim_002.png`, and `anim_003.png`, you could load the three-frame animation by passing "anim_001.png" to this function.

Parameters

firstFilename	The starting file for the animation
clampmode	The clamp mode to be used by the SetSprite function
filtermode	The filter mode to be used by the SetSprite function

Definition at line 488 of file Actor.cpp.

2.2.3.27 void Actor::PlaySpriteAnimation (float *delay*, spriteAnimationType *animType* = `SAT_Loop`, int *startFrame* = `-1`, int *endFrame* = `-1`, const char * *animName* = `NULL`)

Actually triggers the loaded animation to start playing.

Parameters

delay	The amount of time between frames
animType	How the animation should behave when it's finished. Options are SAT_Loop, SAT_PingPong, and SAT_OneShot.
startFrame	The starting frame of the animation to play
endFrame	The ending frame of the animation to play
animName	The name of the animation so you can get the event when it finishes.

Definition at line 473 of file Actor.cpp.

2.2.3.28 void Actor::SetSpriteFrame (int *frame*)

Manually set the frame to draw. This way you can have an "animation" of frames that just represent states of your actor and swap between them without having to reload the textures all the time.

Parameters

frame	The frame to switch to

Definition at line 461 of file Actor.cpp.

2.2.3.29 int Actor::GetSpriteFrame ( ) const `[inline]`

Get the current animation frame, ranging from 0 to MAX_SPRITE_FRAMES.

Returns

The current animation frame.

Definition at line 381 of file Actor.h.

2.2.3.30 const bool Actor::IsSpriteAnimPlaying ( ) const `[inline]`

Lets you find out if an animation is currently playing on this Actor.

Returns

True if there's an animation playing, false if there isn't.

Definition at line 388 of file Actor.h.

2.2.3.31 virtual void Actor::AnimCallback (String *animName*) `[inline]`, `[virtual]`

A function you can override in the subclass if you you want your Actor to do certain things when an animation finishes. This function will get called by the animation system and pass in the string you assigned when calling PlaySpriteAnimation.

Parameters

animName	The animation's name

Definition at line 401 of file Actor.h.

2.2.3.32 void Actor::SetUVs (const Vector2 & *lowleft*, const Vector2 & *upright*)

If you're doing fancy things with moving textures, this function lets you alter the UV (texture coordinates) of the actor.

Parameters

lowleft	The desired lower left UV
upright	The desired upper right UV

Definition at line 582 of file Actor.cpp.

2.2.3.33 void Actor::GetUVs (Vector2 & *lowleft*, Vector2 & *upright*) const

Get the current UV coordinates being used by the Actor to draw.

Parameters

lowleft	An out parameter that will be set to the current lower left UV
upright	An out parameter that will be set to the current upper right UV

Definition at line 594 of file Actor.cpp.

2.2.3.34 const bool Actor::IsTagged (const String & *tag*)

Returns whether or not this Actor has been given a particular tag.

See Also

TagCollection

Parameters

tag	the tag in question

Returns

True if the Actor has the tag

Definition at line 602 of file Actor.cpp.

2.2.3.35 void Actor::Tag (const String & *newTag*)

Adds a tag to an Actor. If the Actor already has this tag, no action is taken.

See Also

TagCollection

Parameters

newTag	The tag to add

Definition at line 616 of file Actor.cpp.

2.2.3.36 void Actor::Untag (const String & *oldTag*)

Removes a tag from an Actor. If the Actor doesn't have this tag, no action is taken.

See Also

TagCollection

Parameters

oldTag	The tag to remove

Definition at line 627 of file Actor.cpp.

2.2.3.37 const StringSet & Actor::GetTags ( ) const

Get all the tags for this ACtor.

See Also

TagCollection

Returns

A StringSet (std::vector<std::string>) of all the Actor's tags

Definition at line 634 of file Actor.cpp.

2.2.3.38 const String & Actor::SetName (String *newName*)

Give this Actor a name that can later be used as a unique identifier. The the actual name given may differ from what is passed in, but is guaranteed to be unique. (A global monotonically increasing number will be appended until the name is distinct.)

Parameters

newName	The desired name

Returns

The actual name that was given

Definition at line 639 of file Actor.cpp.

2.2.3.39 const String & Actor::GetName ( ) const

Get the unique name assigned to this Actor.

Returns

The Actor's current name.

Definition at line 666 of file Actor.cpp.

2.2.3.40 Actor *const Actor::GetNamed (const String & *nameLookup*) [static]

A static function of the Actor class which returns an Actor from a name.

Parameters

nameLookup	The name index to look for

Returns

The Actor with the given name. Will be NULL if there's no match

Definition at line 671 of file Actor.cpp.

2.2.3.41 virtual void Actor::ReceiveMessage (Message * *message*) [inline], [virtual]

An implementation of the MessageListener interface, which will be called when a message gets delivered.

There is no actual implementation in the base class, but you can override in the subclass.

See Also

MessageListener

Parameters

message	The message getting delivered.

Implements MessageListener.

Reimplemented in TextActor, and FullScreenActor.

Definition at line 494 of file Actor.h.

2.2.3.42 void Actor::SetLayer (int *layerIndex*)

Set a new rendering layer for this Actor.

Layers are ordered from bottom to top by index. No space is wasted by leaving empty layers in between, so feel free to pad out your indices if you want.

Parameters

layerIndex	the index of the render layer you want to assign

Definition at line 703 of file Actor.cpp.

2.2.3.43 void Actor::SetLayer (const String & *layerName*)

Set a new rendering layer for this Actor by the name of the layer.

The name of the layer has to be set up first by calling World::NameLayer. If you pass an invalid layer name, this Actor will be put on layer 0.

See Also

SetLayer

Parameters

layerName	the name of the render layer you want to assign

Definition at line 708 of file Actor.cpp.

2.2.3.44 void Actor::Update (float *dt*) [virtual]

A function which makes the necessary updates to the Actor. The base implementation just updates the animations and intervals, but a subclass override can perform whatever extra magic is necessary. Make sure to call the base class's Update if you subclass.

Parameters

dt	The amount of time that's elapsed since the beginning of the last frame.

Reimplemented from Renderable.

Reimplemented in Camera, ParticleActor, and Sentient.

Definition at line 110 of file Actor.cpp.

2.2.3.45 void Actor::Render ( ) [virtual]

A function to draw the Actor to the screen. By default, this does the basic drawing based on the texture, color, shape, size, position, and rotation that have been applied to the Actor. Can be overridden in a subclass if necessary.

This will get called on every Actor once per frame, after the Update.

Reimplemented from Renderable.

Reimplemented in Camera, TextActor, ParticleActor, HUDActor, and Sentient.

Definition at line 268 of file Actor.cpp.

2.2.3.46 virtual void Actor::LevelUnloaded ( ) [inline], [virtual]

Called for every actor that doesn't get unloaded in World::UnloadAll(). This is a good place to clear out any cached pointers, etc.

Definition at line 544 of file Actor.h.

2.2.3.47 Actor∗ Actor::GetSelf ( ) [inline]

Yes, this looks pointless and redundant. But it has a function for the scripting layer – it activates the inheritance downcasts so you can get a derived object from its base pointer.

In most instances, an Actor∗ getting thrown to the script layer will be wrapped correctly as a PhysicsActor, TextActor, etc. BUT, if it's coming from an STL container, then it won't. Rather than trying to fiddle with the SWIG typemaps which wrap the STL, this solution is more robust. It does require an extra call in the script, but c'est la vie.

Returns

the Actor's "this" pointer

Definition at line 560 of file Actor.h.

2.2.3.48 Actor ∗ Actor::Create (const String & *archetype*) [static]

Create an Actor from an archetype defined in a .lua file in Config/ActorDef. Automatically adds the Actor to the World.

The table names in the .lua files designate the name of the archetype, while the values in each table specify the properties for that archetype. Any function that can be called on an Actor can be used as a property – things like SetSize can be called simply "size."

Colors and Vectors can be defined as tables, so the following definition is valid.

```
my_actor = {
  color = {1, 0, 1},
  alpha = 0.5,
  size = 5,
}
```

Parameters

archetype	the name of the Actor archetype (the table name from the .lua file)

Definition at line 684 of file Actor.cpp.

2.2.3.49 static void Actor::SetScriptCreatedActor (Actor ∗ *a*) [inline], [static]

This static function is used internally by the scripting layer to let the core engine get at Actors that created in script. If that doesn't make complete sense to you, you probably have no need to call this function.

Parameters

a	The actor that was just created in script

Definition at line 598 of file Actor.h.

2.2.3.50 virtual const String Actor::GetClassName ( ) const [inline], [virtual]

Used by the SetName function to create a basename for this class. Overridden in derived classes.

Returns

The string "Actor"

Reimplemented in PhysicsActor, Camera, ParticleActor, TextActor, FullScreenActor, and HUDActor.

Definition at line 606 of file Actor.h.

2.2.4 Member Data Documentation

2.2.4.1 const float Actor::_squareVertices [static], [protected]

Initial value:

```
= {
    -0.5f,  0.5f,
    -0.5f, -0.5f,
     0.5f,  0.5f,
     0.5f, -0.5f,
}
```

Definition at line 617 of file Actor.h.

The documentation for this class was generated from the following files:

- Actors/Actor.h
- Actors/Actor.cpp

2.3 AIBrain Class Reference

Public Member Functions

- void **SetActor** (Sentient ∗actor)
- virtual void **AddState** (const String &id, AIBrainState ∗state)
- virtual void **Update** (float dt)
- virtual void **GotoState** (const String &id)
- Sentient ∗ **GetActor** ()
- void **Render** ()
- void **GotoNullState** ()
- void **EnableDrawing** (bool enable)

Protected Attributes

- BrainStateTable **_brainStateTable**
- BrainStateTable::iterator **_current**
- Sentient ∗ **_actor**
- bool **_drawMe**

2.3.1 Detailed Description

Definition at line 37 of file Brain.h.

The documentation for this class was generated from the following files:

- AI/Brain.h
- AI/Brain.cpp

2.4 AIBrainState Class Reference

Public Member Functions

- virtual void **Initialize** (AIBrain ∗brain)
- void **Update** (float dt)
- virtual void **CustomUpdate** (float)
- virtual void **BeginState** (AIBrainState ∗)
- void **EndState** (AIBrainState ∗nextState)
- virtual void **CustomEndState** (AIBrainState ∗)

Protected Member Functions

- virtual void **GotoState** (const String &id)
- virtual AIEvent ∗ **RegisterEvent** (AIEvent ∗newEvent)
- virtual void **UnregisterEvent** (AIEvent ∗oldEvent)
- Sentient ∗ **GetActor** ()

Protected Attributes

- AIBrain ∗ **_brain**
- EventList **_eventList**

2.4.1 Detailed Description

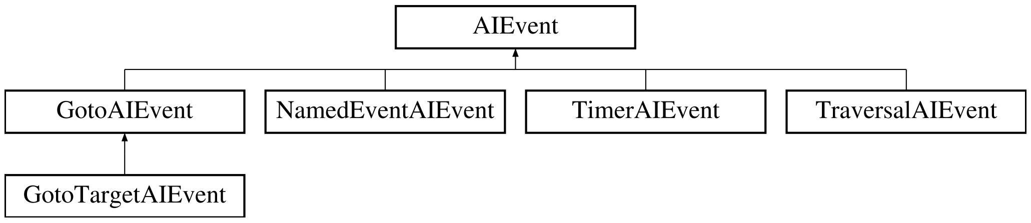
Definition at line 68 of file Brain.h.

The documentation for this class was generated from the following files:

- AI/Brain.h
- AI/Brain.cpp

2.5 AIEvent Class Reference

Inheritance diagram for AIEvent:



Public Member Functions

- virtual void **Stop** ()
- virtual void **Update** (float)
- void **SetBrain** (AIBrain ∗pBrain)

Protected Member Functions

- AIBrain ∗ **GetBrain** ()
- Sentient ∗ **GetActor** ()
- virtual void **IssueCallback** ()

2.5.1 Detailed Description

Definition at line 116 of file Brain.h.

The documentation for this class was generated from the following file:

- AI/Brain.h

2.6 AStarSearch< UserState > Class Template Reference

Classes

- class HeapCompare_f
- class Node

Public Types

- enum {
SEARCH_STATE_NOT_INITIALISED, **SEARCH_STATE_SEARCHING**, **SEARCH_STATE_SUCCEEDE-D**, **SEARCH_STATE_FAILED**,
SEARCH_STATE_OUT_OF_MEMORY, **SEARCH_STATE_INVALID** }

Public Member Functions

- int **GetState** ()
- void **CancelSearch** ()
- void **SetStartAndGoalStates** (UserState &Start, UserState &Goal)
- unsigned int **SearchStep** ()
- bool **AddSuccessor** (UserState &State)

- void **FreeSolutionNodes** ()
- UserState ∗ **GetSolutionStart** ()
- UserState ∗ **GetSolutionNext** ()
- UserState ∗ **GetSolutionEnd** ()
- UserState ∗ **GetSolutionPrev** ()
- UserState ∗ **GetOpenListStart** ()
- UserState ∗ **GetOpenListStart** (float &f, float &g, float &h)
- UserState ∗ **GetOpenListNext** ()
- UserState ∗ **GetOpenListNext** (float &f, float &g, float &h)
- UserState ∗ **GetClosedListStart** ()
- UserState ∗ **GetClosedListStart** (float &f, float &g, float &h)
- UserState ∗ **GetClosedListNext** ()
- UserState ∗ **GetClosedListNext** (float &f, float &g, float &h)
- int **GetStepCount** ()
- void **EnsureMemoryFreed** ()

2.6.1 Detailed Description

template<class UserState>class AStarSearch< UserState >

Definition at line 53 of file stlastar.h.

The documentation for this class was generated from the following file:

- AI/stlastar.h

2.7 BoundingBox Struct Reference

Public Member Functions

- **BoundingBox** (const Vector2 &min, const Vector2 &max)
- Vector2 **Centroid** () const
- Vector2 **HalfLength** () const
- void **GetCorners** (Vector2 corners[ ]) const
- bool **Intersects** (const BoundingBox &box) const
- bool **Intersects** (const Ray2 &ray, float &distanceAlongRay) const
- bool **Intersects** (const Vector2 &point, float radius) const
- ContainmentType **Contains** (const BoundingBox &box) const
- bool **Contains** (const Vector2 &point) const
- void **RenderOutline** () const
- void **RenderBox** () const

Static Public Member Functions

- static BoundingBox **CreateMerged** (const BoundingBox &original, const BoundingBox &additional)
- static BoundingBox **CreateFromPoints** (Vector2 points[ ], int count)

Public Attributes

- Vector2 **Min**
- Vector2 **Max**

2.7.1 Detailed Description

Definition at line 43 of file BoundingShapes.h.

The documentation for this struct was generated from the following files:

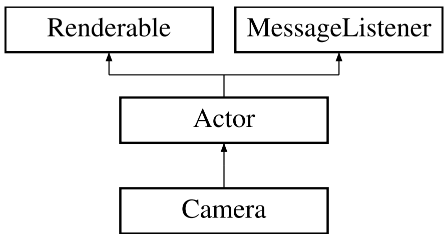
- AI/BoundingShapes.h
- AI/BoundingShapes.cpp

2.8 Camera Class Reference

The class that handles displaying the appropriate viewport.

`#include <Camera.h>`

Inheritance diagram for Camera:



Public Member Functions

- void Destroy ()
- virtual void Update (float dt)
- void Render ()
- void Reset ()
- void LockTo (Actor ∗locked, bool lockX=true, bool lockY=true, bool lockRotation=false)
- Actor ∗ GetLockedActor ()
- const int GetWindowHeight () const
- const int GetWindowWidth () const
- const double GetViewRadius () const
- const Vector2 GetWorldMaxVertex () const
- const Vector2 GetWorldMinVertex () const
- virtual void SetPosition (float x, float y, float z)
- virtual void SetPosition (float x, float y)
- virtual void SetPosition (const Vector3 &v3)
- virtual void SetPosition (const Vector2 &v2)
- void MoveTo (const Vector3 &newPosition, float duration, bool smooth=false, String onCompletion-Message="")
- virtual Vector2 GetPosition () const
- virtual float GetZ () const
- virtual void SetRotation (float rotation)
- virtual float GetZForViewRadius (float radius)
- virtual float GetNearClipDist ()
- virtual float GetFarClipDist ()
- virtual void SetZByViewRadius (float newRadius)
- virtual void SetNearClipDist (float dist)
- virtual void SetFarClipDist (float dist)
- virtual void SetViewCenter (float x, float y, float z)
- virtual const Vector3 & GetViewCenter () const
- virtual const String GetClassName () const

Static Public Member Functions

- static Camera & GetInstance ()
- static void ResizeCallback (GLFWwindow ∗window, int w, int h)

Protected Member Functions

- void **Resize** (int w, int h)

Static Protected Attributes

- static Camera ∗ **s_Camera** = NULL

Additional Inherited Members

2.8.1 Detailed Description

The Camera class is how you control what your players see at any time. It uses the singleton pattern; you can't actually declare a new instance of a Camera. To access the Camera in your world, use "theCamera" to retrieve the singleton object. "theCamera" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Camera is an Actor so that you can apply motion to it the same way you would any other Actor in your world.

By default the camera is positioned 10 units away from the origin, looking down the Z-axis (0.0, 0.0, 10.0). The visible world stretches from (-13.3, -10) to (13.3, 10).

Whenever the Camera updates its position or the viewport, it broadcasts a "CameraChange" Message that you can subscribe to if you need notifications.

Definition at line 59 of file Camera.h.

2.8.2 Member Function Documentation

2.8.2.1 Camera & Camera::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theCamera".

Returns

The singleton

Definition at line 47 of file Camera.cpp.

2.8.2.2 void Camera::ResizeCallback (GLFWwindow ∗ *window*, int *w*, int *h*) `[static]`

The callback used by GLFW to alert us when the user changes the size of the window. You shouldn't need to mess with this.

Parameters

w	New width in pixels
h	New height in pixels

Definition at line 64 of file Camera.cpp.

2.8.2.3 void Camera::Destroy ( )

Deletes and NULLs out the singleton – should only get called at World destruction when the program is exiting.

Definition at line 57 of file Camera.cpp.

2.8.2.4 void Camera::Update (float *dt*) `[virtual]`

Override of actor's Update method to deal with the actor lock

Reimplemented from Actor.

Definition at line 134 of file Camera.cpp.

2.8.2.5 void Camera::Render ( ) `[virtual]`

Makes sure the view matrix is properly set up on every frame. Called by the world before rendering anything else.

Reimplemented from Actor.

Definition at line 178 of file Camera.cpp.

2.8.2.6 void Camera::Reset ( )

Resets the viewport to its default.

Definition at line 70 of file Camera.cpp.

2.8.2.7 void Camera::LockTo (Actor ∗ *locked*, bool *lockX* = `true`, bool *lockY* = `true`, bool *lockRotation* = `false`)

Locks the camera to a specific Actor so it will track it around.

Parameters

locked	The actor to track
lockX	Whether to track the actor's X position
lockY	Whether to track the actor's Y position
lockRotation	Whether to track the actor's rotation

Definition at line 81 of file Camera.cpp.

2.8.2.8 Actor∗ Camera::GetLockedActor ( ) `[inline]`

Get the actor that the camera is currently tracking.

Returns

The tracked actor (NULL if not tracking anything)

Definition at line 119 of file Camera.h.

2.8.2.9 const int Camera::GetWindowHeight ( ) const

Get the window's current height.

Returns

Height in pixels.

Definition at line 108 of file Camera.cpp.

2.8.2.10 const int Camera::GetWindowWidth ( ) const

Get the window's current width.

Returns

Width in pixels.

Definition at line 113 of file Camera.cpp.

2.8.2.11 const double Camera::GetViewRadius ( ) const

If you were to draw a circle inscribed in the viewport, this function will let you know the size of its radius. This is useful for determining how much of the world is currently being displayed.

Note that if you have a non-square viewport, the circle is bounded by the **smaller** dimension.

Returns

The radius size in GL units.

Definition at line 118 of file Camera.cpp.

2.8.2.12 const Vector2 Camera::GetWorldMaxVertex ( ) const

Get the world coordinate of the top-right point of the window.

Returns

The world coordinate (GL units).

Definition at line 124 of file Camera.cpp.

2.8.2.13 const Vector2 Camera::GetWorldMinVertex ( ) const

Get the world coordinate of the bottom-left point of the window.

Returns

The world coordinate (GL units).

Definition at line 129 of file Camera.cpp.

2.8.2.14 void Camera::SetPosition (float *x*, float *y*, float *z*) `[virtual]`

Set the position of the camera. Note that the camera is the only Actor that can take a Z coordinate for its position – you can zoom in and out.

Parameters

x	The new X position for the camera
y	The new Y position for the camera
z	The new Z position for the camera

Definition at line 194 of file Camera.cpp.

2.8.2.15 void Camera::SetPosition (float *x*, float *y*) `[virtual]`

Set the position of the camera. Using this two-dimensional function, the position on the Z-axis stays fixed.

Parameters

x	The new X position for the camera
y	The new Y position for the camera

Reimplemented from Actor.

Definition at line 200 of file Camera.cpp.

2.8.2.16 void Camera::SetPosition (const Vector3 & *v3*) [virtual]

Set the position of the camera. Note that the camera is the only Actor that can take a Z coordinate for its position – you can zoom in and out.

Parameters

v3	The new position for the camera.

Definition at line 214 of file Camera.cpp.

2.8.2.17 void Camera::SetPosition (const Vector2 & *v2*) [virtual]

Set the position of the camera. Using this two-dimensional function, the position on the Z-axis stays fixed.

Parameters

v2	The new position for the Camera

Reimplemented from Actor.

Definition at line 207 of file Camera.cpp.

2.8.2.18 void Camera::MoveTo (const Vector3 & *newPosition*, float *duration*, bool *smooth* = false, String *onCompletionMessage* = " ")

Interval movement for the camera in three dimensions.

See Also

Actor::MoveTo

Parameters

newPosition	The target position for the movement
duration	How long it should take to get there
smooth	Whether the function should use MathUtil::SmoothStep instead of MathUtil::Lerp
onCompletion-Message	If specified, a Message of this type will be sent when the movement is complete, letting you know when it's done. You will have to manually subscribe to this Message, though.

Definition at line 221 of file Camera.cpp.

2.8.2.19 Vector2 Camera::GetPosition ( ) const [virtual]

Gets the position of the Camera. Only returns the X and Y position so as to fit with the other GetPosition functions in the engine. To get the Z coordinate, see below.

Returns

The (X, Y) position of the camera.

Definition at line 227 of file Camera.cpp.

2.8.2.20 float Camera::GetZ ( ) const [virtual]

Get the position of the camera on the Z-axis.

Returns

The camera's Z coordinate.

Definition at line 232 of file Camera.cpp.

2.8.2.21 void Camera::SetRotation (float *rotation*) [virtual]

Set the rotation of the camera. Only rotates about the Z-axis, since we prefer the 2-dimensional.

Parameters

v2	The new rotation for the Camera. As with Actors, positive rotations are counter-clockwise.

Reimplemented from Actor.

Definition at line 237 of file Camera.cpp.

2.8.2.22 float Camera::GetZForViewRadius (float *radius*) [virtual]

Get the Z value necessary to achieve the requested view radius.

Parameters

radius	The desired view radius.

Returns

The Z value.

Definition at line 243 of file Camera.cpp.

2.8.2.23 float Camera::GetNearClipDist ( ) [virtual]

Get the near clip distance.

Returns

The near clip distance.

Definition at line 249 of file Camera.cpp.

2.8.2.24 float Camera::GetFarClipDist ( ) [virtual]

Get the far clip distance.

Returns

The far clip distance.

Definition at line 254 of file Camera.cpp.

2.8.2.25 void Camera::SetZByViewRadius (float *newRadius*) [virtual]

Set the Z value necessary to achieve the requested view radius.

Parameters

newRadius	The desired view radius.

Definition at line 259 of file Camera.cpp.

2.8.2.26 void Camera::SetNearClipDist (float *dist*) `[virtual]`

Set the near clip distance. Note this will cause a Resize() to properly update the clipping planes

Parameters

dist	The near clip distance.

Definition at line 265 of file Camera.cpp.

2.8.2.27 void Camera::SetFarClipDist (float *dist*) `[virtual]`

Set the far clip distance. Note this will cause a Resize() to properly update the clipping planes

Parameters

dist	The far clip distance.

Definition at line 271 of file Camera.cpp.

2.8.2.28 void Camera::SetViewCenter (float *x*, float *y*, float *z*) `[virtual]`

Set the point towards which the camera should aim. Since Angel is a predominantly 2D world, be very careful setting this too far off of perpendicular.

Parameters

x	The X coordinate at which the Camera will aim.
y	The Y coordinate at which the Camera will aim.
z	The Z coordinate at which the Camera will aim.

Definition at line 277 of file Camera.cpp.

2.8.2.29 const Vector3 & Camera::GetViewCenter ( ) const `[virtual]`

Get the current look-at target of the camera.

Returns

The point where the Camera is currently looking.

Definition at line 283 of file Camera.cpp.

2.8.2.30 virtual const String Camera::GetClassName ( ) const `[inline]`, `[virtual]`

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "Camera"

Reimplemented from Actor.

Definition at line 306 of file Camera.h.

The documentation for this class was generated from the following files:

- Infrastructure/Camera.h
- Infrastructure/Camera.cpp

2.9 Color Class Reference

A class to encapsulate color information.

`#include <Color.h>`

Public Member Functions

- Color ()
- Color (float r, float g, float b, float a=1.0f, bool clamp=true)
- bool **operator==** (const Color &c) const
- bool **operator!=** (const Color &c) const
- Color **operator-** (const Color &c) const
- Color **operator+** (const Color &c) const
- Color **operator/** (float divider) const
- Color **operator**∗ (float scaleFactor) const

Static Public Member Functions

- static Color FromInts (int r, int g, int b, int a=255, bool clamp=true)
- static Color FromHexString (String hexString)

Public Attributes

- float R
- float G
- float B
- float A

2.9.1 Detailed Description

This class consolidates all color information into a single unit that can be easily passed around functions, manipulated, and lerped.

The four color components are public members since they are frequently accessed and there is no real reason to hide them behind accessors. They are always stored as floats and range from 0.0f to 1.0f.

Note that the common arithmetical and comparison operators are defined for this class, to make it easy to mathematically manipulate the color of your Actors.

Definition at line 47 of file Color.h.

2.9.2 Constructor & Destructor Documentation

2.9.2.1 Color::Color ( )

The default constructor creates an opaque, pure white (all components are 1.0f).

Definition at line 36 of file Color.cpp.

2.9.2.2 Color::Color (float *r*, float *g*, float *b*, float *a* = `1.0f`, bool *clamp* = `true`)

A constructor to specify component values right from the start.

Parameters

r	The Red component
g	The Green component
b	The Blue component
a	The Alpha component
clamp	Whether or not to clamp the components to the range 0.0 to 1.0. Usually you want to do this (and the default reflects this case), unless you're doing some kind of nutty color math, in which case you should set this to false.

Definition at line 42 of file Color.cpp.

2.9.3 Member Function Documentation

2.9.3.1 Color Color::FromInts (int *r*, int *g*, int *b*, int *a* = `255`, bool *clamp* = `true`) `[static]`

A function to specify colors as integers from 0 to 255. Useful if you've got a color picker you like that gives you values in this range.

Note that internally the numbers are converted to floats from 0 to 1.

Parameters

r	The Red component
g	The Green component
b	The Blue component
a	The Alpha component
clamp	Whether or not to clamp the components to the range 0.0 to 1.0. Usually you want to do this (and the default reflects this case), unless you're doing some kind of nutty color math, in which case you should set this to false.

Definition at line 55 of file Color.cpp.

2.9.3.2 Color Color::FromHexString (String *hexString*) `[static]`

A function to specify a color as a hex string, like in CSS. For those used to specifying colors for web pages, you have an option here.

Parameters

hexString	The string identifying the color ("0xfff", "#fa6244", etc.)

Definition at line 71 of file Color.cpp.

2.9.4 Member Data Documentation

2.9.4.1 float Color::R

The Red component

Definition at line 53 of file Color.h.

2.9.4.2 float Color::G

The Green component

Definition at line 57 of file Color.h.

2.9.4.3 float Color::B

The Blue component

Definition at line 61 of file Color.h.

2.9.4.4 float Color::A

The Alpha component (1.0 == opaque, 0.0 == invisible)

Definition at line 65 of file Color.h.

The documentation for this class was generated from the following files:

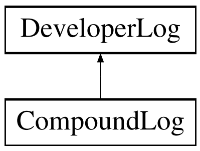
- Infrastructure/Color.h
- Infrastructure/Color.cpp

2.10 CompoundLog Class Reference

Lets you write to multiple logs at once.

`#include <Log.h>`

Inheritance diagram for CompoundLog:



Public Member Functions

- void AddLog (DeveloperLog ∗addLog)
- virtual void Log (const String &val)

Static Public Member Functions

- static CompoundLog & GetSystemLog ()

2.10.1 Detailed Description

This class collects various other logs together and lets you write the same value to them simultaneously. This is useful if you want to write something to both the screen and a file at the same time.

Definition at line 146 of file Log.h.

2.10.2 Member Function Documentation

2.10.2.1 void CompoundLog::AddLog (DeveloperLog ∗ *addLog*)

Add a log to the list of receivers.

Parameters

addLog	The log you wish to add.

Definition at line 72 of file Log.cpp.

2.10.2.2 void CompoundLog::Log (const String & *val*) [virtual]

Writes a string to all registered logs

Parameters

val	The string to log.

Implements DeveloperLog.

Definition at line 84 of file Log.cpp.

2.10.2.3 CompoundLog & CompoundLog::GetSystemLog ( ) [static]

A reference to the system log (where Angel will spew its information, and to which you can attach another log if you want).

Returns

The system log

Definition at line 92 of file Log.cpp.

The documentation for this class was generated from the following files:

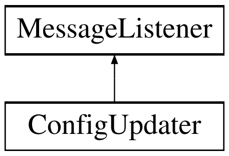
- Infrastructure/Log.h
- Infrastructure/Log.cpp

2.11 ConfigUpdater Class Reference

(Internal) Just sits and checks the tuning file periodically

`#include <LuaModule.h>`

Inheritance diagram for ConfigUpdater:



Public Member Functions

- void **Reload** ()
- virtual void ReceiveMessage (Message ∗message)

2.11.1 Detailed Description

This internal class periodically checks the tuning.lua file in the Config directory and reloads it if it's been modified. This allows you to change the file at runtime and see your tuning affects right away.

Definition at line 48 of file LuaModule.h.

2.11.2 Member Function Documentation

2.11.2.1 void ConfigUpdater::ReceiveMessage (**Message** ∗ *m*) `[virtual]`

The Switchboard class will call this function for every Message to be deliverd to a designated listener. Implementations of this function should filter on Message::GetMessageName to make sure they're responding to the appropriate signals.

Parameters

m	The message to be delivered.

Implements MessageListener.

Definition at line 216 of file LuaModule.cpp.

The documentation for this class was generated from the following files:

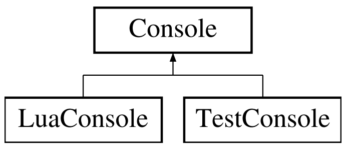
- Scripting/LuaModule.h
- Scripting/LuaModule.cpp

2.12 Console Class Reference

The on-screen console: handles input, history, and executing commands.

`#include <Console.h>`

Inheritance diagram for Console:



Public Member Functions

- Console ()
- virtual ∼Console ()
- void Render ()
- void Update (float)
- void Enable (bool bEnable=true)
- bool IsEnabled ()
- bool GetInput (int key)
- bool GetSpecialInputDown (int key)
- void ToggleConsole ()
- unsigned char GetToggleConsoleKey ()
- void WriteToOutput (String output)
- void SetPrompt (const String &prompt)
- const unsigned int GetTabWidth ()
- void SetTabWidth (unsigned int newTabWidth)
- void AdvanceInputHistory (int byVal)
- void AcceptAutocomplete ()
- virtual void Execute (String input)=0
- virtual StringList GetCompletions (const String &input)=0

Protected Member Functions

- bool **IsTextKey** (unsigned char key)
- void **AcceptCurrentInput** ()

Protected Attributes

- String **_currentInput**
- StringList **_inputHistory**
- int **_inputHistoryPos**
- String **_prompt**
- StringList **_buffer**
- String **_unsplitBuffer**
- float **_lineHeight**
- StringList **_autoCompleteList**
- int **_cursorPos**
- unsigned int **_tabWidth**

2.12.1 Detailed Description

This abstract base class contains everything you need to create your own console that responds to user input. The base class handles displaying and hiding the console, taking input, and recording command history.

To make a console, your subclass needs to implement the Execute function (which gets called with a string whenever the user enters input) and the GetCompletions function, which returns a StringList of potential auto-complete values.

Most of this is academic, as you'll probably use the Lua console, which has already implemented all of this for you.

There's also a TestConsole implementation that does nothing but echo your commands back at you. It's the default console, so if you ever see its prompt (::>), you know something has gone wrong with loading the real console.

Definition at line 55 of file Console.h.

2.12.2 Constructor & Destructor Documentation

2.12.2.1 Console::Console ( )

The default constructor makes sure our console font (Inconsolata at) 18 points) is registered with the text rendering system.

Definition at line 44 of file Console.cpp.

2.12.2.2 Console::∼Console ( ) `[virtual]`

Does nothing in the base class, but you gotta have it

Definition at line 60 of file Console.cpp.

2.12.3 Member Function Documentation

2.12.3.1 void Console::Render ( )

Override of the Renderable::Render function that actually draws the console, if necessary.

Definition at line 399 of file Console.cpp.

2.12.3.2 void Console::Update (float *dt*)

Oddly enough, doesn't do anything. All the changes happen at user input instead of in the update loop.

Parameters

dt	

Definition at line 386 of file Console.cpp.

2.12.3.3 void Console::Enable (bool *bEnable* = `true`)

Turns the console on (or off), which makes it get drawn in the render loop and steals keyboard input from the rest of the game.

Parameters

bEnable	If true, the Console is activate; if false, it's deactivated.

Definition at line 108 of file Console.cpp.

2.12.3.4 bool Console::IsEnabled ( ) `[inline]`

Whether the Console is currently being displayed and accepting input right now.

Returns

True if it's enabled, false if it's not

Definition at line 96 of file Console.h.

2.12.3.5 bool Console::GetInput (int *key*)

Adds the given key to the Console's current input string. Should only be called by the input handling functions.

Parameters

key	The numeric representation of the key

Returns

Whether or not the key was added to the input string (false if the Console is currently disabled or that was the key which toggled it.)

Definition at line 113 of file Console.cpp.

2.12.3.6 bool Console::GetSpecialInputDown (int *key*)

Handles special input like Enter, Escape, arrow keys, etc. Again, this should only be called by the input handling functions.

Parameters

key	The numeric representation of the special key

Returns

Whether or not the input was accepted by the Console (false if the Console is currently disabled.)

Definition at line 141 of file Console.cpp.

2.12.3.7 void Console::ToggleConsole ( )

Switches the console from on to off, or from off to on.

Definition at line 366 of file Console.cpp.

2.12.3.8 unsigned char Console::GetToggleConsoleKey ( ) [inline]

Lets you know what key is set to toggle the Console. (Hard-coded to '`' at the moment.)

Returns

The toggle console key

Definition at line 129 of file Console.h.

2.12.3.9 void Console::WriteToOutput (String *output*)

Writes a string to the log area of the console. Used internally by Console implementations to display the results of commands.

Parameters

output	The string to be appended to the log area.

Definition at line 246 of file Console.cpp.

2.12.3.10 void Console::SetPrompt (const String & *prompt*)

Set the prompt characters that users will see at the beginning of their input line. Some consoles like to change this to reflect their current state.

Parameters

prompt	The new prompt string

Definition at line 371 of file Console.cpp.

2.12.3.11 const unsigned int Console::GetTabWidth ( )

Get the width of tabs when output to this console.

Definition at line 381 of file Console.cpp.

2.12.3.12 void Console::SetTabWidth (unsigned int *newTabWidth*)

Get the width of tabs when output to this console. Defaults to 8.

Parameters

newTabWidth	The desired tab width in spaces

Definition at line 376 of file Console.cpp.

2.12.3.13 void Console::AdvanceInputHistory (int *byVal*)

Replaces the current input string with a value from the input history.

Parameters

byVal	The offset from the current history. Negative numbers will go back in the history, positive ones will go forward.

Definition at line 338 of file Console.cpp.

2.12.3.14 void Console::AcceptAutocomplete ( )

Replaces the input string with the current top auto-complete choice.

Definition at line 302 of file Console.cpp.

2.12.3.15 virtual void Console::Execute (String *input*) `[pure virtual]`

Process the current input string, work whatever magic is appropriate.

Pure virtual function; must be implemented in the subclass.

Parameters

input	The string to process in the Console

Implemented in TestConsole, and LuaConsole.

2.12.3.16 virtual StringList Console::GetCompletions (const String & *input*) `[pure virtual]`

Gets a set of strings that are potential auto-complete matches for the current input.

Pure virtual function; must be implemented in the subclass.

Parameters

input	The input string to try and match.

Returns

The list of potential matches

Implemented in TestConsole, and LuaConsole.

The documentation for this class was generated from the following files:

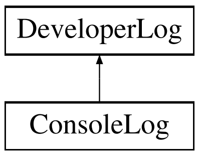
- Infrastructure/Console.h
- Infrastructure/Console.cpp

2.13 ConsoleLog Class Reference

A log that writes to the current Console.

`#include <Log.h>`

Inheritance diagram for ConsoleLog:



Public Member Functions

- virtual void Log (const String &val)

2.13.1 Detailed Description

This type of Log will append any text you give it to the Console's log area (visible by toggling the Console on [the '∼' key by default]).

Definition at line 73 of file Log.h.

2.13.2 Member Function Documentation

2.13.2.1 void ConsoleLog::Log (const String & *val*) `[virtual]`

Writes the string to the Console

Parameters

val	The string to be logged

Implements DeveloperLog.

Definition at line 103 of file Log.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/Log.h
- Infrastructure/Log.cpp

2.14 Controller Class Reference

A class representing a gamepad controller.

```
#include <Controller.h>
```

Public Member Functions

- void Setup ()
- void UpdateState ()
- const ControllerInput GetState ()
- const Vector2 GetLeftThumbVec2 ()
- const Vector2 GetRightThumbVec2 ()
- const Vec2i GetLeftThumbstick ()
- const Vec2i GetRightThumbstick ()
- const int GetRightTrigger ()
- const int GetLeftTrigger ()
- const bool IsButtonDown (int buttonMask)
- const bool IsAButtonDown ()
- const bool IsBButtonDown ()
- const bool IsXButtonDown ()
- const bool IsYButtonDown ()
- const bool IsLeftThumbstickButtonDown ()
- const bool IsRightThumbstickButtonDown ()
- const bool IsStartButtonDown ()
- const bool IsBackButtonDown ()
- const bool IsLeftBumperDown ()
- const bool IsRightBumperDown ()
- const bool IsLeftTriggerPressed ()
- const bool IsRightTriggerPressed ()

- void SetLeftVibrationRaw (unsigned int vibration)
- void SetRightVibrationRaw (unsigned int vibration)
- void SetLeftVibration (unsigned int vibration)
- void SetRightVibration (unsigned int vibration)
- const unsigned int GetLeftVibration ()
- const unsigned int GetRightVibration ()
- const bool IsConnected ()
- const int GetControllerID ()

Static Protected Attributes

- static Controller ∗ **s_Controller** = NULL

Friends

- class **ControllerManager**

2.14.1 Detailed Description

This class is your means of getting data from a controller. At the moment we only test and support the Xbox 360 controller over USB, but this interface could be abstracted to support other controllers.

On Windows you'll need DirectX installed to play a game made with Angel, and the DirectX SDK to develop one. A pretty big download/install just to get access to a controller, but we figure most gamers/developers have already done so anyway.

On the Mac both players and developers need to install a kernel extension (sorry!) included in the Tools directory. We'd be more than happy to remove this requirement, but unless someone who knows more about how to use IOKit and raw USB values wants to help, this is what we've got.

Definition at line 168 of file Controller.h.

2.14.2 Member Function Documentation

2.14.2.1 void Controller::Setup ( )

Sets the controller up to read data every frame. If it's successful, the Controller::IsConnected function will return true.

Definition at line 139 of file Controller.cpp.

2.14.2.2 void Controller::UpdateState ( )

Called once per frame to read the data in from the controller. Called automatically by the ControllerManager.

Definition at line 302 of file Controller.cpp.

2.14.2.3 const ControllerInput Controller::GetState ( )

Directly access all current input on the controller.

Returns

The current input data from the buttons and thumbsticks of this controller.

Definition at line 502 of file Controller.cpp.

2.14.2.4 const Vector2 Controller::GetLeftThumbVec2 ( )

Get the X and Y positions of the left thumbstick as a Vector2.

Returns

The current direction of the left thumbstick, as a Vector2 with both dimensions ranging from -1.0 to 1.0.

Definition at line 507 of file Controller.cpp.

2.14.2.5 const Vector2 Controller::GetRightThumbVec2 ( )

Get the X and Y positions of the right thumbstick as a Vector2.

Returns

The current direction of the right thumbstick, as a Vector2 with both dimensions ranging from -1.0 to 1.0.

Definition at line 515 of file Controller.cpp.

2.14.2.6 const Vec2i Controller::GetLeftThumbstick ( )

Get the raw values of the left thumbstick as an integer vector.

Returns

The current raw values of the left thumbstick, with both X and Y ranging from -32768 to 32768.

Definition at line 523 of file Controller.cpp.

2.14.2.7 const Vec2i Controller::GetRightThumbstick ( )

Get the raw values of the right thumbstick as an integer vector.

Returns

The current raw values of the right thumbstick, with both X and Y ranging from -32768 to 32768.

Definition at line 530 of file Controller.cpp.

2.14.2.8 const int Controller::GetRightTrigger ( )

Get the current value of the right trigger.

Returns

How much the right analog trigger is pressed, ranging from 0 (untouched) to 255 (all the way down).

Definition at line 538 of file Controller.cpp.

2.14.2.9 const int Controller::GetLeftTrigger ( )

Get the current value of the left trigger.

Returns

How much the left analog trigger is pressed, ranging from 0 (untouched) to 255 (all the way down).

Definition at line 543 of file Controller.cpp.

2.14.2.10 const bool Controller::IsButtonDown (int *buttonMask*)

Pass a button mask to find out if a combination of buttons is depressed at the same time. Here are the mask values:

```
#define XINPUT_GAMEPAD_DPAD_UP          0x00000001
#define XINPUT_GAMEPAD_DPAD_DOWN        0x00000002
#define XINPUT_GAMEPAD_DPAD_LEFT        0x00000004
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x00000008
#define XINPUT_GAMEPAD_START            0x00000010
#define XINPUT_GAMEPAD_BACK             0x00000020
#define XINPUT_GAMEPAD_LEFT_THUMB       0x00000040
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x00000080
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200
#define XINPUT_GAMEPAD_A                0x1000
#define XINPUT_GAMEPAD_B                0x2000
#define XINPUT_GAMEPAD_X                0x4000
#define XINPUT_GAMEPAD_Y                0x8000
```

Parameters

buttonMask	The values you care about, combined as a bitmask

Returns

True if all the masked buttons are pressed, false if they're not

Definition at line 548 of file Controller.cpp.

2.14.2.11 const bool Controller::IsAButtonDown ( )

Find out if the A button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 553 of file Controller.cpp.

2.14.2.12 const bool Controller::IsBButtonDown ( )

Find out if the B button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 565 of file Controller.cpp.

2.14.2.13 const bool Controller::IsXButtonDown ( )

Find out if the X button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 577 of file Controller.cpp.

2.14.2.14 const bool Controller::IsYButtonDown ( )

Find out if the Y button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 589 of file Controller.cpp.

2.14.2.15 const bool Controller::IsLeftThumbstickButtonDown ( )

Find out if the left thumbstick is currently pressed down.

Returns

True if it is, false if it's not.

Definition at line 601 of file Controller.cpp.

2.14.2.16 const bool Controller::IsRightThumbstickButtonDown ( )

Find out if the right thumbstick is currently pressed down.

Returns

True if it is, false if it's not.

Definition at line 613 of file Controller.cpp.

2.14.2.17 const bool Controller::IsStartButtonDown ( )

Find out if the start button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 625 of file Controller.cpp.

2.14.2.18 const bool Controller::IsBackButtonDown ( )

Find out if the back button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 637 of file Controller.cpp.

2.14.2.19 const bool Controller::IsLeftBumperDown ( )

Find out if the left shoulder button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 649 of file Controller.cpp.

2.14.2.20 const bool Controller::IsRightBumperDown ( )

Find out if the right shoulder button is currently pressed.

Returns

True if it is, false if it's not.

Definition at line 661 of file Controller.cpp.

2.14.2.21 const bool Controller::IsLeftTriggerPressed ( )

Find out if the left trigger is pressed down. Since it's an analog trigger, there's a driver-defined threshold that this function will use to determine when it can be considered "activated."

Returns

True if it is, false if it's not.

Definition at line 673 of file Controller.cpp.

2.14.2.22 const bool Controller::IsRightTriggerPressed ( )

Find out if the left trigger is pressed down. Since it's an analog trigger, there's a driver-defined threshold that this function will use to determine when it can be considered "activated."

Returns

True if it is, false if it's not.

Definition at line 685 of file Controller.cpp.

2.14.2.23 void Controller::SetLeftVibrationRaw (unsigned int *vibration*)

Set the rumble of the controller's lower frequencies. This sets the raw driver value.

Not available on Mac OS X. :-(

Parameters

frequencies	The desired intensity, ranging from 0 to 65535.

Definition at line 708 of file Controller.cpp.

2.14.2.24 void Controller::SetRightVibrationRaw (unsigned int *vibration*)

Set the rumble of the controller's higher frequencies. This sets the raw driver value.

Not available on Mac OS X. :-(

Parameters

frequencies	The desired intensity, ranging from 0 to 65535.

Definition at line 734 of file Controller.cpp.

2.14.2.25 void Controller::SetLeftVibration (unsigned int *vibration*)

Set the rumble of the controller's lower frequencies. This uses more manageable numbers if you don't feel like dealing with INT_MAX.

Not available on Mac OS X. :-(

Parameters

frequencies	The desired intensity, ranging from 0 to 255.

Definition at line 760 of file Controller.cpp.

2.14.2.26 void Controller::SetRightVibration (unsigned int *vibration*)

Set the rumble of the controller's higher frequencies. This uses more manageable numbers if you don't feel like dealing with INT_MAX.

Not available on Mac OS X. :-(

Parameters

frequencies	The desired intensity, ranging from 0 to 255.

Definition at line 773 of file Controller.cpp.

2.14.2.27 const unsigned int Controller::GetLeftVibration ( )

Get the current vibration setting for the lower frequencies.

Returns

Current intensity, ranging from 0 to 65535.

Definition at line 698 of file Controller.cpp.

2.14.2.28 const unsigned int Controller::GetRightVibration ( )

Get the current vibration setting for the higher frequencies.

Returns

Current intensity, ranging from 0 to 65535.

Definition at line 703 of file Controller.cpp.

2.14.2.29 const bool Controller::IsConnected ( ) `[inline]`

Find out if this controller object is currently mapped to a real-world controller and receiving input.

Returns

True if this controller gets input

Definition at line 419 of file Controller.h.

2.14.2.30 const int Controller::GetControllerID ( ) `[inline]`

Get the index this controller corresponds to in the ControllerManager.

Returns

The index that can be used in ControllerManager::GetController

Definition at line 426 of file Controller.h.

The documentation for this class was generated from the following files:

- Input/Controller.h
- Input/Controller.cpp

2.15 ControllerInput Struct Reference

A struct that wraps all controller input values into one unit.

`#include <Controller.h>`

Public Attributes

- int **LeftThumbstickX**
- int **LeftThumbstickY**
- int **RightThumbstickX**
- int **RightThumbstickY**
- int **LeftTriggerValue**
- int **RightTriggerValue**
- unsigned int **Buttons**

2.15.1 Detailed Description

Definition at line 57 of file Controller.h.

The documentation for this struct was generated from the following file:

- Input/Controller.h

2.16 ControllerManager Class Reference

(Internal) Manages multiple controllers (currently up to 2)

`#include <Controller.h>`

Public Member Functions

- Controller & GetController (int controllerIndex=0)
- void Setup ()
- void UpdateState ()
- ∼ControllerManager ()

Static Public Member Functions

- static ControllerManager & GetInstance ()

Static Protected Attributes

- static ControllerManager ∗ **s_ControllerManager** = NULL

2.16.1 Detailed Description

This internal class represents the coordinator for gamepad controllers. You should only have to deal with it if you're using it with multiple controllers, and even then, it's best to access them with the "controllerOne" and "controllerTwo" shortcuts.

Like the World, it uses the singleton pattern; you can't actually declare a new instance of a ControllerManager. To access controllers in your world, use "theControllerManager" to retrieve the singleton object. "theController-Manager" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Definition at line 91 of file Controller.h.

2.16.2 Constructor & Destructor Documentation

2.16.2.1 ControllerManager::∼ControllerManager ( )

The destructor releases all the controller input handles.

Definition at line 106 of file Controller.cpp.

2.16.3 Member Function Documentation

2.16.3.1 ControllerManager & ControllerManager::GetInstance ( ) [static]

Used to access the singleton instance of this class. As a shortcut, you can just use "theControllerManager".

Returns

The singleton

Definition at line 88 of file Controller.cpp.

2.16.3.2 Controller& ControllerManager::GetController (int *controllerIndex* = 0) [inline]

Get a reference to one of the controllers.

Parameters

controllerIndex	The index of the controller you want to retrieve. Since we only support 2 controllers, this needs to be either 0 or 1.

Returns

A Controller object

Definition at line 110 of file Controller.h.

2.16.3.3 void ControllerManager::Setup ( )

Attempt to initialize both controllers.

Definition at line 114 of file Controller.cpp.

2.16.3.4 void ControllerManager::UpdateState ( )

Called once per frame, reads all input data from the controller for polling by the rest of the game.

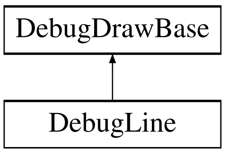
Definition at line 120 of file Controller.cpp.

The documentation for this class was generated from the following files:

- Input/Controller.h
- Input/Controller.cpp

2.17 DebugDrawBase Class Reference

Inheritance diagram for DebugDrawBase:



Protected Member Functions

- virtual void **Draw** ()=0
- void **SetupDraw** ()

Protected Attributes

- float **_timeRemaining**
- bool **_bPermanent**
- Color **_color**

Friends

- class **World**

2.17.1 Detailed Description

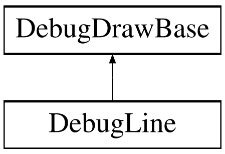
Definition at line 3 of file DebugDraw.h.

The documentation for this class was generated from the following file:

- Infrastructure/DebugDraw.h

2.18 DebugLine Class Reference

Inheritance diagram for DebugLine:



Protected Member Functions

- virtual void **Draw** ()

Protected Attributes

- float **_points** [4]

Friends

- class **World**

2.18.1 Detailed Description

Definition at line 20 of file DebugDraw.h.

The documentation for this class was generated from the following file:

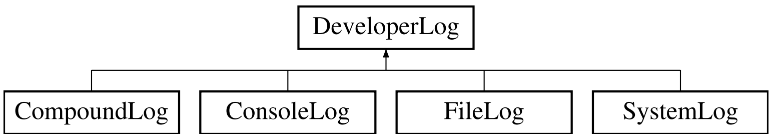
- Infrastructure/DebugDraw.h

2.19 DeveloperLog Class Reference

Abstract base class for logs.

`#include <Log.h>`

Inheritance diagram for DeveloperLog:



Public Member Functions

- virtual ∼DeveloperLog ()
- virtual void Log (const String &val)=0
- void Printf (const char ∗format,...)

2.19.1 Detailed Description

The DeveloperLog simply provides an interface for other Log classes to implement.

Definition at line 40 of file Log.h.

2.19.2 Constructor & Destructor Documentation

2.19.2.1 virtual DeveloperLog::∼DeveloperLog ( ) `[inline], [virtual]`

Virtual destructor needed in abstract base class.

Definition at line 46 of file Log.h.

2.19.3 Member Function Documentation

2.19.3.1 virtual void DeveloperLog::Log (const String & *val*) `[pure virtual]`

Logs a string.

Pure virtual function; must be implemented in the subclass.

Parameters

val	The string to be logged.

Implemented in CompoundLog, SystemLog, FileLog, and ConsoleLog.

2.19.3.2 void DeveloperLog::Printf (const char ∗ *format,* *...*)

Log a formatted string using printf syntax.

http://www.cplusplus.com/reference/clibrary/cstdio/printf.html

Parameters

format	The format string
...	The parameters to substitute into the format string

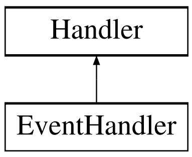
Definition at line 61 of file Log.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/Log.h
- Infrastructure/Log.cpp

2.20 EventHandler Class Reference

Inheritance diagram for EventHandler:



Public Member Functions

- void **AddButtonCallback** (Gwen::Controls::Base ∗control, void(∗callback)())
- void **AddChoiceCallback** (Gwen::Controls::Base ∗control, void(∗callback)(int))
- void **OnPress** (Gwen::Controls::Base ∗control)

2.20.1 Detailed Description

Definition at line 59 of file UserInterface.cpp.

The documentation for this class was generated from the following file:

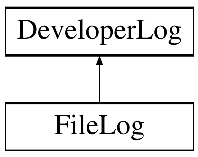
- UI/UserInterface.cpp

2.21 FileLog Class Reference

A log which writes to a file on disk.

```
#include <Log.h>
```

Inheritance diagram for FileLog:



Public Member Functions

- FileLog (const String &fileName)
- virtual void Log (const String &val)

Static Public Member Functions

- static String MakeLogFileName (const String &fileName)

2.21.1 Detailed Description

This type of Log appends its text to a specified file in the Logs directory.

Definition at line 88 of file Log.h.

2.21.2 Constructor & Destructor Documentation

2.21.2.1 FileLog::FileLog (const String & *fileName*)

The constructor takes a filename (which can be generated from FileLog::MakeLogFileName). **NB:** If the file already exists, it will be cleared. The file will get a timestamp at the top saying when it was first opened.

Parameters

fileName	

Definition at line 128 of file Log.cpp.

2.21.3 Member Function Documentation

2.21.3.1 String FileLog::MakeLogFileName (const String & *fileName*) [static]

Gives the relative path to the log file from a desired name.

Parameters

fileName	The desired name (for example: "StartupLog")

Returns

The path to the file (for example: "Logs/StartupLog.log"). This path is relative to the executable.

Definition at line 113 of file Log.cpp.

2.21.3.2 void FileLog::Log (const String & *val*) [virtual]

The string to be logged in the file.

Parameters

val	The string to put in the file

Implements DeveloperLog.

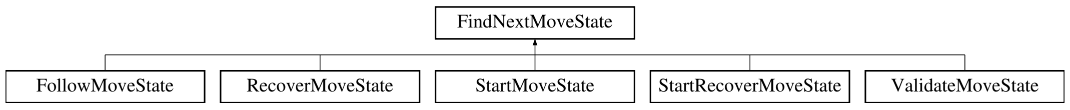
Definition at line 140 of file Log.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/Log.h
- Infrastructure/Log.cpp

2.22 FindNextMoveState Class Reference

Inheritance diagram for FindNextMoveState:



Public Member Functions

- virtual void **Initialize** (PathFinder ∗pathFinder)
- virtual bool **Update** (PathFinderMove &move)=0
- virtual void **BeginState** (PathFinder::ePFMoveState)
- virtual void **EndState** (PathFinder::ePFMoveState)
- virtual const char ∗ **GetName** ()=0

Protected Member Functions

- void **SetNewState** (PathFinder::ePFMoveState newState)
- Vector2List & **GetCurrentPath** ()
- const Vector2 & **GetCurrentPosition** ()
- const Vector2 & **GetCurrentDestination** ()
- int & **GetCurrentPathIndex** ()
- float **GetCurrentArrivalDist** ()

Protected Attributes

- PathFinder ∗ **_pathFinder**

2.22.1 Detailed Description

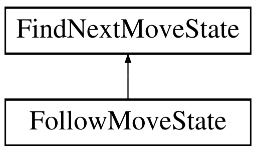
Definition at line 87 of file PathFinder.h.

The documentation for this class was generated from the following files:

- AI/PathFinder.h
- AI/PathFinder.cpp

2.23 FollowMoveState Class Reference

Inheritance diagram for FollowMoveState:



Public Member Functions

- virtual const char ∗ **GetName** ()
- virtual bool **Update** (PathFinderMove &move)

Additional Inherited Members

2.23.1 Detailed Description

Definition at line 186 of file PathFinder.cpp.

The documentation for this class was generated from the following file:

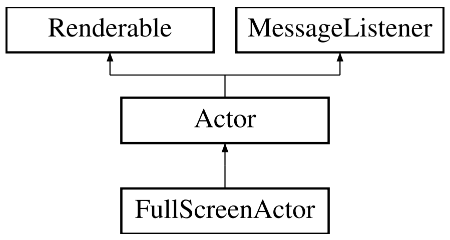
- AI/PathFinder.cpp

2.24 FullScreenActor Class Reference

An Actor which takes up the whole drawing space.

```
#include <FullScreenActor.h>
```

Inheritance diagram for FullScreenActor:



Public Member Functions

- FullScreenActor ()
- void SetLock (bool locked)
- const bool IsLocked ()
- virtual void ReceiveMessage (Message ∗message)
- virtual const String GetClassName () const

Additional Inherited Members

2.24.1 Detailed Description

A FullScreenActor will resize itself whenever the size of the drawing space changes. This makes it useful for backdrops, curtains, splash screens, etc.

Definition at line 39 of file FullScreenActor.h.

2.24.2 Constructor & Destructor Documentation

2.24.2.1 FullScreenActor::FullScreenActor ( )

The constructor subscribes the Actor to CameraChange messages and does the initial orientation to the camera.

Definition at line 36 of file FullScreenActor.cpp.

2.24.3 Member Function Documentation

2.24.3.1 void FullScreenActor::SetLock (bool *locked*)

You can lock a FullScreenActor to keep it from tracking the changes to the drawing area or camera. When you unlock it, it will snap back to taking up the whole screen.

Parameters

locked	whether or not the FullScreenActor should continue to track camera changes

Definition at line 43 of file FullScreenActor.cpp.

2.24.3.2 const bool FullScreenActor::IsLocked ( )

To check on the lock status.

Returns

whether or not this FullScreenActor is currently tracking the camera

Definition at line 52 of file FullScreenActor.cpp.

2.24.3.3 void FullScreenActor::ReceiveMessage (Message $*$ *message*) [virtual]

An implementation of the MessageListener interface, which listens for CameraChange messages and responds appropriately.

See Also

MessageListener

Parameters

message	The message getting delivered.

Reimplemented from Actor.

Definition at line 57 of file FullScreenActor.cpp.

2.24.3.4 virtual const String FullScreenActor::GetClassName ( ) const [inline], [virtual]

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "FullScreenActor"

Reimplemented from Actor.

Definition at line 80 of file FullScreenActor.h.

The documentation for this class was generated from the following files:

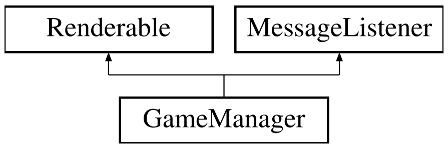
- Actors/FullScreenActor.h
- Actors/FullScreenActor.cpp

2.25 GameManager Class Reference

A class to oversee the high-level aspects of your game.

`#include <GameManager.h>`

Inheritance diagram for GameManager:



Public Member Functions

- GameManager ()
- ∼GameManager ()
- virtual void Render ()
- virtual void Update (float dt)
- virtual bool IsProtectedFromUnloadAll (Renderable ∗renderable)
- virtual void ReceiveMessage (Message ∗message)
- virtual void SoundEnded (AngelSoundHandle sound)

Additional Inherited Members

2.25.1 Detailed Description

Oftentimes you want to have a central class that handles game-flow. Things like keeping score, managing the creation of new Actors, etc. You could do this with a centralized Actor that doesn't get drawn, but there's a lot of functionality (baggage) on the Actor class that you wouldn't be using in that case.

The GameManager class is a Renderable, so it will get all the Render and Update calls that an Actor would receive. You register it with your World (via the `theWorld.SetGameManager` function) and can then retrieve it from anywhere in your game.

Definition at line 49 of file GameManager.h.

2.25.2 Constructor & Destructor Documentation

2.25.2.1 GameManager::GameManager ( ) `[inline]`

The default constructor is empty; we expect you to subclass to do anything meaningful.

Definition at line 56 of file GameManager.h.

2.25.2.2 GameManager::∼GameManager ( )

The destructor makes sure that the GameManager unsubscribes from all Message notifications.

Definition at line 43 of file GameManager.cpp.

2.25.3 Member Function Documentation

2.25.3.1 void GameManager::Render ( ) `[virtual]`

Override of the Renderable::Render function. Doesn't do anything in this base class.

Reimplemented from Renderable.

Definition at line 39 of file GameManager.cpp.

2.25.3.2 void GameManager::Update (float *dt*) [virtual]

Override of the Renderable::Update function. Also doesn't do anything in the base class, but in the future it might, so make sure you call this in any override implementations.

Parameters

dt	The amount of time elapsed since the start of the last frame.

Reimplemented from Renderable.

Definition at line 35 of file GameManager.cpp.

2.25.3.3 virtual bool GameManager::IsProtectedFromUnloadAll (Renderable ∗ *renderable*) [inline], [virtual]

In your GameManager, you can define a custom function to determine whether specific Renderables should be protected from the World::UnloadAll function. This function will get called for each Renderable during an UnloadAll attempt.

Parameters

renderable	The Renderable to test

Returns

Whether it should be protected (returning false means it will be removed from the world)

Definition at line 89 of file GameManager.h.

2.25.3.4 virtual void GameManager::ReceiveMessage (Message ∗ *message*) [inline], [virtual]

Override of the MessageListener::ReceiveMessage function. Does nothing in this base class.

Parameters

message	The received message.

Implements MessageListener.

Definition at line 97 of file GameManager.h.

2.25.3.5 virtual void GameManager::SoundEnded (AngelSoundHandle *sound*) [inline], [virtual]

The sound system needs a callback that gets triggered when any sound ends. Since the GameManager is a logical place to have sound control functionality, it has this function for you to implement.

Parameters

sound	The handle to the sound that has just completed.

Definition at line 106 of file GameManager.h.

The documentation for this class was generated from the following files:

- Infrastructure/GameManager.h
- Infrastructure/GameManager.cpp

2.26 GestureData Struct Reference

```
#include <MultiTouch.h>
```

Public Attributes

- float Velocity
- float GestureMagnitude

2.26.1 Detailed Description

Structure which bundles the data about individual gestures that are received and sent via messages.

Definition at line 67 of file MultiTouch.h.

2.26.2 Member Data Documentation

2.26.2.1 float GestureData::Velocity

The "Velocity" of the gesture (only recorded on the actual iOS hardware.)

Definition at line 72 of file MultiTouch.h.

2.26.2.2 float GestureData::GestureMagnitude

In a rotation gesture, this will be the angle in radians. For a pinch, it will be the scale.

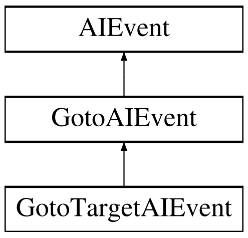
Definition at line 78 of file MultiTouch.h.

The documentation for this struct was generated from the following file:

- Input/MultiTouch.h

2.27 GotoAIEvent Class Reference

Inheritance diagram for GotoAIEvent:



Public Member Functions

- virtual GotoAIEvent ∗ **Initialize** (const Vector2 &destination, float moveSpeed, float arrivalDist=0.2f)
- virtual void **Update** (float dt)

Protected Member Functions

- PathFinder & **GetPathfinder** ()

Protected Attributes

- bool **_moveFailed**
- Vector2 **_destination**
- float **_moveSpeed**
- float **_arrivalDist**

2.27.1 Detailed Description

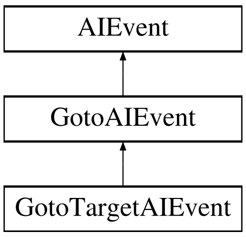
Definition at line 37 of file GotoAIEvent.h.

The documentation for this class was generated from the following files:

- AIEvents/GotoAIEvent.h
- AIEvents/GotoAIEvent.cpp

2.28 GotoTargetAIEvent Class Reference

Inheritance diagram for GotoTargetAIEvent:



Public Member Functions

- virtual GotoTargetAIEvent ∗ **Initialize** (const String &targetTag, float moveSpeed, float arrivalDist=0.2f)
- virtual void **Update** (float dt)

Protected Attributes

- String **_targetTag**

Additional Inherited Members

2.28.1 Detailed Description

Definition at line 35 of file GotoTargetAIEvent.h.

The documentation for this class was generated from the following files:

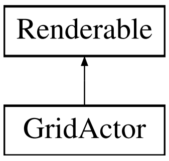
- AIEvents/GotoTargetAIEvent.h
- AIEvents/GotoTargetAIEvent.cpp

2.29 GridActor Class Reference

An Actor to draw lines at regular intervals.

```
#include <GridActor.h>
```

Inheritance diagram for GridActor:



Public Member Functions

- GridActor ()
- GridActor (const Color &lines, const Color &axis, float interval, const Vector2 &minCoord, const Vector2 &maxCoord)
- void SetLineColor (const Color &lineCol)
- const Color & GetLineColor () const
- void SetAxisColor (const Color &axisCol)
- const Color & GetAxisColor () const
- void SetInterval (float interval)
- const float GetInterval () const
- void SetMinCoord (const Vector2 &minCoord)
- const Vector2 GetMinCoord () const
- void SetMaxCoord (const Vector2 &maxCoord)
- const Vector2 GetMaxCoord () const
- virtual void Render ()
- virtual void Update (float dt)

Additional Inherited Members

2.29.1 Detailed Description

A GridActor just makes it easy to draw grids in your game. This can be useful for placing objects while you're creating the world, or to just look cool. :-)

Definition at line 41 of file GridActor.h.

2.29.2 Constructor & Destructor Documentation

2.29.2.1 GridActor::GridActor ( )

The default constructor creates the "Angel grid." An interval of 1.0 GL units, drawing from (-100, -100) to (100, 100), with light blue lines and red axis lines.

Definition at line 35 of file GridActor.cpp.

2.29.2.2 GridActor::GridActor (const Color & *lines*, const Color & *axis*, float *interval*, const Vector2 & *minCoord*, const Vector2 & *maxCoord*)

If you have something else in mind for your grid, you can specify its visual appearance with this constructor.

Parameters

lines	the color of the lines in the grid
axis	the color of the lines drawn at the X and Y axes
interval	the amount of space (in GL units) between each line
minCoord	the bottom-left coordinate from which to start drawing
maxCoord	the top-right coordinate at which to stop drawing

Definition at line 47 of file GridActor.cpp.

2.29.3 Member Function Documentation

2.29.3.1 void GridActor::SetLineColor (const Color & *lineCol*)

Set the color of the non-axis lines of the grid.

Parameters

lineCol	the color of the lines

Definition at line 58 of file GridActor.cpp.

2.29.3.2 const Color & GridActor::GetLineColor ( ) const

Return the current line color of this GridActor.

Returns

the current line color as a Color object

Definition at line 63 of file GridActor.cpp.

2.29.3.3 void GridActor::SetAxisColor (const Color & *axisCol*)

Set the color of the axis lines of the grid.

Parameters

axisCol	the color of the axis lines

Definition at line 68 of file GridActor.cpp.

2.29.3.4 const Color & GridActor::GetAxisColor ( ) const

Return the current axis line color of this GridActor.

Returns

the current axis line color as a Color object

Definition at line 73 of file GridActor.cpp.

2.29.3.5 void GridActor::SetInterval (float *interval*)

Set the spacing of the lines.

Parameters

interval	the amount of space between each line in GL units

Definition at line 78 of file GridActor.cpp.

2.29.3.6 const float GridActor::GetInterval ( ) const

Return the line spacing interval for this GridActor.

Returns

the current interval in GL units

Definition at line 83 of file GridActor.cpp.

2.29.3.7 void GridActor::SetMinCoord (const Vector2 & *minCoord*)

Set the bottom-left coordinate at which to start drawing the grid.

Parameters

minCoord	the bottom-left X and Y position in GL units

Definition at line 88 of file GridActor.cpp.

2.29.3.8 const Vector2 GridActor::GetMinCoord ( ) const

Return the bottom-left start coordinate of the grid.

Returns

the bottom-left X and Y position in GL unites

Definition at line 94 of file GridActor.cpp.

2.29.3.9 void GridActor::SetMaxCoord (const Vector2 & *maxCoord*)

Set the top-right coordinate at which to stop drawing the grid.

Parameters

maxCoord	the top-right X and Y position in GL units

Definition at line 99 of file GridActor.cpp.

2.29.3.10 const Vector2 GridActor::GetMaxCoord ( ) const

Return the top-right coordinate of the grid.

Returns

the top-right X and Y position in GL units

Definition at line 105 of file GridActor.cpp.

2.29.3.11 void GridActor::Render ( ) `[virtual]`

Override of the normal Renderable::Render function. Draws the lines that have been specified.

Reimplemented from Renderable.

Definition at line 140 of file GridActor.cpp.

2.29.3.12 virtual void GridActor::Update (float *dt*) `[inline]`, `[virtual]`

Override of the normal Renderable::Update function. Does nothing, here to satisfy the abstract base class.

Reimplemented from Renderable.

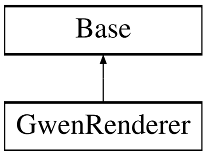
Definition at line 143 of file GridActor.h.

The documentation for this class was generated from the following files:

- Actors/GridActor.h
- Actors/GridActor.cpp

2.30 GwenRenderer Class Reference

Inheritance diagram for GwenRenderer:



Public Member Functions

- **GwenRenderer** (const String &texturePath)
- virtual void **FinishInit** ()
- virtual void **Begin** ()
- virtual void **End** ()
- virtual void **SetDrawColor** (Gwen::Color color)
- virtual void **DrawFilledRect** (Gwen::Rect rect)
- virtual void **StartClip** ()
- virtual void **EndClip** ()
- virtual void **LoadTexture** (Gwen::Texture ∗pTexture)
- virtual void **FreeTexture** (Gwen::Texture ∗pTexture)
- virtual void **DrawTexturedRect** (Gwen::Texture ∗pTexture, Gwen::Rect pTargetRect, float u1=0.0f, float v1=0.0f, float u2=1.0f, float v2=1.0f)
- virtual Gwen::Color **PixelColour** (Gwen::Texture ∗pTexture, unsigned int x, unsigned int y, const Gwen::Color &col_default=Gwen::Color(255, 255, 255, 255))
- virtual void **LoadFont** (Gwen::Font ∗pFont)
- virtual void **FreeFont** (Gwen::Font ∗pFont)
- virtual void **RenderText** (Gwen::Font ∗pFont, Gwen::Point pos, const Gwen::UnicodeString &text)
- virtual Gwen::Point **MeasureText** (Gwen::Font ∗pFont, const Gwen::UnicodeString &text)

2.30.1 Detailed Description

Definition at line 37 of file GwenRenderer.h.

The documentation for this class was generated from the following files:

- UI/GwenRenderer.h
- UI/GwenRenderer.cpp

2.31 AStarSearch< UserState >::HeapCompare_f Class Reference

Public Member Functions

- bool **operator()** (const Node ∗x, const Node ∗y) const

2.31.1 Detailed Description

template<class UserState>class AStarSearch< UserState >::HeapCompare_f

Definition at line 101 of file stlastar.h.

The documentation for this class was generated from the following file:

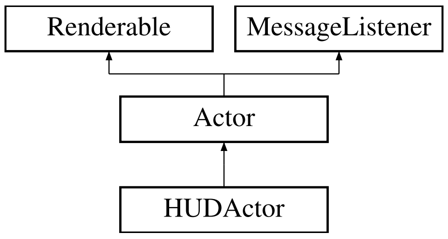
- AI/stlastar.h

2.32 HUDActor Class Reference

An Actor that gets drawn in screen-space.

`#include <HUDActor.h>`

Inheritance diagram for HUDActor:



Public Member Functions

- virtual void Render ()
- virtual const String GetClassName () const

Additional Inherited Members

2.32.1 Detailed Description

A HUDActor is drawn in screen-space rather than world-space. You can treat it just like any other normal Actor, but both its position and size are described in pixels instead of GL units. The screenspace coordinate system starts at the top-left of the window.

For example, to have an actor be 100 pixels wide and situated in the bottom-right quadrant of the window (assuming the default window size of 1024x768):

```
HUDActor *h = new HUDActor();
h->SetSize(100.0f);
h->SetPosition(974.0f, 718.0f);
theWorld.Add(h);
```

Definition at line 51 of file HUDActor.h.

2.32.2 Member Function Documentation

2.32.2.1 void HUDActor::Render ( ) [virtual]

Override of the Renderable::Render function to handle drawing in screen-space.

Reimplemented from Actor.

Definition at line 38 of file HUDActor.cpp.

2.32.2.2 virtual const String HUDActor::GetClassName ( ) const [inline], [virtual]

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "HUDActor"

Reimplemented from Actor.

Definition at line 66 of file HUDActor.h.

The documentation for this class was generated from the following files:

- Actors/HUDActor.h
- Actors/HUDActor.cpp

2.33 InputBinding Class Reference

(Internal) Handles the binding of keypresses to Messages

```
#include <InputManager.h>
```

Public Member Functions

- void **SetKeyDownMessage** (const String &keyDownMessage)
- void **SetKeyUpMessage** (const String &keyUpMessage)
- void **OnKeyDown** ()
- void **OnKeyUp** ()

2.33.1 Detailed Description

An internal class used by the InputManager, mapping keydown and keyup events to the appropriate Messages.

Definition at line 40 of file InputManager.h.

The documentation for this class was generated from the following files:

- Input/InputManager.h
- Input/InputManager.cpp

2.34 InputManager Class Reference

(Internal) Handles keyboard input, and mapping controller input to messages

```
#include <InputManager.h>
```

Public Member Functions

- void **BindKey** (const String &keyId, const String &command)
- void **UnbindKey** (const String &keyId)
- bool **OnKeyDown** (int keyVal)
- bool **OnKeyUp** (int keyVal)
- bool IsKeyDown (int keyVal)
- void **HandleControl** (class Controller &controller)

Static Public Member Functions

- static InputManager & **GetInstance** ()
- static void **Destroy** ()

Protected Member Functions

- void **Initialize** ()

2.34.1 Detailed Description

This internal class is used by the engine to take the settings from input_bindings.ini and process all keyboard and controller data.

Unless you're working on the engine itself, the only function in here that matters to you would be InputManager::IsKeyDown.

Definition at line 73 of file InputManager.h.

2.34.2 Member Function Documentation

2.34.2.1 bool InputManager::IsKeyDown (int *keyVal*)

Find out whether a key is currently pressed. Can either be passed an individual char or any of the defined values in GL/glfw.h.

Parameters

keyVal	The key value to test for (usually a char)

Returns

Whether the user is currently pressing that key.

Definition at line 333 of file InputManager.cpp.

The documentation for this class was generated from the following files:

- Input/InputManager.h
- Input/InputManager.cpp

2.35 Interval< T > Class Template Reference

Simplifies lerping Actor properties across a timeframe.

```
#include <Interval.h>
```

Public Member Functions

- Interval (T start, T end, float duration, bool smooth=false)
- Interval ()
- T Step (float dt)
- float GetCurrent ()
- bool ShouldStep ()

2.35.1 Detailed Description

template<class T>class Interval< T >

This is a helper template class that manages Actor properties during lerp transitions. The following functions show how they are setup, and the Actor::Update function shows how to update and get the next values.

See Also

Actor::MoveTo
Actor::RotateTo
Actor::ChangeColorTo
Actor::ChangeSizeTo

You can use Intervals for ints, floats, and any class that has defined subtraction and division (by a float) operators that return members of that same class. Our Color, Vector2, and Vector3 classes are all able to be used in Intervals.

Definition at line 51 of file Interval.h.

2.35.2 Constructor & Destructor Documentation

2.35.2.1 template<class T> Interval< T >::Interval (T *start*, T *end*, float *duration*, bool *smooth* = `false`) `[inline]`

Setting up the interval requires a starting value, a target ending state, and how long it should take to get there.

Parameters

start	The starting value
end	The ending value
duration	Length of the transition in seconds
smooth	Whether to use a smooth interpolation with ease-in and ease-out instead of a simple linear interpolation.

Definition at line 65 of file Interval.h.

2.35.2.2 template<class T> Interval< T >::Interval ( ) `[inline]`

The default constructor just sets up an Interval that does nothing. It's here because some compilers want it.

Definition at line 80 of file Interval.h.

2.35.3 Member Function Documentation

2.35.3.1 template<class T> T Interval< T >::Step (float *dt*) `[inline]`

The step function processes the appropriate change and shuts the interval down if it's finished.

Parameters

dt	How much time has elapsed since the last Step (so how much the Interval should be incremented)

Returns

The current value of the Interval after the Step has been completed.

Definition at line 94 of file Interval.h.

2.35.3.2 template<class T> float Interval< T >::GetCurrent ( ) `[inline]`

Get the current value without Stepping the Interval

Returns

The current value

Definition at line 131 of file Interval.h.

2.35.3.3 template<class T> bool Interval< T >::ShouldStep ( ) [inline]

Whether or not the Interval still has changes to process. If this returns false, the Interval is done and you can discard it.

Returns

Definition at line 142 of file Interval.h.

The documentation for this class was generated from the following file:

- Infrastructure/Interval.h

2.36 LoadedVariable Struct Reference

An internal structure used by the preferences and tuning system.

```
#include <LoadedVariable.h>
```

Public Member Functions

- **LoadedVariable** (int val)
- **LoadedVariable** (float val)
- **LoadedVariable** (String val)
- **LoadedVariable** (Vector2 val)

Public Attributes

- int **_int**
- float **_float**
- String **_string**
- Vector2 **_vector**
- int **_setAs**

2.36.1 Detailed Description

Definition at line 36 of file LoadedVariable.h.

The documentation for this struct was generated from the following file:

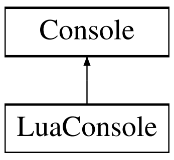
- Infrastructure/LoadedVariable.h

2.37 LuaConsole Class Reference

Console to process Lua input during the game.

```
#include <LuaConsole.h>
```

Inheritance diagram for LuaConsole:



Public Member Functions

- LuaConsole ()
- virtual void Execute (String input)
- virtual StringList GetCompletions (const String &input)

Additional Inherited Members

2.37.1 Detailed Description

An implementation of the Console class that runs input strings of Lua code, and prints output from the Lua runtime.

Definition at line 39 of file LuaConsole.h.

2.37.2 Constructor & Destructor Documentation

2.37.2.1 LuaConsole::LuaConsole ( )

Basic constructor; sets up our prompts and prints the welcome message.

Definition at line 36 of file LuaConsole.cpp.

2.37.3 Member Function Documentation

2.37.3.1 void LuaConsole::Execute (String *input*) [virtual]

Attempts to execute the given string as Lua. Any errors will be output the console.

Parameters

input	the string to execute

Implements Console.

Definition at line 52 of file LuaConsole.cpp.

2.37.3.2 StringList LuaConsole::GetCompletions (const String & *input*) [virtual]

Gets potential autocompletions from the Lua runtime given a starting string.

Parameters

input	the beginning of the string to try to complete

Returns

a list of potential matches

Implements Console.

Definition at line 111 of file LuaConsole.cpp.

The documentation for this class was generated from the following files:

- Scripting/LuaConsole.h
- Scripting/LuaConsole.cpp

2.38 LuaScriptingModule Class Reference

(Internal) Handles all interactions with our Lua layer

`#include <LuaModule.h>`

Static Public Member Functions

- static void Prep ()
- static void Initialize ()
- static void Finalize ()
- static void ExecuteInScript (const String &code)
- static lua_State ∗ GetLuaState ()
- static void DumpStack ()

2.38.1 Detailed Description

This internal class is the glue between the compiled Angel simulation engine and the scripting layer generated by SWIG.

The World class handles all interactions for setup and teardown; the only bit you should care about for your game code is the ExecuteInScript function.

For more information about Lua, visit `http://www.lua.org`

Definition at line 73 of file LuaModule.h.

2.38.2 Member Function Documentation

2.38.2.1 void LuaScriptingModule::Prep ( ) `[static]`

Does the very initial stages of setting up Lua so that it can be used to load the preferences files.

Called by the World during setup.

Definition at line 52 of file LuaModule.cpp.

2.38.2.2 void LuaScriptingModule::Initialize ( ) `[static]`

Work the magic to set up Lua and load our extension module.

Note that the start.lua file located in Scripting/EngineScripts will get executed when this function is called. It's copied to the Resources/Scripts directory of all games as they're built.

Called by the World during setup.

Definition at line 78 of file LuaModule.cpp.

2.38.2.3 void LuaScriptingModule::Finalize ( ) `[static]`

Closes down Lua. Called by the World when it's destroyed.

Definition at line 100 of file LuaModule.cpp.

2.38.2.4 void LuaScriptingModule::ExecuteInScript (const String & *code*) `[static]`

Execute a string in the Lua interpreter. Can be dangerous, but can also make generating new interactions a lot easier. If you're using this function, just be careful about what you pass in.

Parameters

code	The code to execute.

Definition at line 110 of file LuaModule.cpp.

2.38.2.5 lua_State ∗ LuaScriptingModule::GetLuaState ( ) `[static]`

Get the internal Lua state that all our script code is using to run.

Returns

The a pointer to the lua_State object at the heart of things.

Definition at line 155 of file LuaModule.cpp.

2.38.2.6 void LuaScriptingModule::DumpStack ( ) `[static]`

Print information about the current Lua stack to the system log. Useful for debugging any custom Lua integration.

Definition at line 160 of file LuaModule.cpp.

The documentation for this class was generated from the following files:

- Scripting/LuaModule.h
- Scripting/LuaModule.cpp

2.39 MathUtil Class Reference

A set of static functions that handle typical math needed for games.

`#include <MathUtil.h>`

Public Types

- enum **AABBSplittingAxis** { **AA_X**, **AA_Y** }

Static Public Member Functions

- template<typename T >
  static T Abs (T val)
- template<typename T >
  static T Max (T value1, T value2)
- template<typename T >
  static T Min (T value1, T value2)
- template<typename T >
  static T Distance (T value1, T value2)
- template<typename T >
  static T Lerp (T value1, T value2, float amount)
- template<typename T >
  static T SmoothStep (T value1, T value2, float amount)
- static int Clamp (int value, int min, int max)
- static float Clamp (float value, float min, float max)
- static double Clamp (double value, double min, double max)
- static float ToDegrees (float radians)
- static float ToRadians (float degrees)
- static Vector2 VectorFromAngle (float angle_in_degrees)
- static float AngleFromVector (const Vector2 &v1)
- static float AngleFromVectors (const Vector2 &v1, const Vector2 &v2)
- static int RoundToInt (double x)
- static int RandomInt (int maximum)

- static int RandomIntInRange (int min, int max)
- static int RandomIntWithError (int target, int error)
- static float RandomFloat (float maximum=1.0f)
- static float RandomFloatInRange (float min, float max)
- static float RandomFloatWithError (float target, float error)
- static bool RandomBool ()
- static Vector2 RandomVector ()
- static Vector2 RandomVector (const Vector2 &maxValues)
- static Vector2 RandomVector (const Vector2 &minValues, const Vector2 &maxValues)
- static Vector2List RandomPointField (int numPoints, const Vector2 &minValue, const Vector2 &maxValue, float minDistance=0.5f)
- static bool FuzzyEquals (float value1, float value2, float epsilon=Epsilon)
- static bool FuzzyEquals (const Vector2 &v1, const Vector2 &v2, float epsilon=Epsilon)
- static Vector2 ScreenToWorld (int x, int y)
- static Vector2 ScreenToWorld (const Vec2i &screenCoordinates)
- static Vector2 WorldToScreen (float x, float y)
- static Vector2 WorldToScreen (const Vector2 &worldCoordinates)
- static Vector2 GetWorldDimensions ()
- static float PixelsToWorldUnits (float pixels)
- static float WorldUnitsToPixels (float worldUnits)
- static AABBSplittingAxis GetMajorAxis (const BoundingBox &source)
- static void SplitBoundingBox (const BoundingBox &source, AABBSplittingAxis axis, BoundingBox &LHS, BoundingBox &RHS)
- static float DeltaAngle (float A1, float A2)
- static float VectorDeltaAngle (const Vector2 &v1, const Vector2 &v2)

Static Public Attributes

- static const float E = 2.718282f
- static const float Log10E = 0.4342945f
- static const float Log2E = 1.442695f
- static const float Pi = 3.141593f
- static const float PiOver2 = 1.570796f
- static const float PiOver4 = 0.7853982f
- static const float TwoPi = 6.283185f
- static const float MaxFloat = 3.402823E+38f
- static const float MinFloat = -3.402823E+38f
- static const float Epsilon = 0.000001f

2.39.1 Detailed Description

This class is just a wrapper around a whole bunch of otherwise-loose functions that handle all kinds of math that is typically required in a game.

It also contains a set of constants so you don't have to look them up or declare them yourself.

Definition at line 45 of file MathUtil.h.

2.39.2 Member Function Documentation

2.39.2.1 template<typename T > static T MathUtil::Abs (T *val*) [inline], [static]

A templated absolute value function that can handle any class where a comparison against 0 makes sense (floats and ints, mostly).

Parameters

val	The number

Returns

The absolute value of that number

Definition at line 108 of file MathUtil.h.

2.39.2.2 template<typename T > static T MathUtil::Max (T *value1*, T *value2*) `[inline]`, `[static]`

A templated max function that returns the greater of two values. Works for any class that implements a > operator.

Parameters

value1	The first value
value2	The second value

Returns

The greater of the two

Definition at line 122 of file MathUtil.h.

2.39.2.3 template<typename T > static T MathUtil::Min (T *value1*, T *value2*) `[inline]`, `[static]`

A templated min function that returns the lesser of two values. Works for any class that implements a < operator.

Parameters

value1	The first value
value2	The second value

Returns

The lesser of the two

Definition at line 136 of file MathUtil.h.

2.39.2.4 template<typename T > static T MathUtil::Distance (T *value1*, T *value2*) `[inline]`, `[static]`

Find the distance between two values. Works for any class that has a subtraction operator and will work with the Abs function.

Parameters

value1	The first value
value2	The second value

Returns

The distance between them

Definition at line 150 of file MathUtil.h.

2.39.2.5 template<typename T > static T MathUtil::Lerp (T *value1*, T *value2*, float *amount*) `[inline]`, `[static]`

Linearly interpolates between two values. Works for any classes that define addition, subtraction, and multiplication (by a float) operators.

http://en.wikipedia.org/wiki/Lerp_(computing)

Parameters

value1	The starting value
value2	The ending value
amount	The amount to lerp (from 0.0 to 1.0)

Returns

The interpolated value

Definition at line 168 of file MathUtil.h.

2.39.2.6 template<typename T > static T MathUtil::SmoothStep (T *value1,* T *value2,* float *amount*) `[inline]`, `[static]`

Smoothly step between two values. Works for any classes that Lerp would work for (and is essentially a drop-in replacement). Often looks visually better than a simple linear interpolation as it gives ease-in and ease-out.

http://www.fundza.com/rman_shaders/smoothstep/index.html

Parameters

value1	The starting value
value2	The ending value
amount	The amount to interpolate (from 0.0 to 1.0)

Returns

The interpolated value

Definition at line 187 of file MathUtil.h.

2.39.2.7 static int MathUtil::Clamp (int *value,* int *min,* int *max*) `[inline]`, `[static]`

Clamps an integer to a specified range

Parameters

value	The integer in question
min	The minimum of the range
max	The maximum of the range

Returns

The clamped value

Definition at line 201 of file MathUtil.h.

2.39.2.8 static float MathUtil::Clamp (float *value,* float *min,* float *max*) `[inline]`, `[static]`

Clamps a float to a specified range

Parameters

value	The float in question
min	The minimum of the range
max	The maximum of the range

Returns

The clamped value

Definition at line 214 of file MathUtil.h.

2.39.2.9 static double MathUtil::Clamp (double *value,* double *min,* double *max*) `[inline], [static]`

Clamps a double-precision float to a specified range

Parameters

value	The double in question
min	The minimum of the range
max	The maximum of the range

Returns

The clamped value

Definition at line 227 of file MathUtil.h.

2.39.2.10 float MathUtil::ToDegrees (float *radians*) `[static]`

Convert radians to degrees

Parameters

radians	The angle in radians

Returns

The angle in degrees

Definition at line 51 of file MathUtil.cpp.

2.39.2.11 float MathUtil::ToRadians (float *degrees*) `[static]`

Convert degrees to radians

Parameters

degrees	The angle in degrees

Returns

The angle in radians

Definition at line 56 of file MathUtil.cpp.

2.39.2.12 Vector2 MathUtil::VectorFromAngle (float *angle_in_degrees*) `[static]`

Get a unit-length vector which indicate a direction along the given angle relative to the screen.

Parameters

angle_in_-degrees	The angle, in degrees

Returns

A vector moving along that angle

Definition at line 61 of file MathUtil.cpp.

2.39.2.13 float MathUtil::AngleFromVector (const Vector2 & *v1*) [static]

Get an angle from a vector indicating direction

Parameters

v1	The vector direction

Returns

The vector's angle

Definition at line 66 of file MathUtil.cpp.

2.39.2.14 float MathUtil::AngleFromVectors (const Vector2 & *v1*, const Vector2 & *v2*) [static]

Get the angle between two vectors

Parameters

v1	The first vector
v2	The second vector

Returns

The angle between them in radians

Definition at line 74 of file MathUtil.cpp.

2.39.2.15 int MathUtil::RoundToInt (double *x*) [static]

Takes a double or float and removes everything after the decimal point, making it into an integer.

Parameters

x	The double or float to round

Returns

The rounded integer

Definition at line 80 of file MathUtil.cpp.

2.39.2.16 int MathUtil::RandomInt (int *maximum*) [static]

Get a random non-negative integer.

Parameters

maximum	The maximum value you want to see

Returns

A random number between 0 (inclusive) and maximum (exclusive)

Definition at line 85 of file MathUtil.cpp.

2.39.2.17 int MathUtil::RandomIntInRange (int *min*, int *max*) `[static]`

Get a random integer in a specified range.

Parameters

min	The minimum value you want to see
max	The maximum value you want to see

Returns

A random number between min (inclusive) and max (exclusive)

Definition at line 100 of file MathUtil.cpp.

2.39.2.18 int MathUtil::RandomIntWithError (int *target*, int *error*) `[static]`

Get a random integer within a specified range of another one.

Parameters

target	The target integer
error	The maximum amount the returned value can differ from the target (in either direction)

Returns

A random int from (target - error) to (target + error)

Definition at line 105 of file MathUtil.cpp.

2.39.2.19 float MathUtil::RandomFloat (float *maximum* = `1.0f`) `[static]`

Get a random non-negative float.

Parameters

maximum	The maximum value you want to see

Returns

A random number between 0.0 (inclusive) and maximum (exclusive)

Definition at line 110 of file MathUtil.cpp.

2.39.2.20 float MathUtil::RandomFloatInRange (float *min*, float *max*) `[static]`

Get a random float in a specified range.

Parameters

min	The minimum value you want to see
max	The maximum value you want to see

Returns

A random number between min (inclusive) and max (exclusive)

Definition at line 118 of file MathUtil.cpp.

2.39.2.21 float MathUtil::RandomFloatWithError (float *target*, float *error*) [static]

Get a random float within a specified range of another one.

Parameters

target	The target float
error	The maximum amount the returned value can differ from the target (in either direction)

Returns

A random float from (target - error) to (target + error)

Definition at line 123 of file MathUtil.cpp.

2.39.2.22 bool MathUtil::RandomBool ( ) [static]

Get a random bool.

Returns

Either true or false, randomly. :-)

Definition at line 128 of file MathUtil.cpp.

2.39.2.23 Vector2 MathUtil::RandomVector ( ) [static]

Get a random unit-length Vector2

Returns

A unit-length Vector2 pointing in a random direction

Definition at line 133 of file MathUtil.cpp.

2.39.2.24 Vector2 MathUtil::RandomVector (const Vector2 & *maxValues*) [static]

Get a random Vector2 with specified maximum values

Parameters

maxValues	The highest values for both axes

Returns

A random vector ranging from (0, 0) to (maxValues.X, maxValuesY)

Definition at line 138 of file MathUtil.cpp.

2.39.2.25 Vector2 MathUtil::RandomVector (const Vector2 & *minValues*, const Vector2 & *maxValues*) [static]

Get a random Vector2 within a specified range

Parameters

minValues	The lowest values for both axes
maxValues	The highest values for both axes

Returns

A random vector ranging from (minValues.X, minValues.Y) to (maxValues.X, maxValues.y)

Definition at line 143 of file MathUtil.cpp.

2.39.2.26 Vector2List MathUtil::RandomPointField (int *numPoints*, const Vector2 & *minValue*, const Vector2 & *maxValue*, float *minDistance* = `0.5f`) `[static]`

Get a set of random points in a Poisson disc distribution. (If you're not familiar with that term, just know that it's more likely to give a somewhat even distribution of random points, a more "natural" looking distribution, along the lines of what you're imagining when you think of "random."

Parameters

numPoints	The desired number of points
minValue	A point representing the bottom-left coordinate of the field
maxValue	A point representing the top-right coordinate of the field
minDistance	The smallest distance that will be allowed in the field, effectively determining the spacing. Note that this value will iteratively decrease if the field ends up too tightly packed.

Returns

The list of generated points

Definition at line 159 of file MathUtil.cpp.

2.39.2.27 bool MathUtil::FuzzyEquals (float *value1*, float *value2*, float *epsilon* = Epsilon) `[static]`

Compare two floating point values for "equality," with a permissible amount of error. Oftentimes you only care if floats are "close enough for government work," and this function lets you make that determination.

(Because of rounding errors inherent in floating point arithmetic, direct comparison of floats is often inadvisable. `http://en.wikipedia.org/wiki/Floating_point#Accuracy_problems`)

Parameters

value1	The first value
value2	The second value
epsilon	The maximum allowable difference (defaults to MathUtil::Epsilon)

Returns

Whether the two values are within epsilon of each other

Definition at line 193 of file MathUtil.cpp.

2.39.2.28 bool MathUtil::FuzzyEquals (const Vector2 & *v1*, const Vector2 & *v2*, float *epsilon* = Epsilon) `[static]`

A Vector2 comparison function using FuzzyEquals on the components.

Parameters

v1	The first vector
v2	The second vector
epsilon	The maximum allowable difference between them

Returns

Whether the two vectors have components within epsilon of each other

Definition at line 204 of file MathUtil.cpp.

2.39.2.29 Vector2 MathUtil::ScreenToWorld (int *x*, int *y*) `[static]`

Convert screen (pixel) coordinates to world (GL unit) coordinates. This function is not terribly efficient, so be careful calling it too frequently.

Parameters

x	The pixel X coordinate
y	The pixel Y coordinate

Returns

The world space coordinates

Definition at line 218 of file MathUtil.cpp.

2.39.2.30 Vector2 MathUtil::ScreenToWorld (const Vec2i & *screenCoordinates*) `[static]`

Convert screen (pixel) coordinates to world (GL unit) coordinates. This function is not terribly efficient, so be careful calling it too frequently.

Parameters

screen-Coordinates	The pixel coordinates

Returns

The world space coordinates

Definition at line 213 of file MathUtil.cpp.

2.39.2.31 Vector2 MathUtil::WorldToScreen (float *x*, float *y*) `[static]`

Convert world (GL unit) coordinates to screen (pixel) coordinates. This function is not terribly efficient, so be careful calling it too frequently.

Parameters

x	The world X coordinate
y	The world Y coordinate

Returns

The screen space coordinates

Definition at line 253 of file MathUtil.cpp.

2.39.2.32 Vector2 MathUtil::WorldToScreen (const Vector2 & *worldCoordinates*) `[static]`

Convert world (GL unit) coordinates to screen (pixel) coordinates. This function is not terribly efficient, so be careful calling it too frequently.

Parameters

world-Coordinates	The world coordinates

Returns

The screen space coordinates

Definition at line 237 of file MathUtil.cpp.

2.39.2.33 Vector2 MathUtil::GetWorldDimensions ( ) [static]

Find out the dimensions of the area currently displayed in the window.

Returns

The dimensions of the viewport in GL units

Definition at line 258 of file MathUtil.cpp.

2.39.2.34 float MathUtil::PixelsToWorldUnits (float *pixels*) [static]

Take a number of pixels and find out how many GL units they cover

Parameters

pixels	The number of pixels

Returns

The number of GL units

Definition at line 280 of file MathUtil.cpp.

2.39.2.35 float MathUtil::WorldUnitsToPixels (float *worldUnits*) [static]

Take a number of GL units and find out how many pixels stretch across that distance.

Parameters

worldUnits	The number of GL units

Returns

The number of pixels

Definition at line 287 of file MathUtil.cpp.

2.39.2.36 MathUtil::AABBSplittingAxis MathUtil::GetMajorAxis (const BoundingBox & *source*) [static]

Used internally by the SpatialGraph when it generates.

Parameters

source	The bounding box to split

Returns

The appropriate axis on which to split it

Definition at line 294 of file MathUtil.cpp.

2.39.2.37 void MathUtil::SplitBoundingBox (const BoundingBox & *source*, AABBSplittingAxis *axis*, BoundingBox & *LHS*, BoundingBox & *RHS*) [static]

Used internally by the SpatialGraph when it generates

Parameters

source	The bounding box to split
axis	The axis on which to split it
LHS	An out parameter that will be set to the left-hand-side bounding box resulting from the split
RHS	An out parameter that will be set to the right-hand-side bounding box resulting from the split

Definition at line 309 of file MathUtil.cpp.

2.39.2.38 float MathUtil::DeltaAngle (float *A1*, float *A2*) [static]

Calculates the difference in two angles (in radians) and remaps it to a range from -Pi to Pi.

Parameters

A1	The first angle (in radians)
A2	The second angle (in radians)

Returns

The difference, mapped appropriately

Definition at line 327 of file MathUtil.cpp.

2.39.2.39 float MathUtil::VectorDeltaAngle (const Vector2 & *v1*, const Vector2 & *v2*) [static]

Calculate the difference in angles between two vectors.

Parameters

v1	The first vector
v2	The second vector

Returns

The difference in their angles

Definition at line 349 of file MathUtil.cpp.

2.39.3 Member Data Documentation

2.39.3.1 const float MathUtil::E = 2.718282f [static]

Euler's number: `http://en.wikipedia.org/wiki/E_(mathematical_constant)`

Definition at line 51 of file MathUtil.h.

2.39.3.2 const float MathUtil::Log10E = 0.4342945f [static]

Base-10 logarithm of Euler's number

Definition at line 56 of file MathUtil.h.

2.39.3.3 const float MathUtil::Log2E = 1.442695f `[static]`

Base-2 logarithm of Euler's number

Definition at line 61 of file MathUtil.h.

2.39.3.4 const float MathUtil::Pi = 3.141593f `[static]`

Pi: `http://en.wikipedia.org/wiki/Pi`

Definition at line 66 of file MathUtil.h.

2.39.3.5 const float MathUtil::PiOver2 = 1.570796f `[static]`

Pi divided by 2

Definition at line 71 of file MathUtil.h.

2.39.3.6 const float MathUtil::PiOver4 = 0.7853982f `[static]`

Pi divided by 4

Definition at line 76 of file MathUtil.h.

2.39.3.7 const float MathUtil::TwoPi = 6.283185f `[static]`

2 times Pi (Tau)

Definition at line 81 of file MathUtil.h.

2.39.3.8 const float MathUtil::MaxFloat = 3.402823E+38f `[static]`

The maximum value that can be represented by a 32-bit floating point number

Definition at line 87 of file MathUtil.h.

2.39.3.9 const float MathUtil::MinFloat = -3.402823E+38f `[static]`

The minimum value that can be represented by a 32-bit floating point number

Definition at line 93 of file MathUtil.h.

2.39.3.10 const float MathUtil::Epsilon = 0.000001f `[static]`

A very tiny number, useful for floating point comparisons (10e-6)

Definition at line 98 of file MathUtil.h.

The documentation for this class was generated from the following files:

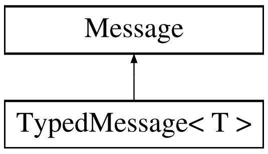
- Util/MathUtil.h
- Util/MathUtil.cpp

2.40 Message Class Reference

The base message class which signals an event.

`#include <Message.h>`

Inheritance diagram for Message:



Public Member Functions

- Message ()
- virtual ∼Message ()
- Message (const String &messageName, MessageListener ∗sender=NULL)
- virtual const String & GetMessageName () const
- MessageListener ∗const GetSender ()

Protected Attributes

- String **_messageName**
- MessageListener ∗ **_sender**

2.40.1 Detailed Description

A message is used by the Switchboard class to signal to any listeners that an event has happened. The only information it conveys is the message name (a user-definable string) and who sent the message.

Classes which implement the MessageListener interface can subscribe to specific types of messages to get notification.

See Also

MessageListener
TypedMessage
Switchboard

Definition at line 50 of file Message.h.

2.40.2 Constructor & Destructor Documentation

2.40.2.1 Message::Message ( )

The default constructor creates a Message with the name "GenericMessage" and no sender.

Definition at line 34 of file Message.cpp.

2.40.2.2 Message::∼Message ( ) [virtual]

Virtual destructor.

Definition at line 40 of file Message.cpp.

2.40.2.3 Message::Message (const String & *messageName*, MessageListener ∗ *sender* = NULL)

This is the constructor you should actually use – gives the Message a name and sender.

Parameters

messageName	The name of this Message; used by MessageListener and the Switchboard to manage delivery.
sender	Who sent this Message; NULL by default

Definition at line 44 of file Message.cpp.

2.40.3 Member Function Documentation

2.40.3.1 const String & Message::GetMessageName ( ) const [virtual]

Get the name of this Message. Since all Messages come to the listener via the same ReceiveMessage function, this can be used to filter for specific notifications.

Returns

The name of this message

Definition at line 50 of file Message.cpp.

2.40.3.2 MessageListener ∗const Message::GetSender ( )

Find out who requested this Message to be sent. This can have all sorts of semantics depending on what the Message itself is communicating.

Returns

The sender

Definition at line 55 of file Message.cpp.

The documentation for this class was generated from the following files:

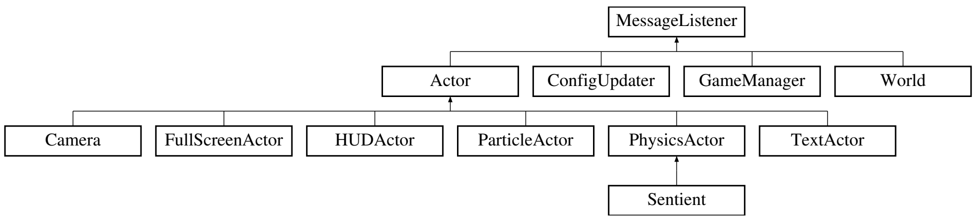
- Messaging/Message.h
- Messaging/Message.cpp

2.41 MessageListener Class Reference

An interface for sending and receiving Messages via the Switchboard.

`#include <Message.h>`

Inheritance diagram for MessageListener:



Public Member Functions

- virtual void ReceiveMessage (Message ∗m)=0

2.41.1 Detailed Description

Any class that wants to participate in messaging must implement this interface. It simply defines a function for the Switchboard to call when delivering Messages.

Definition at line 148 of file Message.h.

2.41.2 Member Function Documentation

2.41.2.1 virtual void MessageListener::ReceiveMessage (Message ∗ *m*) [pure virtual]

The Switchboard class will call this function for every Message to be deliverd to a designated listener. Implementations of this function should filter on Message::GetMessageName to make sure they're responding to the appropriate signals.

Parameters

m	The message to be delivered.

Implemented in Actor, World, TextActor, GameManager, FullScreenActor, and ConfigUpdater.

The documentation for this class was generated from the following files:

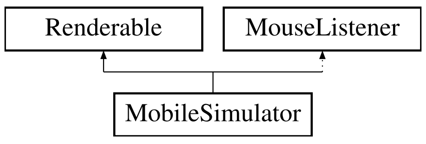
- Messaging/Message.h
- Messaging/Message.cpp

2.42 MobileSimulator Class Reference

A class to somewhat simulate the setup of an iOS app on the desktop.

```
#include <MobileSimulator.h>
```

Inheritance diagram for MobileSimulator:



Public Member Functions

- virtual void Update (float dt)
- virtual void MouseMotionEvent (Vec2i screenCoordinates)
- virtual void MouseDownEvent (Vec2i screenCoordinates, MouseButtonInput button)
- virtual void MouseUpEvent (Vec2i screenCoordinates, MouseButtonInput button)

Additional Inherited Members

2.42.1 Detailed Description

If you add this Renderable to the world, it'll pretend to be the mobile hardware, filling the same data structures that the real hardware would, meaning you can use this to prototype a lot of iOS gameplay without having the hardware immediately accessible.

You shouldn't actually have to call any of the methods on this class. Add it to the world and then use the interfaces defined in MultiTouch.h.

Definition at line 45 of file MobileSimulator.h.

2.42.2 Member Function Documentation

2.42.2.1 void MobileSimulator::Update (float *dt*) [virtual]

This function gets called once per frame. Any game logic for an Actor should be done in this function, since it provides you with a dt for controlling rate of movement, animation, etc.

Parameters

dt	The amount of time (in seconds) that has elapsed since the last frame.

Reimplemented from Renderable.

Definition at line 79 of file MobileSimulator.cpp.

2.42.2.2 void MobileSimulator::MouseMotionEvent (Vec2i *screenCoordinates*) [virtual]

Called whenever the player moves the mouse.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.

Reimplemented from MouseListener.

Definition at line 94 of file MobileSimulator.cpp.

2.42.2.3 void MobileSimulator::MouseDownEvent (Vec2i *screenCoordinates,* MouseButtonInput *button*) [virtual]

Called whenever the player presses down on a mouse button.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.
button	Which button was pressed. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_R-IGHT`.

Reimplemented from MouseListener.

Definition at line 183 of file MobileSimulator.cpp.

2.42.2.4 void MobileSimulator::MouseUpEvent (Vec2i *screenCoordinates,* MouseButtonInput *button*) [virtual]

Called whenever the player releases a mouse button.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.
button	Which button was released. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_R-IGHT`.

Reimplemented from MouseListener.

Definition at line 219 of file MobileSimulator.cpp.

The documentation for this class was generated from the following files:

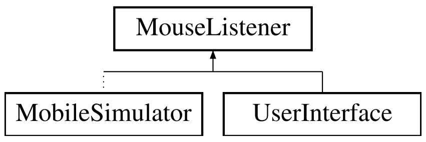
- Input/MobileSimulator.h
- Input/MobileSimulator.cpp

2.43 MouseListener Class Reference

An abstract interface for getting mouse events.

`#include <MouseInput.h>`

Inheritance diagram for MouseListener:



Public Member Functions

- MouseListener ()
- virtual ∼MouseListener ()
- virtual void MouseMotionEvent (Vec2i screenCoordinates)
- virtual void MouseDownEvent (Vec2i screenCoordinates, MouseButtonInput button)
- virtual void MouseUpEvent (Vec2i screenCoordinates, MouseButtonInput button)
- virtual void MouseWheelEvent (const Vector2 &scrollOffset)

2.43.1 Detailed Description

This is an abstract base class which provides an interface to get notifications about what's going on with the mouse. If you want to get mouse data, derive a class from this one and implement the Mouse∗Event member functions.

Definition at line 49 of file MouseInput.h.

2.43.2 Constructor & Destructor Documentation

2.43.2.1 MouseListener::MouseListener ( )

Base constructor adds this object to the list of objects to get notified when mousey things happen.

Definition at line 38 of file MouseInput.cpp.

2.43.2.2 MouseListener::∼MouseListener ( ) `[virtual]`

Base destructor removes this object from the list.

Definition at line 43 of file MouseInput.cpp.

2.43.3 Member Function Documentation

2.43.3.1 void MouseListener::MouseMotionEvent (Vec2i *screenCoordinates*) `[virtual]`

Called whenever the player moves the mouse.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.

Reimplemented in UserInterface, and MobileSimulator.

Definition at line 51 of file MouseInput.cpp.

2.43.3.2 void MouseListener::MouseDownEvent (Vec2i *screenCoordinates*, MouseButtonInput *button*) [virtual]

Called whenever the player presses down on a mouse button.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.
button	Which button was pressed. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_R-IGHT`.

Reimplemented in UserInterface, and MobileSimulator.

Definition at line 49 of file MouseInput.cpp.

2.43.3.3 void MouseListener::MouseUpEvent (Vec2i *screenCoordinates*, MouseButtonInput *button*) [virtual]

Called whenever the player releases a mouse button.

Parameters

screen-Coordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::Screen-ToWorld function if you want GL units.
button	Which button was released. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_R-IGHT`.

Reimplemented in UserInterface, and MobileSimulator.

Definition at line 50 of file MouseInput.cpp.

2.43.3.4 void MouseListener::MouseWheelEvent (const Vector2 & *scrollOffset*) [virtual]

Called whenever the player moves the scroll wheel on the mouse.

Parameters

scrollOffset	The change in position of the scroll wheel. Note that if it's an actual wheel, the X component of the vector will always be 0; the two-dimensional vector also takes into account trackpad scrolling.

Reimplemented in UserInterface.

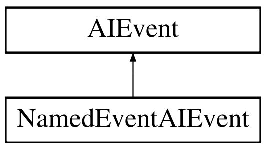
Definition at line 52 of file MouseInput.cpp.

The documentation for this class was generated from the following files:

- Input/MouseInput.h
- Input/MouseInput.cpp

2.44 NamedEventAIEvent Class Reference

Inheritance diagram for NamedEventAIEvent:



Public Member Functions

- virtual NamedEventAIEvent ∗ **Initialize** (const String &eventId, StringList ∗eventIdList)
- virtual void **Update** (float dt)

Protected Attributes

- String **_eventId**
- StringList ∗ **_eventIdList**

Additional Inherited Members

2.44.1 Detailed Description

Definition at line 35 of file NamedEventAIEvent.h.

The documentation for this class was generated from the following files:

- AIEvents/NamedEventAIEvent.h
- AIEvents/NamedEventAIEvent.cpp

2.45 AStarSearch< UserState >::Node Class Reference

Public Attributes

- Node ∗ **parent**
- Node ∗ **child**
- float **g**
- float **h**
- float **f**
- UserState **m_UserState**

2.45.1 Detailed Description

template<class UserState>class AStarSearch< UserState >::Node

Definition at line 74 of file stlastar.h.

The documentation for this class was generated from the following file:

- AI/stlastar.h

2.46 ParticleActor::Particle Struct Reference

Public Attributes

- Vector2 **_pos**
- Vector2 **_vel**
- float **_age**
- float **_lifetime**
- Color **_color**
- float **_scale**

2.46.1 Detailed Description

Definition at line 201 of file ParticleActor.h.

The documentation for this struct was generated from the following file:

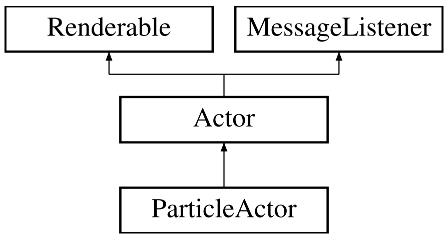
- Actors/ParticleActor.h

2.47 ParticleActor Class Reference

An Actor that draws and keeps track of drawing a particle system on screen.

`#include <ParticleActor.h>`

Inheritance diagram for ParticleActor:



Classes

- struct Particle

Public Member Functions

- ParticleActor ()
- ∼ParticleActor ()
- virtual void Update (float dt)
- virtual void Render ()
- void SetParticlesPerSecond (float pps)
- void SetSystemLifetime (float lifetime)
- void SetParticleLifetime (float lifetime)
- void SetSpread (float radians)
- void SetEndScale (float scale)
- void SetEndColor (const Color &color)
- void SetSpeedRange (float minSpeed, float maxSpeed)
- void SetMinSpeed (float minSpeed)
- void SetMaxSpeed (float maxSpeed)
- void SetGravity (const Vector2 &gravity)
- void SetAttractor (const Vector2 &attractor)
- void SetAttractorStrength (float strength)
- void SetMaxParticles (int maxParticles)
- virtual const String GetClassName () const

Protected Attributes

- Particle ∗ **_particles**
- int **_maxParticlesAlive**
- int **_numParticlesAlive**

- float **_particlesPerSecond**
- int **_maxParticlesToGenerate**
- float **_generationResidue**
- float **_systemLifetime**
- float **_particleLifetime**
- float **_spreadRadians**
- Color **_endColor**
- float **_minSpeed**
- float **_maxSpeed**
- float **_endScale**
- Vector2 **_gravity**
- Vector2 **_attractor**
- float **_attractorStrength**

Additional Inherited Members

2.47.1 Detailed Description

Particle systems are a very common feature of most games – they can be used for effects like fire, smoke, sparklies, etc. A discussion of the general use of particles is beyond the scope of this documentation, but if you're not familiar with the concept, this article by Jeff Lander is a good starting point.

`http://www.double.co.nz/dust/col0798.pdf`

Definition at line 46 of file ParticleActor.h.

2.47.2 Constructor & Destructor Documentation

2.47.2.1 ParticleActor::ParticleActor ( )

The default constructor creates a particle system that doesn't really do anything, since its maximum number of particles is 0. You'll need to call the "Set" functions below to make it look purty.

Definition at line 35 of file ParticleActor.cpp.

2.47.2.2 ParticleActor::∼ParticleActor ( )

Deletes all the particles this Actor is keeping track of.

Definition at line 62 of file ParticleActor.cpp.

2.47.3 Member Function Documentation

2.47.3.1 void ParticleActor::Update (float *dt*) `[virtual]`

Override of the normal Renderable::Update function. Changes the position and appearance of each individual particle appropriately.

Parameters

dt	The amount of time that's elapsed since the beginning of the last frame.

Reimplemented from Actor.

Definition at line 67 of file ParticleActor.cpp.

2.47.3.2 void ParticleActor::Render ( ) `[virtual]`

Override of the normal Renderable::Render function. Draws each particle.

Reimplemented from Actor.

Definition at line 182 of file ParticleActor.cpp.

2.47.3.3 void ParticleActor::SetParticlesPerSecond (float *pps*)

Change the rate at which this system releases particles. Default is 20 particles per second.

Parameters

pps	The new release rate in particles per second.

Definition at line 240 of file ParticleActor.cpp.

2.47.3.4 void ParticleActor::SetSystemLifetime (float *lifetime*)

Change the system lifetime for this Actor. If the lifetime is set less than or equal to 0.0 (the default), the system will last until it's explicitly removed from the world. Otherwise the system will properly remove and deallocate itself when the elapsed time is up. (Useful for when you just want a burst of particles for a period of time and don't want to set up housekeeping timer.)

Parameters

lifetime	The amount of time this system will persist, in seconds.

Definition at line 249 of file ParticleActor.cpp.

2.47.3.5 void ParticleActor::SetParticleLifetime (float *lifetime*)

Change the lifetime of each individual particle.

Parameters

lifetime	Particle lifetime in seconds

Definition at line 258 of file ParticleActor.cpp.

2.47.3.6 void ParticleActor::SetSpread (float *radians*)

Set the angle at which particles will disperse themselves. This affects their intial velocity only. Default is 0.0 (straight line to the right).

Parameters

radians	Emission angle in radians

Definition at line 267 of file ParticleActor.cpp.

2.47.3.7 void ParticleActor::SetEndScale (float *scale*)

Set the relative size each particle will grow (or shrink) to over its lifetime. The starting size is set with the normal Actor::SetSize function. The ending scale respects the starting aspect ratio.

Parameters

scale	The multiplier gradually applied to each particle

Definition at line 272 of file ParticleActor.cpp.

2.47.3.8 void ParticleActor::SetEndColor (const Color & *color*)

Set the color each particle should be at the end of its life. Use an alpha of 0.0 to have the particles fade out over time. Starting color is set with the normal Actor::SetColor function.

Parameters

color	The ending color for each particle in the system.

Definition at line 277 of file ParticleActor.cpp.

2.47.3.9 void ParticleActor::SetSpeedRange (float *minSpeed,* float *maxSpeed*)

Set the range of potential initial speeds for the particles. Each particle, at the start of its life, will randomly pick a speed in the given range.

Parameters

minSpeed	The speed (in GL units per second) of the slowest particle
maxSpeed	The speed (in GL units per second) of the fastest particle

Definition at line 282 of file ParticleActor.cpp.

2.47.3.10 void ParticleActor::SetMinSpeed (float *minSpeed*)

Set the lower range of potential initial speeds for the particles.

Parameters

minSpeed	The speed (in GL units per second) of the slowest particle

Definition at line 288 of file ParticleActor.cpp.

2.47.3.11 void ParticleActor::SetMaxSpeed (float *maxSpeed*)

Set the lower range of potential initial speeds for the particles.

Parameters

maxSpeed	The speed (in GL units per second) of the fastest particle

Definition at line 293 of file ParticleActor.cpp.

2.47.3.12 void ParticleActor::SetGravity (const Vector2 & *gravity*)

Set the vector which will pull the particles in a specific direction at a specified magnitude. The default is (0.0, -4.0).

Parameters

gravity	The vector in which the particles should be pulled (magnitude affecting the force of the pull).

Definition at line 298 of file ParticleActor.cpp.

2.47.3.13 void ParticleActor::SetAttractor (const Vector2 & *attractor*)

Set the attractor point for the particles – in addition to gravity, which gets applied as a vector, particles will be affected by their attraction to this point. Defaults to the origin (0, 0), but since the default attractor strength is zero, it will not actually affect any movement out of the box.

Parameters

attractor	The point to which all particles emitted by the system will be drawn. It is their destiny.

Definition at line 303 of file ParticleActor.cpp.

2.47.3.14 void ParticleActor::SetAttractorStrength (float *strength*)

Set the strength of the attractor, or how quickly particles will move to it. This does a very simple linear movement; it's not a force in the physics sense, so they won't accelerate towards it or anything like that.

Parameters

strength	The magnitude of the vector between each particle and the attractor point.

Definition at line 308 of file ParticleActor.cpp.

2.47.3.15 void ParticleActor::SetMaxParticles (int *maxParticles*)

Set the maximum number of particles this system can keep track of at any one time. Decrease this number if you find you're having performance issues with your particles.

Parameters

maxParticles	The maximum number of particles for this system.

Definition at line 313 of file ParticleActor.cpp.

2.47.3.16 virtual const String ParticleActor::GetClassName ( ) const `[inline]`, `[virtual]`

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "ParticleActor"

Reimplemented from Actor.

Definition at line 198 of file ParticleActor.h.

The documentation for this class was generated from the following files:

- Actors/ParticleActor.h
- Actors/ParticleActor.cpp

2.48 PathFinder Class Reference

Public Types

- enum **ePFMoveResult** { **PFMR_PATH_NOT_FOUND**, **PFMR_PATH_FOUND**, **PFMR_ARRIVED** }
- enum **ePFMoveState** {
  PFMS_START, **PFMS_VALIDATE**, **PFMS_FOLLOW**, **PFMS_RECOVER**,
  PFMS_STARTRECOVER, **PFMS_COUNT** }

Public Member Functions

- void **FindNextMove** (const Vector2 &from, const Vector2 &to, float arrivalDist, PathFinderMove &move)
- void **Render** ()

Static Public Member Functions

- static void **EnableDrawPaths** (bool enable)

Friends

- class **FindNextMoveState**

2.48.1 Detailed Description

Definition at line 37 of file PathFinder.h.

The documentation for this class was generated from the following files:

- AI/PathFinder.h
- AI/PathFinder.cpp

2.49 PathFinderMove Struct Reference

Public Attributes

- Vector2 **MoveDir**
- Vector2 **NextSubgoalPos**
- PathFinder::ePFMoveResult **LastResult**

2.49.1 Detailed Description

Definition at line 110 of file PathFinder.h.

The documentation for this struct was generated from the following file:

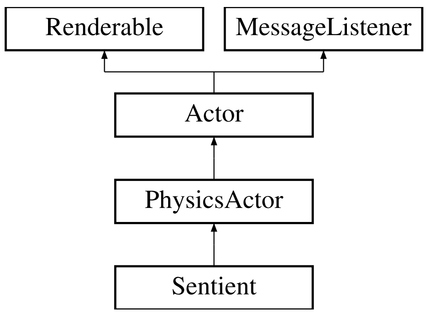
- AI/PathFinder.h

2.50 PhysicsActor Class Reference

An Actor that interacts with other Actors using our built-in physics system.

```
#include <PhysicsActor.h>
```

Inheritance diagram for PhysicsActor:



Public Types

- enum eShapeType { **SHAPETYPE_BOX**, **SHAPETYPE_CIRCLE** }

Public Member Functions

- PhysicsActor ()
- virtual ∼PhysicsActor ()
- void SetDensity (float density)
- void SetFriction (float friction)
- void SetRestitution (float restitution)
- void SetShapeType (eShapeType shapeType)
- void SetIsSensor (bool isSensor)
- void SetGroupIndex (int groupIndex)
- void SetFixedRotation (bool fixedRotation)
- virtual void InitPhysics ()
- virtual void CustomInitPhysics ()
- void ApplyForce (const Vector2 &force, const Vector2 &point)
- void ApplyLocalForce (const Vector2 &force, const Vector2 &point)
- void ApplyTorque (float torque)
- void ApplyLinearImpulse (const Vector2 &impulse, const Vector2 &point)
- void ApplyAngularImpulse (float impulse)
- b2Body ∗ GetBody ()
- void ResetBody ()
- void SetSize (float x, float y=-1.f)
- void SetDrawSize (float x, float y=-1.f)
- void SetPosition (float x, float y)
- void SetPosition (const Vector2 &pos)
- void SetRotation (float rotation)
- void MoveTo (const Vector2 &newPosition, float duration, String onCompletionMessage="")
- void RotateTo (float newRotation, float duration, String onCompletionMessage="")
- void ChangeSizeTo (const Vector2 &newSize, float duration, String onCompletionMessage="")
- void ChangeSizeTo (float newSize, float duration, String onCompletionMessage="")
- virtual const String GetClassName () const

Protected Member Functions

- virtual void **InitShape** (b2Shape ∗)

Protected Attributes

- b2Body ∗ **_physBody**
- float **_density**
- float **_friction**
- float **_restitution**
- eShapeType **_shapeType**
- bool **_isSensor**
- int **_groupIndex**
- bool **_fixedRotation**

Friends

- class **World**

Additional Inherited Members

2.50.1 Detailed Description

Angel incorporates Box2D (`http://www.box2d.org`) to handle our physics simulation. This Actor is a loose wrapping around Box2D so that you don't need to worry about the underlying library if you're just doing simple physics.

Definition at line 44 of file PhysicsActor.h.

2.50.2 Member Enumeration Documentation

2.50.2.1 enum **PhysicsActor::eShapeType**

The two physics shapes we currently support. If you want anything else, you're going to be delving into the Box2D library itself.

Definition at line 67 of file PhysicsActor.h.

2.50.3 Constructor & Destructor Documentation

2.50.3.1 PhysicsActor::PhysicsActor (void)

Sets up a new PhysicsActor with some reasonable default physical properties. Box-shaped; Density = 1.0; Friction = 0.3; Restitution = 0.0.

Note that the PhysicsActor won't actually start doing anything until you call PhysicsActor::InitPhysics().

Definition at line 43 of file PhysicsActor.cpp.

2.50.3.2 PhysicsActor::∼PhysicsActor ( ) `[virtual]`

The destructor removes the PhysicsActor from the internal simulation. It's **very** important to call the destructor when your remove a PhysicsActor from your World, because otherwise the physics engine will think it's still around and you'll get some strange results.

Definition at line 55 of file PhysicsActor.cpp.

2.50.4 Member Function Documentation

2.50.4.1 void PhysicsActor::SetDensity (float *density*)

Set the density to be used by the physics simulation. Box2D assumes that the units are kg/m^2. If you set density to 0.0, the PhysicsActor is static and will never move in the world. (Other objects will still collide with it.)

Note that after you call PhysicsActor::InitPhysics, the density is locked and this function will do nothing but spew a warning.

Parameters

density	Desired density

Definition at line 64 of file PhysicsActor.cpp.

2.50.4.2 void PhysicsActor::SetFriction (float *friction*)

Set the friction to be used by the physics simulation. 0.0 indicates no friction, while 1.0 indicates very high friction.

Note that after you call PhysicsActor::InitPhysics, the friction is locked and this function will do nothing but spew a warning.

Parameters

density	Desired friction

Definition at line 72 of file PhysicsActor.cpp.

2.50.4.3 void PhysicsActor::SetRestitution (float *restitution*)

Set the restitution (bounciness) to be used by the physics simulation. 0.0 indicates no bounce at all, while 1.0 is a superball.

Note that after you call PhysicsActor::InitPhysics, the restitution is locked and this function will do nothing but spew a warning.

Parameters

density	Desired restitution

Definition at line 80 of file PhysicsActor.cpp.

2.50.4.4 void PhysicsActor::SetShapeType (eShapeType *shapeType*)

Set the shape you want to use for this object.

Note that after you call PhysicsActor::InitPhysics, the restitution is locked and this function will do nothing but spew a warning.

Parameters

shapeType	Either SHAPETYPE_BOX or SHAPETYPE_CIRCLE.

Definition at line 88 of file PhysicsActor.cpp.

2.50.4.5 void PhysicsActor::SetIsSensor (bool *isSensor*)

If you set an object up as a sensor, it will cause collision messages to be sent, but not actually physically react to the collision. Useful for detecting when objects have entered certain regions.

Note that after you call PhysicsActor::InitPhysics, the restitution is locked and this function will do nothing but spew a warning.

Parameters

isSensor	True if the object should be a sensor, false if not. PhysicsActors are assumed to **not** be sensors unless otherwise specified.

Definition at line 96 of file PhysicsActor.cpp.

2.50.4.6 void PhysicsActor::SetGroupIndex (int *groupIndex*)

Set a group index if you want to have objects only collide with certain other groups of objects. Positive numbers mean that objects always collide, negative numbers mean they never will.

See the Box2D documentation for more information on collision groups

Note that after you call PhysicsActor::InitPhysics, the group index is locked and this function will do nothing but spew a warning.

Parameters

groupIndex	The new group index for this PhysicsActor

Definition at line 104 of file PhysicsActor.cpp.

2.50.4.7 void PhysicsActor::SetFixedRotation (bool *fixedRotation*)

If true, this PhysicsActor will not rotate (useful for characters).

Note that after you call PhysicsActor::InitPhysics, the this value is locked and the function will do nothing but spew a warning.

Parameters

fixedRotation	

Definition at line 112 of file PhysicsActor.cpp.

2.50.4.8 void PhysicsActor::InitPhysics ( ) [virtual]

Start simulating this PhysicsActor in the world.

Definition at line 121 of file PhysicsActor.cpp.

2.50.4.9 virtual void PhysicsActor::CustomInitPhysics ( ) [inline], [virtual]

If you want to have your own setup in a derived class, you can implement this function there. It's called at the end of the base class's PhysicsActor::InitPhysics.

Definition at line 166 of file PhysicsActor.h.

2.50.4.10 void PhysicsActor::ApplyForce (const Vector2 & *force*, const Vector2 & *point*)

Apply a force to an object. The standard units are Newtons (kg ∗ m/s^2). The point parameter is a location in actor space; if you want world space, you can either convert it yourself or use GetBody() to manipulate the underlying Box2D API.

ApplyForce should be called every timestep if you want the force to be continuously applied.

Parameters

force	The force to apply (direction and magnitude are significant) in Newtons
point	The point in actor space to which the force should be applied

Definition at line 181 of file PhysicsActor.cpp.

2.50.4.11 void PhysicsActor::ApplyLocalForce (const Vector2 & *force*, const Vector2 & *point*)

Applies a force relative to the Actor. The standard units are Newtons (kg ∗ m/s^2). The point parameter is a location in actor space; if you want world space, you can either convert it yourself or use GetBody() to manipulate the underlying Box2D API.

ApplyLocalForce should be called every timestep if you want the force to be continuously applied.

Parameters

force	The force to apply (direction and magnitude are significant) in Newtons
point	The point in actor space to which the force should be applied

Definition at line 189 of file PhysicsActor.cpp.

2.50.4.12 void PhysicsActor::ApplyTorque (float *torque*)

Apply torque to affect angular velocity without affecting linear velocity. The standard units are Newton-meters.

Parameters

torque	Amount of torque to apply around the Z-axis.

Definition at line 197 of file PhysicsActor.cpp.

2.50.4.13 void PhysicsActor::ApplyLinearImpulse (const Vector2 & *impulse*, const Vector2 & *point*)

Similar to ApplyForce, but will immediately affect a PhysicsActor's velocity, as opposed to forces which need to be applied every frame. The result is that ApplyImpulse will appear to have roughly 30x the affect of ApplyForce.

For more information: `http://www.box2d.org/forum/viewtopic.php?f=3&t=260`

The point parameter is a location in actor space; if you want world space, you can either convert it yourself or use GetBody() to manipulate the underlying Box2D API.

Parameters

impulse	The strength of the impulse to apply.
point	

Definition at line 205 of file PhysicsActor.cpp.

2.50.4.14 void PhysicsActor::ApplyAngularImpulse (float *impulse*)

Similar to ApplyLinearImpulse, but affects the angular rather than linear velocity.

Parameters

impulse	The strength of the impulse to apply.

Definition at line 213 of file PhysicsActor.cpp.

2.50.4.15 b2Body∗ PhysicsActor::GetBody ( ) `[inline]`

Get the Box2D representation of this PhysicsActor. If you're going to be directly manipulating the Box2D API, this will be useful to you.

Returns

The Box2D data about the actor

Definition at line 240 of file PhysicsActor.h.

2.50.4.16 void PhysicsActor::ResetBody ( ) `[inline]`

Resets the internal pointer to the Box2D physics body to NULL. Call this if you're manually destroying the body for some reason and need to make sure the PhysicsActor doesn't keep trying to track it.

Definition at line 248 of file PhysicsActor.h.

2.50.4.17 void PhysicsActor::SetSize (float *x*, float *y* = `-1.f`)

An override of the Actor::SetSize function that disables itself after InitPhysics has been called.

Definition at line 221 of file PhysicsActor.cpp.

2.50.4.18 void PhysicsActor::SetDrawSize (float *x*, float *y* = `-1.f`)

Sometimes you want the visible size of an Actor to be larger or smaller than its collision geometry in the world. This function lets you alter the former without affecting the latter.

Parameters

x	Horizontal draw size in OpenGL units – negative numbers treated as zero
y	Vertical draw size in OpenGL units – if less than or equal to zero, assumed to be equal to x

Definition at line 229 of file PhysicsActor.cpp.

2.50.4.19 void PhysicsActor::SetPosition (float *x*, float *y*) [virtual]

An override of the Actor::SetPosition function that disables itself after InitPhysics has been called.

Reimplemented from Actor.

Definition at line 234 of file PhysicsActor.cpp.

2.50.4.20 void PhysicsActor::SetPosition (const Vector2 & *pos*) [virtual]

An override of the Actor::SetPosition function that disables itself after InitPhysics has been called.

Reimplemented from Actor.

Definition at line 242 of file PhysicsActor.cpp.

2.50.4.21 void PhysicsActor::SetRotation (float *rotation*) [virtual]

An override of the Actor::SetRotation function that disables itself after InitPhysics has been called.

Reimplemented from Actor.

Definition at line 250 of file PhysicsActor.cpp.

2.50.4.22 void PhysicsActor::MoveTo (const Vector2 & *newPosition*, float *duration*, String *onCompletionMessage* = "") [inline]

An override of the Actor::MoveTo function that doesn't allow the interval to be applied to PhysicsActors.

Definition at line 290 of file PhysicsActor.h.

2.50.4.23 void PhysicsActor::RotateTo (float *newRotation*, float *duration*, String *onCompletionMessage* = "") [inline]

An override of the Actor::RotateTo function that doesn't allow the interval to be applied to PhysicsActors.

Definition at line 296 of file PhysicsActor.h.

2.50.4.24 void PhysicsActor::ChangeSizeTo (const Vector2 & *newSize*, float *duration*, String *onCompletionMessage* = "") [inline]

An override of the Actor::ChangeSizeTo function that doesn't allow the interval to be applied to PhysicsActors.

Definition at line 302 of file PhysicsActor.h.

2.50.4.25 void PhysicsActor::ChangeSizeTo (float *newSize*, float *duration*, String *onCompletionMessage* = "") [inline]

An override of the Actor::ChangeSizeTo function that doesn't allow the interval to be applied to PhysicsActors.

Definition at line 308 of file PhysicsActor.h.

2.50.4.26 virtual const String PhysicsActor::GetClassName ( ) const [inline], [virtual]

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "PhysicsActor"

Reimplemented from Actor.

Definition at line 316 of file PhysicsActor.h.

The documentation for this class was generated from the following files:

- Actors/PhysicsActor.h

- Actors/PhysicsActor.cpp

2.51 Preferences Class Reference

A centralized class that handles any persistent preferences.

`#include <Preferences.h>`

Public Member Functions

- void SavePreferences ()
- int GetInt (const String &category, const String &name)
- float GetFloat (const String &category, const String &name)
- String GetString (const String &category, const String &name)
- LoadedVariableMap GetTable (const String &category)
- void SetInt (const String &category, const String &name, int val)
- void SetFloat (const String &category, const String &name, float val)
- void SetString (const String &category, const String &name, const String &val)
- int OverrideInt (const String &category, const String &name, int val)
- float OverrideFloat (const String &category, const String &name, float val)
- String OverrideString (const String &category, const String &name, String val)
- const String GetDefaultPath ()
- const String GetUserPrefsPath ()

Static Public Member Functions

- static Preferences & GetInstance ()

2.51.1 Detailed Description

We very often want to define persistent preferences for the user. They are stored out in .lua files, so they can be easily written or inspected by hand if needed.

When the game starts, the game will first load preferences from `Config/defaults.lua`, then load saved preferences, overriding any default preferences.

Preferences are organized into categories, represented as discrete tables in the .lua files. For example, if this was the preferences file:

```
GameStart = {
    enemies = 15,
    heroName = "Superdude",
}
```

Then we would retrieve the preference using "GameStart" as the category, and "enemies" or "heroName" as the value. Preferences can either be integers, floats, or Strings.

Definition at line 60 of file Preferences.h.

2.51.2 Member Function Documentation

2.51.2.1 Preferences & Preferences::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "thePrefs".

Returns

The singleton

Definition at line 42 of file Preferences.cpp.

2.51.2.2 void Preferences::SavePreferences ( )

Writes out all currently set preferences to a user-writable directory, which will override the defaults the next time the game launches.

Definition at line 56 of file Preferences.cpp.

2.51.2.3 int Preferences::GetInt (const String & *category*, const String & *name*)

Retrieves a given preference in integer form.

Parameters

category	The category of the desired preference
name	The value of the desired preference

Returns

The retrieved integer (0 if this preference has never been set)

Definition at line 93 of file Preferences.cpp.

2.51.2.4 float Preferences::GetFloat (const String & *category*, const String & *name*)

Retrieves a given preference in float form.

Parameters

category	The category of the desired preference
name	The value of the desired preference

Returns

The retrieved float (0.0f if this preference has never been set)

Definition at line 109 of file Preferences.cpp.

2.51.2.5 String Preferences::GetString (const String & *category*, const String & *name*)

Retrieves a given preference in String form.

Parameters

category	The category of the desired preference
name	The value of the desired preference

Returns

The retrieved String (Empty string if this preference has never been set)

Definition at line 125 of file Preferences.cpp.

2.51.2.6 LoadedVariableMap Preferences::GetTable (const String & *category*)

Gets all variables in a given category.

Parameters

category	The desired category

Returns

All defined preferences for this category

Definition at line 141 of file Preferences.cpp.

2.51.2.7 void Preferences::SetInt (const String & *category,* const String & *name,* int *val*)

Sets a preference as an integer. If it did not previously exist, it will be created.

Parameters

category	The category of the preference to set
name	The value of the preference to set
val	The actual value we want to store

Definition at line 161 of file Preferences.cpp.

2.51.2.8 void Preferences::SetFloat (const String & *category,* const String & *name,* float *val*)

Sets a preference as a float. If it did not previously exist, it will be created.

Parameters

category	The category of the preference to set
name	The value of the preference to set
val	The actual value we want to store

Definition at line 175 of file Preferences.cpp.

2.51.2.9 void Preferences::SetString (const String & *category,* const String & *name,* const String & *val*)

Sets a preference as a String. If it did not previously exist, it will be created.

Parameters

category	The category of the preference to set
name	The value of the preference to set
val	The actual value we want to store

Definition at line 189 of file Preferences.cpp.

2.51.2.10 int Preferences::OverrideInt (const String & *category,* const String & *name,* int *val*)

Takes a given integer, and returns either that integer or, if an appropriate preference exists, will return it. Handy for allowing hardcoded values to be overridden by user-defined ones.

Parameters

category	The category of the preference to check
name	The value of the preference to check
val	The value to return if there is no such preference

Returns

Either the given integer or the appropriate preference integer

Definition at line 203 of file Preferences.cpp.

2.51.2.11 float Preferences::OverrideFloat (const String & *category,* const String & *name,* float *val*)

Takes a given float, and returns either that float or, if an appropriate preference exists, will return it. Handy for allowing hardcoded values to be overridden by user-defined ones.

Parameters

category	The category of the preference to check
name	The value of the preference to check
val	The value to return if there is no such preference

Returns

Either the given float or the appropriate preference float

Definition at line 219 of file Preferences.cpp.

2.51.2.12 String Preferences::OverrideString (const String & *category,* const String & *name,* String *val*)

Takes a given String, and returns either that String or, if an appropriate preference exists, will return it. Handy for allowing hardcoded values to be overridden by user-defined ones.

Parameters

category	The category of the preference to check
name	The value of the preference to check
val	The value to return if there is no such preference

Returns

Either the given String or the appropriate preference String

Definition at line 235 of file Preferences.cpp.

2.51.2.13 const String Preferences::GetDefaultPath ( )

Used by the scripting system when it attempts to load the preferences files at game start.

Returns

The absolute path to the `Config/defaults.lua` file

Definition at line 82 of file Preferences.cpp.

2.51.2.14 const String Preferences::GetUserPrefsPath ( )

Used by the scripting system when it attempts to load the preferences files at game start.

Returns

The absolute path to the file which stores any preferences that were saved with thePrefs.SavePreferences()

Definition at line 87 of file Preferences.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/Preferences.h
- Infrastructure/Preferences.cpp

2.52 Ray2 Struct Reference

Public Member Functions

- **Ray2** (const Vector2 &_position, const Vector2 &_direction)

Static Public Member Functions

- static Ray2 **CreateRayFromTo** (const Vector2 &vFrom, const Vector2 &vTo)

Public Attributes

- Vector2 **Position**
- Vector2 **Direction**

2.52.1 Detailed Description

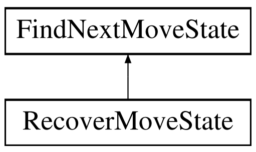
Definition at line 34 of file Ray2.h.

The documentation for this struct was generated from the following file:

- AI/Ray2.h

2.53 RecoverMoveState Class Reference

Inheritance diagram for RecoverMoveState:



Public Member Functions

- virtual const char ∗ **GetName** ()
- virtual bool **Update** (PathFinderMove &move)

Additional Inherited Members

2.53.1 Detailed Description

Definition at line 239 of file PathFinder.cpp.

The documentation for this class was generated from the following file:

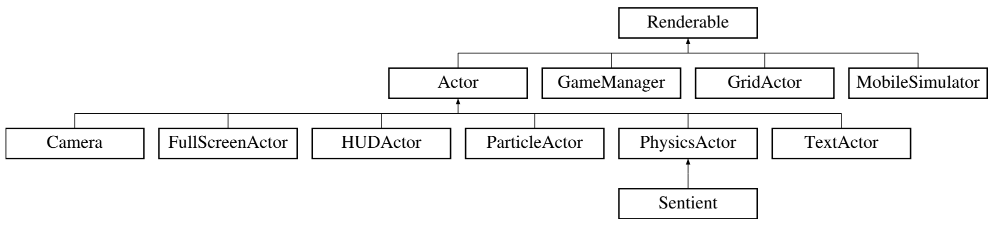
- AI/PathFinder.cpp

2.54 Renderable Class Reference

Our base simulation element that gets inserted to the World.

`#include <Renderable.h>`

Inheritance diagram for Renderable:



Public Member Functions

- Renderable ()
- virtual ~Renderable ()
- virtual void Update (float dt)
- virtual void Render ()
- void Destroy ()
- bool IsDestroyed ()
- int GetLayer ()

Protected Member Functions

- virtual void PreDestroy ()

Protected Attributes

- bool **_deleteMe**
- int **_layer**

Friends

- class **World**

2.54.1 Detailed Description

This is the base class from which most of the simulation elements in Angel derive. When you insert a Renderable into the World, it will receive Update and Render calls once per frame.

Definition at line 38 of file Renderable.h.

2.54.2 Constructor & Destructor Documentation

2.54.2.1 Renderable::Renderable ( ) `[inline]`

The base constructor just sets a flag on the Renderable to not delete it, since that would be kind of a buzzkill out of the gate.

Parameters

_deleteMe	

Definition at line 49 of file Renderable.h.

2.54.2.2 virtual Renderable::∼Renderable ( ) [inline], [virtual]

Abstract base class needs a virtual destructor.

Definition at line 54 of file Renderable.h.

2.54.3 Member Function Documentation

2.54.3.1 virtual void Renderable::Update (float *dt*) [inline], [virtual]

This function gets called once per frame. Any game logic for an Actor should be done in this function, since it provides you with a dt for controlling rate of movement, animation, etc.

Parameters

dt	The amount of time (in seconds) that has elapsed since the last frame.

Reimplemented in Actor, GridActor, Camera, GameManager, ParticleActor, MobileSimulator, and Sentient.

Definition at line 64 of file Renderable.h.

2.54.3.2 virtual void Renderable::Render ( ) [inline], [virtual]

This function also gets called once per frame, **after** the Renderable::Update call. Anything you do in this function should be strictly related to drawing something on the screen. Moving it, changing its appearance/properties/etc. should happen in Update.

Reimplemented in Actor, GridActor, Camera, TextActor, ParticleActor, GameManager, HUDActor, and Sentient.

Definition at line 72 of file Renderable.h.

2.54.3.3 void Renderable::Destroy ( ) [inline]

A safe way to kill off a Renderable – it will be removed from the world and deleted from memory at the end of the current Update loop.

The protected Renderable::PreDestroy function is guaranteed to be called exactly once by this function, even if you call it many times.

Definition at line 81 of file Renderable.h.

2.54.3.4 bool Renderable::IsDestroyed ( ) [inline]

Used by the World to see if a Renderable is flagged to be deleted.

Returns

True if the Renderable should be removed and deleted.

Definition at line 94 of file Renderable.h.

2.54.3.5 int Renderable::GetLayer ( ) [inline]

The layer to which this Renderable has been assigned in the world. returns garbage if it hasn't been added to the World yet. (Negative layers are valid.)

Returns

The world layer for this Renderable

Definition at line 103 of file Renderable.h.

2.54.3.6 virtual void Renderable::PreDestroy ( ) [inline], [protected], [virtual]

Will get called before this Renderable is destroyed (if you do it via the Destroy function, obviously). Override this function in a subclass to do any cleanup work that is appropriate to your element.

Definition at line 110 of file Renderable.h.

The documentation for this class was generated from the following file:

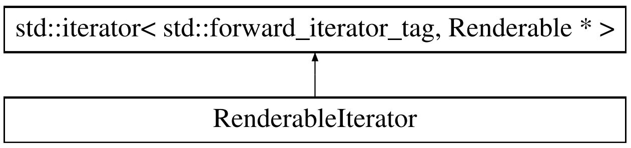
- Infrastructure/Renderable.h

2.55 RenderableIterator Class Reference

An iterator class to access Renderables in the World.

`#include <RenderableIterator.h>`

Inheritance diagram for RenderableIterator:



Public Member Functions

- Renderable ∗ **operator∗** ()
- RenderableIterator & **begin** ()
- RenderableIterator & **end** ()
- bool **operator!=** (const RenderableIterator &iter) const
- RenderableIterator & **erase** (RenderableIterator &item_to_remove)
- const RenderableIterator & **operator++** ()

2.55.1 Detailed Description

This class is an iterator that gives access to all the Renderables that have been added to the World. Access is sequential, going through one layer at at a time, but current ordering behavior shouldn't be relied on.

Definition at line 43 of file RenderableIterator.h.

The documentation for this class was generated from the following files:

- Infrastructure/RenderableIterator.h
- Infrastructure/RenderableIterator.cpp

2.56 SearchInterface Struct Reference

Public Member Functions

- **SearchInterface** (SpatialGraphKDNode ∗node)

- float **GoalDistanceEstimate** (const SearchInterface &goal)
- bool **IsGoal** (const SearchInterface &goal)
- bool **IsSameState** (const SearchInterface &goal)
- bool **GetSuccessors** (AStarSearch< SearchInterface > ∗pSearch, SearchInterface ∗pParent)
- float **GetCost** (const SearchInterface &successor)

Public Attributes

- SpatialGraphKDNode ∗ **pNode**

2.56.1 Detailed Description

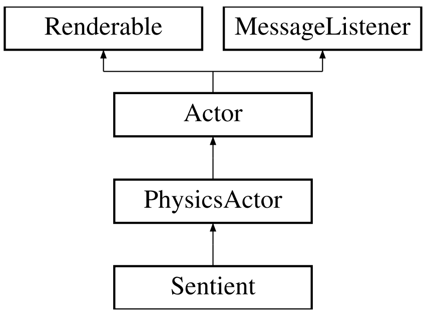
Definition at line 671 of file SpatialGraph.cpp.

The documentation for this struct was generated from the following file:

- AI/SpatialGraph.cpp

2.57 Sentient Class Reference

Inheritance diagram for Sentient:



Public Member Functions

- virtual void Update (float dt)
- virtual void Render ()
- PathFinder & **GetPathfinder** ()
- virtual void **OnNamedEvent** (const String &)
- virtual void **InitializeBrain** ()
- virtual void **StartBrain** ()

Protected Attributes

- PathFinder **_pathFinder**
- AIBrain **_brain**

Additional Inherited Members

2.57.1 Detailed Description

Definition at line 37 of file Sentient.h.

2.57.2 Member Function Documentation

2.57.2.1 void Sentient::Update (float *dt*) `[virtual]`

A function which makes the necessary updates to the Actor. The base implementation just updates the animations and intervals, but a subclass override can perform whatever extra magic is necessary. Make sure to call the base class's Update if you subclass.

Parameters

dt	The amount of time that's elapsed since the beginning of the last frame.

Reimplemented from Actor.

Definition at line 42 of file Sentient.cpp.

2.57.2.2 void Sentient::Render ( ) `[virtual]`

A function to draw the Actor to the screen. By default, this does the basic drawing based on the texture, color, shape, size, position, and rotation that have been applied to the Actor. Can be overridden in a subclass if necessary.

This will get called on every Actor once per frame, after the Update.

Reimplemented from Actor.

Definition at line 48 of file Sentient.cpp.

The documentation for this class was generated from the following files:

- AI/Sentient.h
- AI/Sentient.cpp

2.58 SoundDevice Class Reference

Our (very simple) sound system.

`#include <SoundDevice.h>`

Public Member Functions

- AngelSampleHandle LoadSample (const String &filename, bool isStream)
- AngelSoundHandle PlaySound (AngelSampleHandle sample, float volume=1.0f, bool looping=false, int flags=0)
- void StopSound (AngelSoundHandle sound)
- void PauseSound (AngelSoundHandle sound, bool paused)
- bool IsPlaying (AngelSoundHandle sound)
- bool IsPaused (AngelSoundHandle sound)
- void SetPan (AngelSoundHandle sound, float newPan)
- void SetVolume (AngelSoundHandle sound, float newVolume)
- void SetSoundCallback (GameManager ∗instance, void(GameManager::∗function)(AngelSoundHandle param))
- void Update ()
- void Shutdown ()

Static Public Member Functions

- static SoundDevice & GetInstance ()

2.58.1 Detailed Description

Our sound system is pretty much just a light wrapper around FMOD and OpenAL. We don't expose all of their functionality, but if you're interested in more advanced usage, it shouldn't be hard to expand this class.

For more information on FMOD: `http://www.fmod.org/`

For more information on OpenAL: `http://connect.creativelabs.com/openal/default.aspx`

Note that FMOD requires licensing fees if you want to distribute your game for money. Because we're focused on prototyping, and FMOD supports a wide array of sound formats while producing very high quality audio, it's the default sound system in Angel. If it doesn't fit your needs for some reason, though, no worries. This same interface can play sound through OpenAL, but only in the Ogg Vorbis format.

To switch to OpenAL, set the ANGEL_DISABLE_FMOD flag in AngelConfig.h to 1.

This class uses the singleton pattern; you can't actually declare a new instance of a SoundDevice. To access sound in your world, use "theSound" to retrieve the singleton object. "theSound" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Definition at line 98 of file SoundDevice.h.

2.58.2 Member Function Documentation

2.58.2.1 SoundDevice & SoundDevice::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theSound".

Returns

The singleton

Definition at line 113 of file SoundDevice.cpp.

2.58.2.2 AngelSampleHandle SoundDevice::LoadSample (const String & *filename*, bool *isStream*)

Loads a sound file from disk and gets it ready to be played by the system. Depending on how many sounds you're loading at once and/or how large they are, this could cause a hiccup. It's best to call this in advance of when you actually want to play the sound.

Parameters

filename	The path to the file you want to load
isStream	Whether or not the sound should stream or get loaded all at once

Returns

The AngelSampleHandle that you should hold on to for when you actually want to play the sound. You'll pass this to SoundDevice::PlaySound.

Definition at line 289 of file SoundDevice.cpp.

2.58.2.3 AngelSoundHandle SoundDevice::PlaySound (AngelSampleHandle *sample*, float *volume* = `1.0f`, bool *looping* = `false`, int *flags* = `0`)

Plays a sound that has been previously loaded.

Parameters

sample	The AngelSampleHandle that you got from the SoundDevice::LoadSample function
volume	The desired loudness of the sound, as a multiplier of its normal volume. (1.0 is maximum volume.)
looping	Whether you want the sound to repeat when it's done
flags	Currently unused; this will eventually let you pass in flags to the underlying sound system (only supported with FMOD)

Returns

The AngelSoundHandle that you can use to monitor or affect playback

Definition at line 436 of file SoundDevice.cpp.

2.58.2.4 void SoundDevice::StopSound (AngelSoundHandle *sound*)

Stops a sound that is currently playing.

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound

Definition at line 575 of file SoundDevice.cpp.

2.58.2.5 void SoundDevice::PauseSound (AngelSoundHandle *sound,* bool *paused*)

Pauses a sound that is currently playing. Its playback can be resumed from the same point later.

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound
paused	If true, the sound will be paused. If false, it will be unpaused.

Definition at line 588 of file SoundDevice.cpp.

2.58.2.6 bool SoundDevice::IsPlaying (AngelSoundHandle *sound*)

Find out whether or not a sound is still playing

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound

Returns

True if the sound is still playing, false if it's not

Definition at line 607 of file SoundDevice.cpp.

2.58.2.7 bool SoundDevice::IsPaused (AngelSoundHandle *sound*)

Find out whether a sound is paused

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound

Returns

True if the sound is paused, false if it's playing or stopped

Definition at line 624 of file SoundDevice.cpp.

2.58.2.8 void SoundDevice::SetPan (AngelSoundHandle *sound,* float *newPan*)

Change the stereo positioning of a sound while it's playing.

NB: If you've disabled FMOD (and are thus using the OpenAL backend), **only mono sounds** will pan properly.

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound
newPan	The new pan value. Should range from -1.0 (full left) to 1.0 (full right)

Definition at line 641 of file SoundDevice.cpp.

2.58.2.9 void SoundDevice::SetVolume (AngelSoundHandle *sound,* float *newVolume*)

Change the volume of a sound while it's playing

Parameters

sound	The handle to the playing sound, gotten from SoundDevice::PlaySound
newVolume	The new volume level. Should range from 0.0 (silent) to 1.0 (full volume)

Definition at line 654 of file SoundDevice.cpp.

2.58.2.10 void SoundDevice::SetSoundCallback (GameManager ∗ *instance,* void(GameManager::∗)(AngelSoundHandle param) *function*) [inline]

If you set a callback here, it will be executed whenever a sound finishes playing. If you're doing a music-or-sound-intensive game, this can be very important to you.

Note that the callback you're passing must be a member function of a GameManager.

Parameters

instance	The GameManager instance on which to execute the function
void	The function to execute, which will be passed the AngelSoundHandle

Definition at line 202 of file SoundDevice.h.

2.58.2.11 void SoundDevice::Update ( )

Calls the underlying FMOD update function to keep the sound thread chugging along appropriately.

NB: Must be called once (and only once) per frame. The World already calls this function appropriately, so you should only be calling it if you really know what you're doing.

Definition at line 507 of file SoundDevice.cpp.

2.58.2.12 void SoundDevice::Shutdown ( )

Releases all sounds (invalidating your leftover AngelSoundHandle and AngelSampleHandle pointers) and shuts down FMOD if necessary. Should really only be called at the end of the game, which the World handles for you by default.

Definition at line 233 of file SoundDevice.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/SoundDevice.h
- Infrastructure/SoundDevice.cpp

2.59 SpatialGraph Class Reference

Public Member Functions

- **SpatialGraph** (float entityWidth, const BoundingBox &startBox)
- SpatialGraphKDNode ∗ **FindNode** (SpatialGraphKDNode ∗node, const BoundingBox &bbox)
- SpatialGraphKDNode ∗ **FindNode** (SpatialGraphKDNode ∗node, const Vector2 &point)
- SpatialGraphKDNode ∗ **FindNode** (const BoundingBox &bbox)
- SpatialGraphKDNode ∗ **FindNode** (const Vector2 &point)
- void **Render** ()
- int **GetDepth** ()
- Vector2 **GetSmallestDimensions** ()
- bool **CanGo** (const Vector2 &vFrom, const Vector2 vTo)

2.59.1 Detailed Description

Definition at line 85 of file SpatialGraph.h.

The documentation for this class was generated from the following files:

- AI/SpatialGraph.h
- AI/SpatialGraph.cpp

2.60 SpatialGraphKDNode Class Reference

Public Member Functions

- **SpatialGraphKDNode** (const BoundingBox &bb, SpatialGraphKDNode ∗_parent)
- void **Render** ()
- bool **HasChildren** ()
- void **GetGridPoints** (Vector2List &points, int &xPoints, int &yPoints)

Public Attributes

- BoundingBox **BBox**
- SpatialGraphKDNode ∗ **LHC**
- SpatialGraphKDNode ∗ **RHC**
- SpatialGraphKDNode ∗ **Parent**
- SpatialGraph ∗ **Tree**
- int **Index**
- int **Depth**
- bool **bBlocked**
- SpatialGraphNeighborList **Neighbors**
- BoolList **NeighborLOS**

2.60.1 Detailed Description

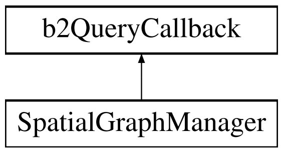
Definition at line 42 of file SpatialGraph.h.

The documentation for this class was generated from the following files:

- AI/SpatialGraph.h
- AI/SpatialGraph.cpp

2.61 SpatialGraphManager Class Reference

Inheritance diagram for SpatialGraphManager:



Public Member Functions

- bool **ReportFixture** (b2Fixture ∗fixture)
- SpatialGraph ∗ **GetGraph** ()
- void **CreateGraph** (float entityWidth, const BoundingBox &bounds)
- void **Render** ()
- bool **GetPath** (const Vector2 &source, const Vector2 &dest, Vector2List &path)
- bool **CanGo** (const Vector2 &from, const Vector2 to)
- bool **IsInPathableSpace** (const Vector2 &point)
- bool **FindNearestNonBlocked** (const Vector2 &fromPoint, Vector2 &goTo)
- void **EnableDrawBounds** (bool enable)
- const bool **ToggleDrawBounds** ()
- const bool **GetDrawBounds** ()
- void **EnableDrawBlocked** (bool enable)
- const bool **ToggleDrawBlocked** ()
- const bool **GetDrawBlocked** ()
- void **EnableDrawGridPoints** (bool enable)
- const bool **ToggleDrawGridPoints** ()
- const bool **GetDrawGridPoints** ()
- void **EnableDrawGraph** (bool enable)
- const bool **ToggleDrawGraph** ()
- const bool **GetDrawGraph** ()
- void **EnableDrawNodeIndex** (bool enable)
- const bool **ToggleDrawNodeIndex** ()
- const bool **GetDrawNodeIndex** ()

Static Public Member Functions

- static SpatialGraphManager & **GetInstance** ()

Protected Member Functions

- void **Initialize** ()

Static Protected Attributes

- static SpatialGraphManager ∗ **s_SpatialGraphManager** = NULL

2.61.1 Detailed Description

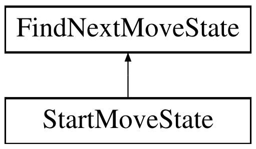
Definition at line 122 of file SpatialGraph.h.

The documentation for this class was generated from the following files:

- AI/SpatialGraph.h
- AI/SpatialGraph.cpp

2.62 StartMoveState Class Reference

Inheritance diagram for StartMoveState:



Public Member Functions

- virtual const char ∗ **GetName** ()
- virtual bool **Update** (PathFinderMove &move)

Additional Inherited Members

2.62.1 Detailed Description

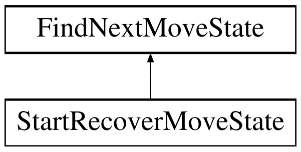
Definition at line 115 of file PathFinder.cpp.

The documentation for this class was generated from the following file:

- AI/PathFinder.cpp

2.63 StartRecoverMoveState Class Reference

Inheritance diagram for StartRecoverMoveState:



Public Member Functions

- virtual const char ∗ **GetName** ()
- virtual bool **Update** (PathFinderMove &move)

Additional Inherited Members

2.63.1 Detailed Description

Definition at line 263 of file PathFinder.cpp.

The documentation for this class was generated from the following file:

- AI/PathFinder.cpp

2.64 Switchboard Class Reference

The central class which handles delivery of Messages.

`#include <Switchboard.h>`

Public Member Functions

- void Broadcast (Message ∗message)
- void DeferredBroadcast (Message ∗message, float delay)
- void Update (float dt)
- const bool SubscribeTo (MessageListener ∗subscriber, const String &messageType)
- const bool UnsubscribeFrom (MessageListener ∗subscriber, const String &messageType)
- const std::set< MessageListener ∗ > GetSubscribersTo (const String &messageName)
- const StringSet GetSubscriptionsFor (MessageListener ∗subscriber)
- void SendAllMessages ()

Static Public Member Functions

- static Switchboard & GetInstance ()

Static Protected Attributes

- static Switchboard ∗ **s_Switchboard** = NULL

2.64.1 Detailed Description

This class is where all Messages pass through to get to their subscribers. It manages subscribers lists, delivery, and broadcast of messages.

Like the World, it uses the singleton pattern; you can't actually declare a new instance of a Switchboard. To access messaging in your world, use "theSwitchboard" to retrieve the singleton object. "theSwitchboard" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Definition at line 55 of file Switchboard.h.

2.64.2 Member Function Documentation

2.64.2.1 Switchboard & Switchboard::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theSwitchboard".

Returns

The singleton

Definition at line 42 of file Switchboard.cpp.

2.64.2.2 void Switchboard::Broadcast (**Message** ∗ *message*)

Send a Message to all the MessageListeners who have subscribed to Messages of that particular name. All Messages are sent at the end of the current frame, outside the Update loop. (Which means you can safely remove objects from the World in response to a Message.)

Parameters

message	The message to send

Definition at line 51 of file Switchboard.cpp.

2.64.2.3 void Switchboard::DeferredBroadcast (**Message** ∗ *message*, float *delay*)

Lets you send a Message after a designated delay. Oftentimes you want something to happen a little while *after* an event, and it can be be easier to simply defer the sending of the Message rather than make the MessageListener responsible for implementing the delay.

Parameters

message	The message to send
delay	Amount of time (in seconds) to wait before sending

Definition at line 56 of file Switchboard.cpp.

2.64.2.4 void Switchboard::Update (float *dt*)

Takes the same form as the Renderable::Update function, but Switchboard is not a Renderable. This function gets called by the World to let the Switchboard know about the passage of time so it can manage deferred broadcasts.

Parameters

dt	The amount of time elapsed since the last frame

Definition at line 62 of file Switchboard.cpp.

2.64.2.5 const bool Switchboard::SubscribeTo (**MessageListener** ∗ *subscriber*, const String & *messageType*)

Sign a MessageListener up to receive notifications when Messages of a specific class are broadcast through the Switchboard.

Parameters

subscriber	The MessageListener to sign up
messageType	The name of the Message it's interested in

Returns

True if the MessageListener was successfully subscribed – could be false if the subscription was attempted while messages were being delivered (in which case the subscription will start when this round of delivery is done) or if the MessageListener was already subscribed to Messages of that name.

Definition at line 80 of file Switchboard.cpp.

2.64.2.6 const bool Switchboard::UnsubscribeFrom (**MessageListener** $*$ *subscriber,* const String & *messageType*)

Lets a MessageListener stop receiving notifications of specific name. MessageListeners automatically unsubscribe from all their Messages when their destructors are called, so you don't have to worry about this when destroying an object; this would only be called directly in user code when you no longer care about a particular Message.

Parameters

subscriber	The MessageListener that doesn't want to get these Messages anymore
messageType	The name of the Message they're tired of hearing about

Returns

True if the MessageListener was successfully unsubscribed – could be false if the unsubscription was attempted while messages were being delivered (in which case the subscription will be removed when this round of delivery is done) or if the MessageListener was not subscribed to Messages of that name.

Definition at line 93 of file Switchboard.cpp.

2.64.2.7 const std::set$<$ **MessageListener** $*$ $>$ Switchboard::GetSubscribersTo (const String & *messageName*)

Get a list of all MessageListeners subscribed to Messages with a given name.

Parameters

messageName	The Message you care about

Returns

A std::set of objects subscribed

Definition at line 120 of file Switchboard.cpp.

2.64.2.8 const StringSet Switchboard::GetSubscriptionsFor (**MessageListener** $*$ *subscriber*)

Get a list of all Message subscriptions for a certain MessageListener

Parameters

subscriber	The MessageListener you care about

Returns

A StringSet of all their subscriptions

Definition at line 132 of file Switchboard.cpp.

2.64.2.9 void Switchboard::SendAllMessages ( )

Immediately sends all Messages to the appropriate subscribers. Called by the World at the end of each frame; you shouldn't call this directly in your game code.

Definition at line 144 of file Switchboard.cpp.

The documentation for this class was generated from the following files:

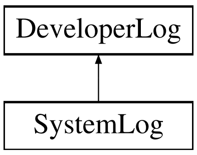
- Messaging/Switchboard.h
- Messaging/Switchboard.cpp

2.65 SystemLog Class Reference

A log which writes to standard output (rather than the in-game console)

`#include <Log.h>`

Inheritance diagram for SystemLog:



Public Member Functions

- virtual void Log (const String &val)

2.65.1 Detailed Description

This type of log writes its output straight to the program's standard output. On Windows, it will output to the output pane of Visual Studio (easier to see while you're debugging).

Definition at line 127 of file Log.h.

2.65.2 Member Function Documentation

2.65.2.1 void SystemLog::Log (const String & *val*) `[virtual]`

The string to be logged to the output.

Parameters

val	The string

Implements DeveloperLog.

Definition at line 146 of file Log.cpp.

The documentation for this class was generated from the following files:

- Infrastructure/Log.h
- Infrastructure/Log.cpp

2.66 TagCollection Class Reference

A helper class that manages the tags you can set on Actors.

`#include <TagCollection.h>`

Public Member Functions

- ActorSet GetObjectsTagged (String findTag)
- StringSet GetTagList ()
- void AddObjToTagList (Actor ∗obj, const String &tag)
- void RemoveObjFromTagList (Actor ∗obj, const String &tag)

Static Public Member Functions

- static TagCollection & GetInstance ()

Static Protected Attributes

- static TagCollection ∗ **s_TagCollection** = NULL

2.66.1 Detailed Description

Whenever you call Actor::Tag or Actor::Untag, the Actor manipulates the tag collection appropriately for you. The only functions here you should be thinking about are TagCollection::GetObjectsTagged and TagCollection::GetTagList.

Like the Camera and the World, it uses the singleton pattern; you can't actually declare a new instance of a TagCollection. To access tags in your world, use "theTagList" to retrieve the singleton object. "theTagList" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Definition at line 53 of file TagCollection.h.

2.66.2 Member Function Documentation

2.66.2.1 TagCollection & TagCollection::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theTagList".

Returns

The singleton

Definition at line 39 of file TagCollection.cpp.

2.66.2.2 ActorSet TagCollection::GetObjectsTagged (String *findTag*)

Returns the set of all Actors who have the given tag.

Parameters

findTag	The tag of interest

Returns

An ActorSet of everyone who is tagged thusly

Definition at line 53 of file TagCollection.cpp.

2.66.2.3 StringSet TagCollection::GetTagList ( )

Get a list of all the tags currently in use.

Returns

All current tags as a StringSet

Definition at line 111 of file TagCollection.cpp.

2.66.2.4 void TagCollection::AddObjToTagList (Actor ∗ *obj*, const String & *tag*)

Adds a tag to a given Actor. Shouldn't be called directly; use the Actor::Tag function.

Parameters

obj	The Actor to tag
tag	The tag to apply

Definition at line 125 of file TagCollection.cpp.

2.66.2.5 void TagCollection::RemoveObjFromTagList (Actor ∗ *obj*, const String & *tag*)

Removes a tag from an Actor. Shouldn't be called directly; use the Actor::Untag function.

Parameters

obj	The Actor to untag
tag	The tag to remove

Definition at line 130 of file TagCollection.cpp.

The documentation for this class was generated from the following files:

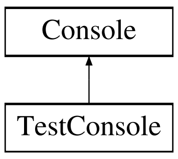
- Infrastructure/TagCollection.h
- Infrastructure/TagCollection.cpp

2.67 TestConsole Class Reference

An example Console implementation.

`#include <Console.h>`

Inheritance diagram for TestConsole:



Public Member Functions

- TestConsole ()
- virtual void Execute (String input)
- virtual StringList GetCompletions (const String &input)

Additional Inherited Members

2.67.1 Detailed Description

A degenerate case of a Console; does nothing but echo the user input back into the screen log.

Definition at line 220 of file Console.h.

2.67.2 Constructor & Destructor Documentation

2.67.2.1 TestConsole::TestConsole ( ) `[inline]`

Sets up the prompt (::>);

Definition at line 226 of file Console.h.

2.67.3 Member Function Documentation

2.67.3.1 virtual void TestConsole::Execute (String *input*) `[inline], [virtual]`

Echos the input right back out.

Parameters

input	The string to echo

Implements Console.

Definition at line 236 of file Console.h.

2.67.3.2 virtual StringList TestConsole::GetCompletions (const String & *input*) `[inline], [virtual]`

Returns an empty StringList.

Parameters

input	The input to be ignored

Returns

An empty list

Implements Console.

Definition at line 244 of file Console.h.

The documentation for this class was generated from the following file:

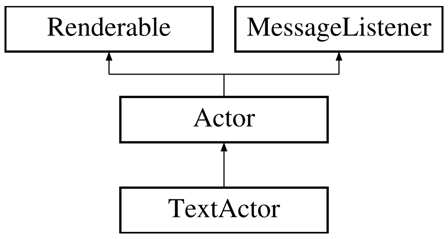
- Infrastructure/Console.h

2.68 TextActor Class Reference

An Actor for displaying text on the screen.

`#include <TextActor.h>`

Inheritance diagram for TextActor:



Public Member Functions

- TextActor (const String &fontNickname="Console", const String &displayString="", TextAlignment align=TXT-_Left, int lineSpacing=5)
- virtual void Render ()
- const String & GetFont () const
- void SetFont (const String &newFont)
- const String & GetDisplayString () const
- void SetDisplayString (const String &newString)
- const TextAlignment GetAlignment ()
- void SetAlignment (TextAlignment newAlignment)
- const int GetLineSpacing ()
- void SetLineSpacing (int newSpacing)
- virtual void SetPosition (float x, float y)
- virtual void SetPosition (const Vector2 &position)
- virtual void SetRotation (float newRotation)
- virtual void ReceiveMessage (Message ∗message)
- const BoundingBox & GetBoundingBox () const
- virtual const String GetClassName () const

Additional Inherited Members

2.68.1 Detailed Description

A TextActor handles drawing text to the screen using world coordinates. The basis text rendering functions operate in screen-space, which can be annoying if you want to have debug data or labels for your Actors.

In addition, a TextActor supports wraps up more functionality than the simple text rendering, allowing varying alignments, newlines, etc.

Definition at line 54 of file TextActor.h.

2.68.2 Constructor & Destructor Documentation

2.68.2.1 TextActor::TextActor (const String & *fontNickname* = `"Console"`, const String & *displayString* = `""`, TextAlignment *align* = `TXT_Left`, int *lineSpacing* = `5`)

The constructor sets up all the textual information that this TextActor will use to draw itself to the screen.

Parameters

fontNickname	the name of the font to be used when drawing this TextActor. The name of the font is set with the RegisterFont function. By default it uses the monospaced font used in the Console, at 24 points.
displayString	the actual text to be displayed
align	the desired alignment of the text
lineSpacing	the amount of space (in pixels) between each line of text

Definition at line 40 of file TextActor.cpp.

2.68.3 Member Function Documentation

2.68.3.1 void TextActor::Render ( ) `[virtual]`

Override of the Renderable::Render function to draw text

Reimplemented from Actor.

Definition at line 55 of file TextActor.cpp.

2.68.3.2 const String & TextActor::GetFont ( ) const

Get the name of the font currently being used for this Actor

Returns

the nickname of the current font, as set in RegisterFont

Definition at line 64 of file TextActor.cpp.

2.68.3.3 void TextActor::SetFont (const String & *newFont*)

Change the font used in drawing this TextActor.

Parameters

newFont	the nickname of the new font. This must be set up with RegisterFont before being set here – the TextActor will not draw at all if this is an invalid nickname.

Definition at line 69 of file TextActor.cpp.

2.68.3.4 const String & TextActor::GetDisplayString ( ) const

Get the current string being drawn by this TextActor.

Returns

the display string

Definition at line 75 of file TextActor.cpp.

2.68.3.5 void TextActor::SetDisplayString (const String & *newString*)

Change the string to be drawn by this TextActor. Newlines can be delimited with the plain newline character (\n).

Parameters

newString	the string that should be drawn by this TextActor in the next frame

Definition at line 80 of file TextActor.cpp.

2.68.3.6 const TextAlignment TextActor::GetAlignment ( )

Get the current alignment being used by this TextActor

Returns

the text alignment (TXT_Left, TXT_Right, or TXT_Center)

Definition at line 97 of file TextActor.cpp.

2.68.3.7 void TextActor::SetAlignment (TextAlignment *newAlignment*)

Change the alignment of this TextActor.

Parameters

newAlignment	the desigred alignment, as described by the enum

Definition at line 102 of file TextActor.cpp.

2.68.3.8 const int TextActor::GetLineSpacing ( )

Get the amount of space between each line that this TextActor draws

Returns

the amount of space (in pixels) that goes between each line

Definition at line 108 of file TextActor.cpp.

2.68.3.9 void TextActor::SetLineSpacing (int *newSpacing*)

Change the line spacing of this TextActor

Parameters

newSpacing	the desired amount of space (in pixels) between each line

Definition at line 113 of file TextActor.cpp.

2.68.3.10 void TextActor::SetPosition (float *x*, float *y*) `[virtual]`

An override of the SetPosition function, since we need to update the screen space drawing coordinates of our text drawing when the TextActor moves in the World.

Parameters

x	the X coordinate (in GL units) of the TextActor
y	the Y coordinate (in GL units) of the TextActor

Reimplemented from Actor.

Definition at line 119 of file TextActor.cpp.

2.68.3.11 void TextActor::SetPosition (const Vector2 & *position*) `[virtual]`

An override of the SetPosition function, since we need to update the screen space drawing coordinates of our text drawing when the TextActor moves in the World.

Parameters

position	a Vector2 indicating the X and Y coordinates of the TextActor (in GL units)

Reimplemented from Actor.

Definition at line 131 of file TextActor.cpp.

2.68.3.12 void TextActor::SetRotation (float *newRotation*) `[virtual]`

An override of the SetRotation function, since we need to update the screen space drawing coordinates of our text drawing when the TextActor moves in the World.

Parameters

position	a float indicating the new rotation counter-clockwise around the z-axis (in degrees)

Reimplemented from Actor.

Definition at line 125 of file TextActor.cpp.

2.68.3.13 void TextActor::ReceiveMessage (**Message** ∗ *message*) `[virtual]`

An implementation of the MessageListener interface, which listens for CameraChange messages and responds appropriately.

See Also

MessageListener

Parameters

message	The message getting delivered.

Reimplemented from Actor.

Definition at line 137 of file TextActor.cpp.

2.68.3.14 const **BoundingBox** & TextActor::GetBoundingBox ( ) const

Since TextActors use their own methods of drawing, it can be difficult to determine what space they're covering. If you want to do any kind of logic dealing with TextActor overlap, this is the way to get the BoundingBox information.

Returns

A BoundingBox describing the area covered by the TextActor in GL units (not pixels).

Definition at line 145 of file TextActor.cpp.

2.68.3.15 virtual const String TextActor::GetClassName ( ) const `[inline]`, `[virtual]`

Used by the SetName function to create a basename for this class. Overridden from Actor::GetClassName.

Returns

The string "TextActor"

Reimplemented from Actor.

Definition at line 191 of file TextActor.h.

The documentation for this class was generated from the following files:

- Actors/TextActor.h
- Actors/TextActor.cpp

2.69 TextureCacheEntry Struct Reference

Public Attributes

- String **filename**
- GLint **clampMode**
- GLint **filterMode**
- GLuint **textureIndex**
- GLuint **width**
- GLuint **height**
- bool **dirty**

2.69.1 Detailed Description

Definition at line 70 of file Textures.cpp.

The documentation for this struct was generated from the following file:

- Infrastructure/Textures.cpp

2.70 TGenericCallback< ClassInstance, ParamType > Class Template Reference

(Internal) A callback template to simplify storage/retrieval of method pointers

```
#include <Callback.h>
```

Public Types

- typedef void(ClassInstance::∗ **FunctionPointer**)(ParamType param)

Public Member Functions

- virtual void **Execute** (ParamType param) const
- void **SetCallback** (ClassInstance ∗instance, FunctionPointer function)
- const ClassInstance ∗ **GetInstance** ()
- const FunctionPointer **GetFunction** ()

2.70.1 Detailed Description

template<class ClassInstance, class ParamType>class TGenericCallback< ClassInstance, ParamType >

Storing pointers to member functions (as opposed to just loose functions) can be complicated and error-prone. This template class is designed to make such things simpler. An example usage is in the SoundDevice class.

Note that this template only lets you store callbacks that take a single parameter.

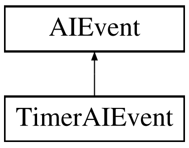
Definition at line 43 of file Callback.h.

The documentation for this class was generated from the following file:

- Infrastructure/Callback.h

2.71 TimerAIEvent Class Reference

Inheritance diagram for TimerAIEvent:



Public Member Functions

- virtual TimerAIEvent ∗ **Initialize** (float duration)
- virtual void **Update** (float dt)

Protected Attributes

- float **_duration**

Additional Inherited Members

2.71.1 Detailed Description

Definition at line 34 of file TimerAIEvent.h.

The documentation for this class was generated from the following files:

- AIEvents/TimerAIEvent.h
- AIEvents/TimerAIEvent.cpp

2.72 Touch Struct Reference

Struct to represent an individual touch in a multi-touch system.

```
#include <MultiTouch.h>
```

Public Attributes

- Vec2i **StartingPoint**
- Vec2i **CurrentPoint**
- float **StartTime**
- float **MotionStartTime**
- void ∗ **__platformTouch**

2.72.1 Detailed Description

Definition at line 37 of file MultiTouch.h.

The documentation for this struct was generated from the following file:

- Input/MultiTouch.h

2.73 TouchListener Class Reference

An abstract interface for getting mouse events.

```
#include <MultiTouch.h>
```

Public Member Functions

- TouchListener ()
- virtual ∼TouchListener ()
- virtual void TouchMotionEvent (Touch ∗movedTouch)
- virtual void TouchEndEvent (Touch ∗endedTouch)
- virtual void TouchStartEvent (Touch ∗startedTouch)

Static Public Member Functions

- static TouchList & GetTouchList ()

2.73.1 Detailed Description

This is an abstract base class which provides an interface to get notifications about what's going on with the multi-touch system. If you want to get touch data, derive a class from this one and implement the Touch∗Event member functions.

Definition at line 91 of file MultiTouch.h.

2.73.2 Constructor & Destructor Documentation

2.73.2.1 TouchListener::TouchListener ( )

Base constructor adds this object to the list of objects to get notified when touchey things happen.

Definition at line 43 of file MultiTouch.cpp.

2.73.2.2 TouchListener::∼TouchListener ( ) `[virtual]`

Base destructor removes this object from the list.

Definition at line 48 of file MultiTouch.cpp.

2.73.3 Member Function Documentation

2.73.3.1 void TouchListener::TouchMotionEvent (Touch ∗ *movedTouch*) `[virtual]`

Called whenever an existing touch moves.

Parameters

movedTouch	A pointer to a Touch object representing the specific touch that moved.

Definition at line 54 of file MultiTouch.cpp.

2.73.3.2 void TouchListener::TouchEndEvent (Touch ∗ *endedTouch*) `[virtual]`

Called whenever a touch ends.

Parameters

endedTouch	A pointer to a Touch object representing the specific touch that just ended.

Definition at line 55 of file MultiTouch.cpp.

2.73.3.3 void TouchListener::TouchStartEvent (Touch ∗ *startedTouch*) `[virtual]`

Called whenever a new touch starts.

Parameters

startedTouch	A pointer to a Touch object representing the specific touch that just started.

Definition at line 56 of file MultiTouch.cpp.

2.73.3.4 TouchList & TouchListener::GetTouchList ( ) `[static]`

A static function to get access to a list of the current Touch objects being detected.

Definition at line 58 of file MultiTouch.cpp.

The documentation for this class was generated from the following files:

- Input/MultiTouch.h
- Input/MultiTouch.cpp

2.74 Traversal Class Reference

Public Member Functions

- virtual void **StartTraversal** (const Vector2 &vStartPoint, int maxResults=-1, int maxIterations=-1)
- virtual bool **DoNextTraversal** ()
- virtual void **ExecuteFullTraversal** ()
- virtual std::vector< Vector2 > & **GetResults** ()

Protected Member Functions

- virtual void **EvaluateNode** (SpatialGraphKDNode ∗pNode)=0
- virtual void **AddSuccessors** (SpatialGraphKDNode ∗pCurrent)
- virtual void **AddNodeToVisit** (SpatialGraphKDNode ∗pNode)
- virtual bool **HasNodesToVist** ()
- virtual SpatialGraphKDNode ∗ **PopNextNode** ()
- virtual void **SetVisited** (SpatialGraphKDNode ∗pNode)
- virtual bool **WasVisited** (SpatialGraphKDNode ∗pNode)
- virtual void **ClearAllVisited** ()

2.74.1 Detailed Description

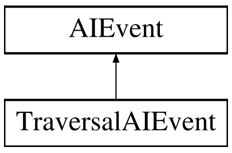
Definition at line 40 of file Traversal.h.

The documentation for this class was generated from the following files:

- AI/Traversal.h
- AI/Traversal.cpp

2.75 TraversalAIEvent Class Reference

Inheritance diagram for TraversalAIEvent:



Public Member Functions

- virtual TraversalAIEvent ∗ **Initialize** (Traversal ∗pTraversal, Vector2 &vStartPos, int numIterationsPerFrame=10, int maxResults=-1, int maxIterations=-1)
- virtual void **Update** (float dt)
- virtual void **Stop** ()
- virtual std::vector< Vector2 > & **GetResults** ()
- virtual std::vector< Vector2 > **CopyResults** ()

Additional Inherited Members

2.75.1 Detailed Description

Definition at line 38 of file TraversalAIEvent.h.

The documentation for this class was generated from the following files:

- AIEvents/TraversalAIEvent.h
- AIEvents/TraversalAIEvent.cpp

2.76 Tuning Class Reference

The class which handles getting and setting tuning variables.

`#include <TuningVariable.h>`

Public Member Functions

- StringSet GetVariables ()
- int GetInt (const String &name)
- float GetFloat (const String &name)
- String GetString (const String &name)
- Vector2 GetVector (const String &name)
- void SetInt (const String &name, int val)
- void SetFloat (const String &name, float val)
- void SetString (const String &name, const String &val)
- void SetVector (const String &name, const Vector2 &val)
- void AddToRuntimeTuningList (const String &varName)
- bool IsRuntimeTuned (const String &varName)

Static Public Member Functions

- static Tuning & GetInstance ()

2.76.1 Detailed Description

This class is used to handle housekeeping of variables you'll want to manually change at runtime to affect the way your game plays. A good example is the height a character can jump – you might declare this as a tuning variable in the `Config/tuning.ini`, change it from the console using the `tune()` function, and then call SaveTuningVariables() from the console to save the new value back into the file.

Some people find it helpful to bind certain tuning functions to keypresses while they're tuning, so you could increase or decrease a certain tuning variable with the arrow keys, for instance.

It's also possible to declare new tuning variables at runtime and do all sorts of crazy things. Go nuts!

Definition at line 54 of file TuningVariable.h.

2.76.2 Member Function Documentation

2.76.2.1 Tuning & Tuning::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theTuning".

Returns

The singleton

Definition at line 35 of file TuningVariable.cpp.

2.76.2.2 StringSet Tuning::GetVariables ( )

Get a list of the currently declared tuning variables. This only gives you a set of variable names; you'll have to look up the values if you want them.

Returns

The set of strings representing the variable names.

Definition at line 49 of file TuningVariable.cpp.

2.76.2.3 int Tuning::GetInt (const String & *name*)

Get an integer representation of a tuning variable. The engine will do its best to convert declared values into reasonable integer representations, and will return 0 if it can't or if the variable has not been declared.

Parameters

name	The name of the variable

Returns

Its integer representation

Definition at line 61 of file TuningVariable.cpp.

2.76.2.4 float Tuning::GetFloat (const String & *name*)

Get a floating point representation of a tuning variable. The engine will do its best to convert declared values into reasonable float representations, and will return 0.0f if it can't or if the variable has not been declared.

Parameters

name	The name of the variable

Returns

Its floating point representation

Definition at line 71 of file TuningVariable.cpp.

2.76.2.5 String Tuning::GetString (const String & *name*)

Get a string representation of a tuning variable. If the variable was declared as another type, you'll still get a string representation of the value. Returns an empty string if the variable has not been declared.

Parameters

name	The name of the variable

Returns

Its string representation

Definition at line 81 of file TuningVariable.cpp.

2.76.2.6 Vector2 Tuning::GetVector (const String & *name*)

Get a Vector2 representation of a tuning variable. This guy is pretty much the odd man out in terms of conversions, since there's no obvious default way to convert between the different types and a 2d vector. Returns a zero-length vector if you haven't actually set a vector to this variable or if the variable has not been declared.

Parameters

name	The name of the variable

Returns

Its Vector2 representation

Definition at line 91 of file TuningVariable.cpp.

2.76.2.7 void Tuning::SetInt (const String & *name*, int *val*)

Change a tuning variable's value to a new integer. The Set∗ functions are also how you declare new tuning variables.

Parameters

name	The name of the variable to change
val	The new integer value

Definition at line 101 of file TuningVariable.cpp.

2.76.2.8 void Tuning::SetFloat (const String & *name*, float *val*)

Change a tuning variable's value to a new float. The Set∗ functions are also how you declare new tuning variables.

Parameters

name	The name of the variable to change.
val	The new float value

Definition at line 106 of file TuningVariable.cpp.

2.76.2.9 void Tuning::SetString (const String & *name*, const String & *val*)

Change a tuning variable's value to a new string. The Set∗ functions are also how you declare new tuning variables.

Parameters

name	The name of the variable to change.
val	The new string value

Definition at line 111 of file TuningVariable.cpp.

2.76.2.10 void Tuning::SetVector (const String & *name*, const Vector2 & *val*)

Change a tuning variable's value to a new vector. The Set∗ functions are also how you declare new tuning variables.

Parameters

name	The name of the variable to change.
val	The new vector value

Definition at line 116 of file TuningVariable.cpp.

2.76.2.11 void Tuning::AddToRuntimeTuningList (const String & *varName*)

For internal usage – lets the engine keep track of what variables have been tuned at runtime so they don't get stomped by the auto-reloading of the `tuning.ini` file.

Parameters

varName	The name of the variable to add to the list.

Definition at line 121 of file TuningVariable.cpp.

2.76.2.12 bool Tuning::IsRuntimeTuned (const String & *varName*)

For internal usage – lets the engine keep check if a variable has been tuned at runtime so it doesn't get stomped when `tuning.ini` gets reloaded.

Parameters

varName	The name of the variable to check

Returns

Whether or not it's been tuned manually during this session

Definition at line 126 of file TuningVariable.cpp.

The documentation for this class was generated from the following files:

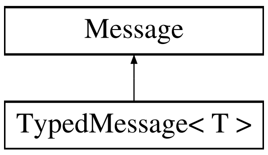
- Infrastructure/TuningVariable.h
- Infrastructure/TuningVariable.cpp

2.77 TypedMessage< T > Class Template Reference

A templated class for delivering additional information with Messages.

`#include <Message.h>`

Inheritance diagram for TypedMessage< T >:



Public Member Functions

- TypedMessage (const String &messageName, T value, MessageListener ∗sender=NULL)
- const T GetValue ()

Protected Attributes

- T **_value**

2.77.1 Detailed Description

template<class T>class TypedMessage< T >

This class lets you define your own message types that carry data with them. For instance, if you wanted to create a Message that carried a world location with it:

```
TypedMessage<Vector2> *m = new TypedMessage<Vector2>("
      SomethingHappenedHere", Vector2(-3, -2));
```

This gets sent through the Switchboard as just a pointer to the base class, though, so your MessageReceivers will have to parse what it is from the Message::GetMessageType function and cast it appropriately from there.

Definition at line 111 of file Message.h.

2.77.2 Constructor & Destructor Documentation

2.77.2.1 template<class T> TypedMessage< T >::TypedMessage (const String & *messageName*, T *value*, MessageListener ∗ *sender* = NULL) [inline]

Creates a new TypedMessage carrying the information you've designated

Parameters

messageName	The name of the message
value	Any data of the type defined in the template constructor
sender	Who is sending this message; NULL by default

Definition at line 121 of file Message.h.

2.77.3 Member Function Documentation

2.77.3.1 template<class T> const T TypedMessage< T >::GetValue ( ) [inline]

Get the data that this Message carries with it

Returns

Data of the type defined in the template constructor

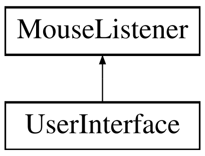
Definition at line 133 of file Message.h.

The documentation for this class was generated from the following file:

- Messaging/Message.h

2.78 UserInterface Class Reference

Inheritance diagram for UserInterface:



Public Member Functions

- void Render ()
- virtual void MouseMotionEvent (Vec2i screenCoordinates)
- virtual void MouseDownEvent (Vec2i screenCoordinates, MouseButtonInput button)
- virtual void MouseUpEvent (Vec2i screenCoordinates, MouseButtonInput button)
- virtual void MouseWheelEvent (const Vector2 &position)
- void HandleKey (int key, bool down)
- void HandleCharacter (wchar_t chr)
- void Shutdown ()
- AngelUIHandle AddButton (const String &label, Vec2i position, void(∗callback)(), bool center=false, const String &font="", Vec2i padding=Vec2i(10, 10))
- AngelUIHandle ShowChoiceBox (const String &choiceLabel, const StringList &labels, void(∗callback)(int), const String &font="", Vec2i padding=Vec2i(10, 10), bool modal=true)
- void RemoveUIElement (AngelUIHandle element)

Static Public Member Functions

- static UserInterface & GetInstance ()

Static Protected Attributes

- static UserInterface ∗ **s_UserInterface** = NULL

2.78.1 Detailed Description

Definition at line 51 of file UserInterface.h.

2.78.2 Member Function Documentation

2.78.2.1 UserInterface & UserInterface::GetInstance ( ) `[static]`

Used to access the singleton instance of this class. As a shortcut, you can just use "theUI".

Returns

The singleton

Definition at line 133 of file UserInterface.cpp.

2.78.2.2 void UserInterface::Render ( )

For internal engine use only; does the actual UI drawing.

Definition at line 182 of file UserInterface.cpp.

2.78.2.3 void UserInterface::MouseMotionEvent (Vec2i *screenCoordinates*) `[virtual]`

Called whenever the player moves the mouse.

Parameters

screenCoordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::ScreenToWorld function if you want GL units.

Reimplemented from MouseListener.

Definition at line 224 of file UserInterface.cpp.

2.78.2.4 void UserInterface::MouseDownEvent (Vec2i *screenCoordinates*, MouseButtonInput *button*) [virtual]

Called whenever the player presses down on a mouse button.

Parameters

screenCoordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::ScreenToWorld function if you want GL units.
button	Which button was pressed. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_RIGHT`.

Reimplemented from MouseListener.

Definition at line 240 of file UserInterface.cpp.

2.78.2.5 void UserInterface::MouseUpEvent (Vec2i *screenCoordinates*, MouseButtonInput *button*) [virtual]

Called whenever the player releases a mouse button.

Parameters

screenCoordinates	The new coordinates of the mouse in screen coordinates (pixels). Use the MathUtil::ScreenToWorld function if you want GL units.
button	Which button was released. Will be one of `MOUSE_LEFT`, `MOUSE_MIDDLE`, or `MOUSE_RIGHT`.

Reimplemented from MouseListener.

Definition at line 252 of file UserInterface.cpp.

2.78.2.6 void UserInterface::MouseWheelEvent (const Vector2 & *scrollOffset*) [virtual]

Called whenever the player moves the scroll wheel on the mouse.

Parameters

scrollOffset	The change in position of the scroll wheel. Note that if it's an actual wheel, the X component of the vector will always be 0; the two-dimensional vector also takes into account trackpad scrolling.

Reimplemented from MouseListener.

Definition at line 264 of file UserInterface.cpp.

2.78.2.7 void UserInterface::HandleKey (int *key*, bool *down*)

For internal engine use only; handles keyboard input.

2.78.2.8 void UserInterface::HandleCharacter (wchar_t *chr*)

For internal engine use only; handles keyboard input.

2.78.2.9 void UserInterface::Shutdown ( )

For internal engine use only; shuts down the UI layer cleanly when the world is destroyed.

Definition at line 174 of file UserInterface.cpp.

2.78.2.10 AngelUIHandle UserInterface::AddButton (const String & *label,* Vec2i *position,* void(∗)() *callback,* bool *center =* `false`*,* const String & *font =* `""`*,* Vec2i *padding =* Vec2i `(10, 10)`)

Adds a UI button to the world with formatting and a callback function.

Parameters

label	The text that will appear on the button
position	Where the button will appear on screen (in pixel units)
callback	The function to be called when the button is pressed. Must return void and not take any parameters.
center	Whether the button should be centered on the given position. If false, the position will be its top left.
font	The font to use for the button's text. If left as an empty string (the default), it will use the UI layer's default font, which is the same as the Angel console.
padding	How much space (in pixels) to leave between the text and the edge of the button.

Returns

A UI handle that can be passed to RemoveUIElement

Definition at line 292 of file UserInterface.cpp.

2.78.2.11 AngelUIHandle UserInterface::ShowChoiceBox (const String & *choiceLabel,* const StringList & *labels,* void(∗)(int) *callback,* const String & *font =* `""`*,* Vec2i *padding =* Vec2i `(10, 10)`*,* bool *modal =* `true`)

Pops a choice box consisting of several buttons and reports to the callback when one is clicked. When a button is clicked, the choice box will automatically remove itself, so the AngelUIHandle returned from this function only has to be tracked if you want to remove the box manually for some reason.

Parameters

choiceLabel	The label at the top of the box, describing the choice the player is making
labels	A list of Strings that will be used for the buttons in the choice
callback	The function to be called when any of the buttons are pressed. Must return void and take a single integer as a parameter. When the function is called, the integer will be the index of the button that was pressed (from zero, so it will match the index in the labels list).
font	The font to use for the buttons' text. If left as an empty string (the default), it will use the UI layer's default font, which is the same as the Angel console.
padding	How much space (in pixels) to leave between the text and the edge of the button. Also how much space will be left between buttons.

Returns

A UI handle that can be passed to RemoveUIElement.

Definition at line 320 of file UserInterface.cpp.

2.78.2.12 void UserInterface::RemoveUIElement (AngelUIHandle *element*)

Removes a UI element that was added through one of the UI functions described elsewhere in this file. Requires the opaque handle that was returned when the element was created.

Parameters

element	The handle for the UI element to remove.

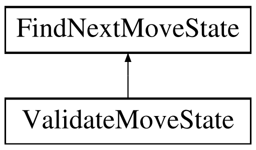
Definition at line 277 of file UserInterface.cpp.

The documentation for this class was generated from the following files:

- UI/UserInterface.h
- UI/UserInterface.cpp

2.79 ValidateMoveState Class Reference

Inheritance diagram for ValidateMoveState:



Public Member Functions

- virtual const char ∗ **GetName** ()
- virtual bool **Update** (PathFinderMove &)

Additional Inherited Members

2.79.1 Detailed Description

Definition at line 152 of file PathFinder.cpp.

The documentation for this class was generated from the following file:

- AI/PathFinder.cpp

2.80 Vec2i Struct Reference

A handy set of structures for passing around sets of numbers.

```
#include <VecStructs.h>
```

Public Member Functions

- **Vec2i** (int x, int y)
- bool **operator==** (const Vec2i &v) const
- bool **operator!=** (const Vec2i &v) const

Public Attributes

- int **X**
- int **Y**

2.80.1 Detailed Description

The little brother of Vector2 and Vector3, which handle floats, Vec2i simply bundles together two integers to make it easier to pass around things like pixel coordinates.

Definition at line 38 of file VecStructs.h.

The documentation for this struct was generated from the following file:

- Infrastructure/VecStructs.h

2.81 Vec2ui Struct Reference

```
#include <VecStructs.h>
```

Public Member Functions

- **Vec2ui** (unsigned int x, unsigned int y)
- bool **operator==** (const Vec2ui &v) const
- bool **operator!=** (const Vec2ui &v) const

Public Attributes

- unsigned int **X**
- unsigned int **Y**

2.81.1 Detailed Description

Similar to Vec2i, but for situations that demand unsigned integers.

Definition at line 51 of file VecStructs.h.

The documentation for this struct was generated from the following file:

- Infrastructure/VecStructs.h

2.82 Vec3i Struct Reference

```
#include <VecStructs.h>
```

Public Member Functions

- **Vec3i** (int x, int y, int z)
- bool **operator==** (const Vec3i &v) const
- bool **operator!=** (const Vec3i &v) const

Public Attributes

- int **X**
- int **Y**
- int **Z**

2.82.1 Detailed Description

The 3-integer version of Vec2i.

Definition at line 64 of file VecStructs.h.

The documentation for this struct was generated from the following file:

- Infrastructure/VecStructs.h

2.83 Vec3ui Struct Reference

```
#include <VecStructs.h>
```

Public Member Functions

- **Vec3ui** (unsigned int x, unsigned int y, unsigned int z)
- bool **operator==** (const Vec3ui &v) const
- bool **operator!=** (const Vec3ui &v) const

Public Attributes

- unsigned int **X**
- unsigned int **Y**
- unsigned int **Z**

2.83.1 Detailed Description

The 3-integer version of Vec2ui.

Definition at line 77 of file VecStructs.h.

The documentation for this struct was generated from the following file:

- Infrastructure/VecStructs.h

2.84 Vector2 Struct Reference

A two-dimensional floating point vector and associated math functions.

```
#include <Vector2.h>
```

Public Member Functions

- Vector2 (float x, float y)
- Vector2 (float value)
- Vector2 (const Vec2i &copy)
- Vector2 ()
- float Length ()
- float LengthSquared ()
- void Normalize ()
- bool **operator==** (const Vector2 &v) const
- bool **operator!=** (const Vector2 &v) const
- Vector2 **operator-** () const
- Vector2 **operator-** (const Vector2 &v) const
- Vector2 **operator+** (const Vector2 &v) const
- Vector2 **operator/** (float divider) const
- Vector2 **operator*** (float scaleFactor) const
- Vector2 & **operator+=** (const Vector2 &v)
- Vector2 & **operator-=** (const Vector2 &v)
- Vector2 & **operator*=** (float f)
- Vector2 & **operator/=** (float f)

Static Public Member Functions

- static float Distance (const Vector2 &value1, const Vector2 &value2)
- static float DistanceSquared (const Vector2 &value1, const Vector2 &value2)
- static float Dot (const Vector2 &value1, const Vector2 &value2)
- static float Cross (const Vector2 &value1, const Vector2 &value2)
- static Vector2 Normalize (const Vector2 &value)
- static Vector2 Reflect (const Vector2 &vector, const Vector2 &normal)
- static Vector2 Min (const Vector2 &value1, const Vector2 &value2)
- static Vector2 Max (const Vector2 &value1, const Vector2 &value2)
- static Vector2 Clamp (const Vector2 &value, const Vector2 &min, const Vector2 &max)
- static Vector2 Lerp (const Vector2 &value1, const Vector2 &value2, float amount)
- static Vector2 Negate (const Vector2 &value)
- static Vector2 Rotate (const Vector2 &value, const float radians)

Public Attributes

- float X
- float Y

Static Public Attributes

- static Vector2 Zero
- static Vector2 One
- static Vector2 UnitX
- static Vector2 UnitY

2.84.1 Detailed Description

A floating point two-dimensional vector. Can be used either as a traditional vector (indicating a direction) or a position (treating X and Y as coordinates).

Definition at line 41 of file Vector2.h.

2.84.2 Constructor & Destructor Documentation

2.84.2.1 Vector2::Vector2 (float *x*, float *y*)

Constructor to initialize the vector to set dimensions

Parameters

x	The X dimension
y	The Y dimension

Definition at line 42 of file Vector2.cpp.

2.84.2.2 Vector2::Vector2 (float *value*)

Constructor to initialize the vector to uniform dimensions

Parameters

value	The value to use for both the X and Y dimension

Definition at line 47 of file Vector2.cpp.

2.84.2.3 Vector2::Vector2 (const Vec2i & *copy*)

Constructor to initalize from a Vec2i struct.

Parameters

copy	The Vec2i to be converted into a Vector2.

Definition at line 57 of file Vector2.cpp.

2.84.2.4 Vector2::Vector2 ( )

Constructor to initialize a zero-length vector (0, 0)

Definition at line 52 of file Vector2.cpp.

2.84.3 Member Function Documentation

2.84.3.1 float Vector2::Length ( )

Get the absolute magnitude of the vector. Uses a square-root, so be careful calling this too much within a loop.

Returns

The length (magnitude) of the vector

Definition at line 62 of file Vector2.cpp.

2.84.3.2 float Vector2::LengthSquared ( )

Get the squared magnitude of the vector – if all you care about is comparison, it's a lot faster to consider the squared lengths of the vectors.

Returns

The length (magnitude) of the vector squared

Definition at line 67 of file Vector2.cpp.

2.84.3.3 float Vector2::Distance (const Vector2 & *value1,* const Vector2 & *value2*) `[static]`

Get the absolute distance between two points (most useful if the Vector2 represents a position). Uses a square-root, so be careful calling this too much within a loop.

Parameters

value1	The first point
value2	The second point

Returns

The distance between the two points

Definition at line 72 of file Vector2.cpp.

2.84.3.4 float Vector2::DistanceSquared (const Vector2 & *value1,* const Vector2 & *value2*) `[static]`

Get the absolute distance between two points – if all you care about is comparison, it's a lot faster to consider the squared lengths of the vectors.

Parameters

value1	The first point
value2	The second point

Returns

The distance between the two points squared

Definition at line 77 of file Vector2.cpp.

2.84.3.5 float Vector2::Dot (const Vector2 & *value1*, const Vector2 & *value2*) [static]

Get the dot product of two vectors.

Parameters

value1	The first vector
value2	The second vector

Returns

The dot product

Definition at line 82 of file Vector2.cpp.

2.84.3.6 float Vector2::Cross (const Vector2 & *value1*, const Vector2 & *value2*) [static]

Get the cross product of two vectors. Note that the **mathematical** definition of a cross product results in another vector perpendicular to the two inputs, but since both of our vectors are 2D, the returned vector will always have X and Y components of 0. Thus this function returns what would be the Z component of that vector.

Parameters

value1	The first vector
value2	The second vector

Returns

The Z component of the cross product

Definition at line 87 of file Vector2.cpp.

2.84.3.7 void Vector2::Normalize ( )

Normalizes a vector in place – retains its direction, but ensures that its magnitude is equal to 1.0.

Definition at line 92 of file Vector2.cpp.

2.84.3.8 Vector2 Vector2::Normalize (const Vector2 & *value*) [static]

Get the normalized value for a Vector2 without affecting the original.

Parameters

value	The Vector2 to normalize

Returns

A normalized version of the passed-in Vector2

Definition at line 110 of file Vector2.cpp.

2.84.3.9 Vector2 Vector2::Reflect (const Vector2 & *vector*, const Vector2 & *normal*) [static]

Reflect one Vector around another

Parameters

vector	The vector to reflect
normal	The normal to reflect it around

Returns

The new vector resulting from the reflection

Definition at line 117 of file Vector2.cpp.

2.84.3.10 Vector2 Vector2::Min (const Vector2 & *value1*, const Vector2 & *value2*) [static]

Get a new vector from the minimum X and minimum Y of the two

Returns

The vector composed from minimums on both axes

Definition at line 122 of file Vector2.cpp.

2.84.3.11 Vector2 Vector2::Max (const Vector2 & *value1*, const Vector2 & *value2*) [static]

Get a new vector from the maximum X and maximum Y of the two

Returns

The vector composed from the maximums on both axes

Definition at line 127 of file Vector2.cpp.

2.84.3.12 Vector2 Vector2::Clamp (const Vector2 & *value*, const Vector2 & *min*, const Vector2 & *max*) [static]

Clamp a vector to a given minimum and maximum

Parameters

value	The vector to be clamped
min	The vector representing the X and Y minimums for the clamping
max	The vector representing the X and Y maximums for the clamping

Returns

The clamped vector

Definition at line 132 of file Vector2.cpp.

2.84.3.13 Vector2 Vector2::Lerp (const Vector2 & *value1*, const Vector2 & *value2*, float *amount*) [static]

Perform a linear interpolation between two vectors

Parameters

value1	The starting point vector
value2	The ending point vector
amount	The amount (from 0.0 to 1.0) to interpolate between them

Returns

The interpolated vector

Definition at line 137 of file Vector2.cpp.

2.84.3.14 Vector2 Vector2::Negate (const Vector2 & *value*) [static]

Get a negated vector – if you add the result and the original, you should get a zero-length vector (0, 0)

Parameters

value	The vector to negate

Returns

The negated vector

Definition at line 142 of file Vector2.cpp.

2.84.3.15 Vector2 Vector2::Rotate (const Vector2 & *value,* const float *radians*) [static]

Rotate a vector

Parameters

value	The original vector to rotate
radians	The rotation angle (in radians)

Returns

The rotated vector

Definition at line 147 of file Vector2.cpp.

2.84.4 Member Data Documentation

2.84.4.1 float Vector2::X

The X dimension, publicly available because it's so often gotten or set and there's no good reason to encapsulate it.

Definition at line 47 of file Vector2.h.

2.84.4.2 float Vector2::Y

The Y dimension, also publicly available.

Definition at line 52 of file Vector2.h.

2.84.4.3 Vector2 Vector2::Zero [static]

A reference to a zero-length vector (0, 0)

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 61 of file Vector2.h.

2.84.4.4 Vector2 Vector2::One [static]

A reference to a (1, 1) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 70 of file Vector2.h.

2.84.4.5 Vector2 Vector2::UnitX [static]

A reference to a (1, 0) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 79 of file Vector2.h.

2.84.4.6 Vector2 Vector2::UnitY [static]

A reference to a (0, 1) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 88 of file Vector2.h.

The documentation for this struct was generated from the following files:

- Infrastructure/Vector2.h
- Infrastructure/Vector2.cpp

2.85 Vector3 Struct Reference

A three-dimensional floating point vector and associated math functions.

```
#include <Vector3.h>
```

Public Member Functions

- Vector3 (float x, float y, float z)
- Vector3 (float value)
- Vector3 ()
- float Length ()
- float LengthSquared ()
- void Normalize ()
- bool **operator==** (const Vector3 &v) const
- bool **operator!=** (const Vector3 &v) const
- Vector3 **operator-** () const
- Vector3 **operator-** (const Vector3 &v) const
- Vector3 **operator+** (const Vector3 &v) const
- Vector3 **operator/** (float divider) const
- Vector3 **operator*** (float scaleFactor) const
- Vector3 & **operator+=** (const Vector3 &v)
- Vector3 & **operator-=** (const Vector3 &v)
- Vector3 & **operator*=** (float f)
- Vector3 & **operator/=** (float f)

Static Public Member Functions

- static float Distance (const Vector3 &value1, const Vector3 &value2)
- static float DistanceSquared (const Vector3 &value1, const Vector3 &value2)
- static float Dot (const Vector3 &value1, const Vector3 &value2)
- static Vector3 Normalize (const Vector3 &value)
- static Vector3 Reflect (const Vector3 &vector, const Vector3 &normal)
- static Vector3 Min (const Vector3 &value1, const Vector3 &value2)
- static Vector3 Max (const Vector3 &value1, const Vector3 &value2)
- static Vector3 Clamp (const Vector3 &value1, const Vector3 &min, const Vector3 &max)
- static Vector3 Lerp (const Vector3 &value1, const Vector3 &value2, float amount)
- static Vector3 Negate (const Vector3 &value)

Public Attributes

- float X
- float Y
- float Z

Static Public Attributes

- static Vector3 Zero
- static Vector3 One
- static Vector3 UnitX
- static Vector3 UnitY
- static Vector3 UnitZ

2.85.1 Detailed Description

A floating point three-dimensional vector. Can be used either as a traditional vector (indicating a direction) or a position (treating X, Y, and Z as coordinates).

Definition at line 38 of file Vector3.h.

2.85.2 Constructor & Destructor Documentation

2.85.2.1 Vector3::Vector3 (float *x*, float *y*, float *z*)

Constructor to initialize the vector to set dimensions

Parameters

x	The X dimension
y	The Y dimension
z	The Z dimension

Definition at line 43 of file Vector3.cpp.

2.85.2.2 Vector3::Vector3 (float *value*)

Constructor to initialize the vector to uniform dimensions

Parameters

value	The value to use for the X, Y, and Z dimensions

Definition at line 49 of file Vector3.cpp.

2.85.2.3 Vector3::Vector3 ( )

Constructor to initialize a zero-length vector (0, 0, 0)

Definition at line 55 of file Vector3.cpp.

2.85.3 Member Function Documentation

2.85.3.1 float Vector3::Length ( )

Get the absolute magnitude of the vector. Uses a square-root, so be careful calling this too much within a loop.

Returns

The length (magnitude) of the vector

Definition at line 61 of file Vector3.cpp.

2.85.3.2 float Vector3::LengthSquared ( )

Get the squared magnitude of the vector – if all you care about is comparison, it's a lot faster to consider the squared lengths of the vectors.

Returns

The length (magnitude) of the vector squared

Definition at line 66 of file Vector3.cpp.

2.85.3.3 float Vector3::Distance (const Vector3 & *value1*, const Vector3 & *value2*) [static]

Get the absolute distance between two points (most useful if the Vector2 represents a position). Uses a square-root, so be careful calling this too much within a loop.

Parameters

value1	The first point
value2	The second point

Returns

The distance between the two points

Definition at line 71 of file Vector3.cpp.

2.85.3.4 float Vector3::DistanceSquared (const Vector3 & *value1*, const Vector3 & *value2*) [static]

Get the absolute distance between two points – if all you care about is comparison, it's a lot faster to consider the squared lengths of the vectors.

Parameters

value1	The first point
value2	The second point

Returns

The distance between the two points squared

Definition at line 76 of file Vector3.cpp.

2.85.3.5 float Vector3::Dot (const Vector3 & *value1,* const Vector3 & *value2*) [static]

Get the dot product of two vectors.

Parameters

value1	The first vector
value2	The second vector

Returns

The dot product

Definition at line 82 of file Vector3.cpp.

2.85.3.6 void Vector3::Normalize ( )

Normalizes a vector in place – retains its direction, but ensures that its magnitude is equal to 1.0.

Definition at line 87 of file Vector3.cpp.

2.85.3.7 Vector3 Vector3::Normalize (const Vector3 & *value*) [static]

Get the normalized value for a Vector2 without affecting the original.

Parameters

value	The Vector3 to normalize

Returns

A normalized version of the passed-in Vector3

Definition at line 107 of file Vector3.cpp.

2.85.3.8 Vector3 Vector3::Reflect (const Vector3 & *vector,* const Vector3 & *normal*) [static]

Reflect one Vector around another

Parameters

vector	The vector to reflect
normal	The normal to reflect it around

Returns

The new vector resulting from the reflection

Definition at line 114 of file Vector3.cpp.

2.85.3.9 Vector3 Vector3::Min (const Vector3 & *value1,* const Vector3 & *value2*) [static]

Get a new vector from the minimum X, Y, and Z of the two

Returns

The vector composed from minimums on all axes

Definition at line 119 of file Vector3.cpp.

2.85.3.10 Vector3 Vector3::Max (const Vector3 & *value1*, const Vector3 & *value2*) [static]

Get a new vector from the maximum X, Y, and Z of the two

Returns

The vector composed from the maximums on all axes

Definition at line 124 of file Vector3.cpp.

2.85.3.11 Vector3 Vector3::Clamp (const Vector3 & *value1*, const Vector3 & *min*, const Vector3 & *max*) [static]

Clamp a vector to a given minimum and maximum

Parameters

value	The vector to be clamped
min	The vector representing the X, Y, and Z minimums for the clamping
max	The vector representing the X, Y, and Z maximums for the clamping

Returns

The clamped vector

Definition at line 129 of file Vector3.cpp.

2.85.3.12 Vector3 Vector3::Lerp (const Vector3 & *value1*, const Vector3 & *value2*, float *amount*) [static]

Perform a linear interpolation between two vectors

Parameters

value1	The starting point vector
value2	The ending point vector
amount	The amount (from 0.0 to 1.0) to interpolate between them

Returns

The interpolated vector

Definition at line 134 of file Vector3.cpp.

2.85.3.13 Vector3 Vector3::Negate (const Vector3 & *value*) [static]

Get a negated vector – if you add the result and the original, you should get a zero-length vector (0, 0)

Parameters

value	The vector to negate

Returns

The negated vector

Definition at line 138 of file Vector3.cpp.

2.85.4 Member Data Documentation

2.85.4.1 float Vector3::X

The X dimension, publicly available because it's so often gotten or set and there's no good reason to encapsulate it.

Definition at line 44 of file Vector3.h.

2.85.4.2 float Vector3::Y

The Z dimension, publicly available because it's so often gotten or set and there's no good reason to encapsulate it.

Definition at line 50 of file Vector3.h.

2.85.4.3 float Vector3::Z

The Z dimension, publicly available because it's so often gotten or set and there's no good reason to encapsulate it.

Definition at line 56 of file Vector3.h.

2.85.4.4 Vector3 Vector3::Zero `[static]`

A reference to a zero-length vector (0, 0, 0)

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 65 of file Vector3.h.

2.85.4.5 Vector3 Vector3::One `[static]`

A reference to a (1, 1, 1) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 74 of file Vector3.h.

2.85.4.6 Vector3 Vector3::UnitX `[static]`

A reference to a (1, 0, 0) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 83 of file Vector3.h.

2.85.4.7 Vector3 Vector3::UnitY `[static]`

A reference to a (0, 1, 0) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 92 of file Vector3.h.

2.85.4.8 Vector3 Vector3::UnitZ `[static]`

A reference to a (0, 0, 1) vector

NB: We can't make a static member variable constant, so it is possible to change this value. That is a horrible idea and will make the engine act "odd" at best.

Definition at line 101 of file Vector3.h.

The documentation for this struct was generated from the following files:

- Infrastructure/Vector3.h

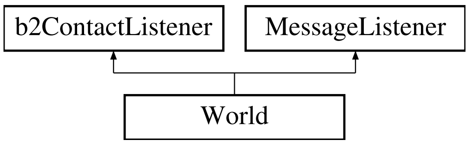
- Infrastructure/Vector3.cpp

2.86 World Class Reference

The central class that manages all aspects of the simulation.

```
#include <World.h>
```

Inheritance diagram for World:



Public Member Functions

- bool Initialize (unsigned int windowWidth=1024, unsigned int windowHeight=768, String windowName="Angel Engine", bool antiAliasing=false, bool fullScreen=false, bool resizable=false)
- std::vector< Vec3ui > GetVideoModes ()
- void AdjustWindow (int windowWidth, int windowHeight, const String &windowName)
- void MoveWindow (int xPosition, int yPosition)
- GLFWwindow * GetMainWindow ()
- bool SetupPhysics (const Vector2 &gravity=Vector2(0,-10), const Vector2 &maxVertex=Vector2(100.0f, 100.-0f), const Vector2 &minVertex=Vector2(-100.0f,-100.0f))
- void Destroy ()
- void ResetWorld ()
- void StartGame ()
- void StopGame ()
- void ScriptExec (const String &code)
- void LoadLevel (const String &levelName)
- const float GetDT ()
- const bool PauseSimulation ()
- const bool ResumeSimulation ()
- const bool PausePhysics ()
- const bool ResumePhysics ()
- void SetBackgroundColor (const Color &bgColor)
- void Add (Renderable *newElement, int layer=0)
- void Add (Renderable *newElement, const String &layer)
- void Remove (Renderable *oldElement)
- void UpdateLayer (Renderable *element, int newLayer)
- void UpdateLayer (Renderable *element, const String &newLayerName)
- void NameLayer (const String &name, int number)
- const int GetLayerByName (const String &name)
- RenderLayers & GetLayers ()
- void RegisterConsole (Console *console)
- Console * GetConsole ()
- b2World & GetPhysicsWorld ()
- const bool IsPhysicsSetUp ()
- void WakeAllPhysics ()
- virtual void BeginContact (b2Contact *contact)
- virtual void EndContact (b2Contact *contact)
- void SetSideBlockers (bool turnOn, float restitution=-1.0f)
- float GetCurrentTimeSeconds ()

- float GetTimeSinceSeconds (float lastTime)
- void DrawDebugLine (const Vector2 &a, const Vector2 &b, float time=5.f, const Color &color=Color(1.f, 0.f, 0.f))
- void PurgeDebugDrawing ()
- const bool IsSimulationOn ()
- void SetGameManager (GameManager ∗gameManager)
- GameManager ∗ GetGameManager ()
- RenderableIterator GetFirstRenderable ()
- RenderableIterator GetLastRenderable ()
- void UnloadAll ()
- virtual void ReceiveMessage (Message ∗m)
- const bool IsHighResScreen ()
- void SetHighResolutionScreen (bool highRes)
- const bool IsAntiAliased ()
- void TickAndRender ()
- void Tick ()
- void Render ()
- void SetDT (float dt)

Static Public Member Functions

- static World & GetInstance ()

Protected Member Functions

- float **CalculateNewDT** ()
- void **UpdateRenderables** (float frame_dt)
- void **CleanupRenderables** ()
- void **DrawRenderables** ()
- void **Simulate** (bool simRunning)
- void **ProcessDeferredAdds** ()
- void **ProcessDeferredLayerChanges** ()
- void **ProcessDeferredRemoves** ()
- void **RunPhysics** (float frame_dt)
- void **UpdateDebugItems** (float frame_dt)
- void **DrawDebugItems** ()

Static Protected Attributes

- static World ∗ **s_World** = NULL

2.86.1 Detailed Description

The World is the class that keeps track of all Renderables, handles the Update/Render loop, processes events, etc.

Like the Camera, it uses the singleton pattern; you can't actually declare a new instance of a TagCollection. To access the World, use "theWorld" to retrieve the singleton object. "theWorld" is defined in both C++ and Lua.

If you're not familiar with the singleton pattern, this paper is a good starting point. (Don't be afraid that it's written by Microsoft.)

`http://msdn.microsoft.com/en-us/library/ms954629.aspx`

Definition at line 66 of file World.h.

2.86.2 Member Function Documentation

2.86.2.1 World & World::GetInstance ( ) [static]

Used to access the singleton instance of this class. As a shortcut, you can just use "theWorld".

Returns

The singleton

Definition at line 81 of file World.cpp.

2.86.2.2 bool World::Initialize (unsigned int *windowWidth* = 1024, unsigned int *windowHeight* = 768, String *windowName* = "Angel Engine", bool *antiAliasing* = false, bool *fullScreen* = false, bool *resizable* = false)

The initial call to get Angel up and running. Opens the window, sets up our display, and begins the Update/Render loop.

The values passed in here will be overridden by any values set in the Preferences, with the WindowSettings table. (See the Preferences documentation and the example preferences file in IntroGame for more information.)

Parameters

windowWidth	The desired width (in pixels) of the window
windowHeight	The desired height (in pixels) of the window
windowName	The string that should appear in the window's title bar
antiAliasing	Whether or not the window should be initialized with anti-aliasing
fullScreen	Whether the game should be started in fullscreen mode. Note that the resolution is determined from windowWidth and windowHeight, and that Angel will attempt to change the system's video mode accordingly. We don't do any detection of what modes are valid, so make sure you're feeding this a legitimate resolution.
resizable	Whether the user will be allowed to resize the game window 8 using the operating system's normal resize widgets.

Returns

Returns true if initialization worked (i.e. if world was not already initialized)

Definition at line 109 of file World.cpp.

2.86.2.3 std::vector< Vec3ui > World::GetVideoModes ( )

Queries the video drivers to get a list of supported video modes for fullscreen gameplay. You'll likely want to get a list of valid modes to give the player a choice of resolutions. (You can then save the selected mode as part of the preferences.)

Returns

A list of video modes bundled, each bundled as a Vec3ui. The X value is the width; the Y value is the height, and the Z-value represents the color depth.

Definition at line 328 of file World.cpp.

2.86.2.4 void World::AdjustWindow (int *windowWidth*, int *windowHeight*, const String & *windowName*)

Changes the dimension of the window while the game is running. Note that the behavior is undefined if this method is called while in fullscreen mode.

Parameters

windowWidth	The new desired width
windowHeight	The new desired height
windowName	The new string to go in the window's title bar

Definition at line 350 of file World.cpp.

2.86.2.5 void World::MoveWindow (int *xPosition*, int *yPosition*)

Moves the window around on screen, relative to the system origin. Note that behavior is undefined if this method is called while in fullscreen mode.

Parameters

xPosition	The new x position offset from the system origin
yPosition	The new y position offset from the system origin

Definition at line 364 of file World.cpp.

2.86.2.6 GLFWwindow ∗ World::GetMainWindow ( )

Returns a handle for the main window. For most games, you'll only have one window, so this seems a little redundant, but there are several GLFW functions that need to operate on a specific window, so this gives you a hook to that.

Returns

a handle for the main game window

Definition at line 322 of file World.cpp.

2.86.2.7 bool World::SetupPhysics (const Vector2 & *gravity* = Vector2 `(0, -10)`, const Vector2 & *maxVertex* = Vector2 `(100.0f, 100.0f)`, const Vector2 & *minVertex* = Vector2 `(-100.0f, -100.0f)`)

Intialize physics. If you're not using our built-in physics, you don't have to call this function, but no PhysicsActors will do anything until you do.

Parameters

gravity	The gravity vector that will be applied to all PhysicsActors
maxVertex	The maximum vertex at which PhysicsActors will simulate. Any PhysicsActors that go beyond this point will get "stuck" in the bounding box.
minVertex	The minimum vertex at which PhysicsActors will simulate. Any PhysicsActors that go beyond this point will get "stuck" in the bounding box.

Returns

True is successfully setup, false if physics were already initialized

Definition at line 372 of file World.cpp.

2.86.2.8 void World::Destroy ( )

Called when the game shuts down, does all cleanup.

Definition at line 392 of file World.cpp.

2.86.2.9 void World::ResetWorld ( )

Removes all Actors from the world (pending a check to the GameManager::IsProtectedFromUnloadAll function) and resets the camera to its default positioning.

Definition at line 738 of file World.cpp.

2.86.2.10 void World::StartGame ( )

Called once, after your world setup is done and you're ready to kick things into motion.

Definition at line 410 of file World.cpp.

2.86.2.11 void World::StopGame ( )

Ends the game, has everything prepare for shutdown. Should only be called once, when you're ready to finish.

Definition at line 443 of file World.cpp.

2.86.2.12 void World::ScriptExec (const String & *code*)

Execute a string of Lua code.

Parameters

code	The string to execute

Definition at line 448 of file World.cpp.

2.86.2.13 void World::LoadLevel (const String & *levelName*)

Loads a level from Config/Level/[levelName].lua.

Each section of a level file specifies an Actor to be added to the world. The name of the section will be the name of the Actor, and the values in each section will be applied in addition to (or overriding) the ones specified in the archetype. Check the Actor::Create function for more information on how archetypes are specified.

See Also

Actor::Create

Parameters

levelName	

Definition at line 453 of file World.cpp.

2.86.2.14 const float World::GetDT ( )

Get the amount of time that elapsed between the start of the last frame and the start of the current frame. Useful for rate controlling any changes on your actors.

Returns

The amount of time elapsed in seconds

Definition at line 695 of file World.cpp.

2.86.2.15 const bool World::PauseSimulation ( )

Toggles the simulation off. Several classes and objects will still receive updates (GameManager, Console, Camera, etc.), but all standard Actors will not receive Update calls until you call World::StartSimulation.

Returns

Whether the simulation was successfully paused

Definition at line 705 of file World.cpp.

2.86.2.16 const bool World::ResumeSimulation ( )

Toggles the simulation back on.

See Also

World::PauseSimulation

Returns

Whether the simulation was successfully resumed

Definition at line 700 of file World.cpp.

2.86.2.17 const bool World::PausePhysics ( )

Toggles the physics simulation off.

Returns

Whether the physics simulation was successfully paused

Definition at line 716 of file World.cpp.

2.86.2.18 const bool World::ResumePhysics ( )

Toggles the physics simulation back on.

See Also

World::PausePhysics

Returns

Whether the physics simulation was successfully resumed

Definition at line 727 of file World.cpp.

2.86.2.19 void World::SetBackgroundColor (const Color & *bgColor*)

Sets the world's background color. White by default.

Parameters

bgColor	The new background color

Definition at line 744 of file World.cpp.

2.86.2.20 void World::Add (Renderable ∗ *newElement*, int *layer* = 0)

Add a Renderable to the World so it will start receiving Update and Render calls every frame.

Parameters

∗*newElement*	The new object to insert into the world
layer	The layer at which to insert it

Definition at line 749 of file World.cpp.

2.86.2.21 void World::Add (Renderable ∗ *newElement*, const String & *layer*)

Add a Renderable to the World with a named layer.

Parameters

∗*newElement*	The new object to insert into the world
layer	The name of the layer at which to insert it

Definition at line 782 of file World.cpp.

2.86.2.22 void World::Remove (Renderable ∗ *oldElement*)

Remove a Renderable from the World. Does not deallocate any memory; you're responsible for doing that yourself.

Parameters

∗*oldElement*	The element to remove

Definition at line 787 of file World.cpp.

2.86.2.23 void World::UpdateLayer (Renderable ∗ *element*, int *newLayer*)

Move a Renderable to a new layer

Parameters

element	The renderable to move
newLayer	Its new home

Definition at line 841 of file World.cpp.

2.86.2.24 void World::UpdateLayer (Renderable ∗ *element*, const String & *newLayerName*)

Move a Renderable to a new layer by name

Parameters

element	The renderable to move
newLayerName	The name of its new home

Definition at line 854 of file World.cpp.

2.86.2.25 void World::NameLayer (const String & *name*, int *number*)

Lets you name layers for easier reference later. This name can be used by the World::Add and World::UpdateLayer functions.

Parameters

name	The string to assign as a layer name
number	The number to which this name should refer

Definition at line 859 of file World.cpp.

2.86.2.26 const int World::GetLayerByName (const String & *name*)

Retrieve the layer number associated with a name.

Parameters

name	The name to look up

Returns

The layer number. Will return 0 if the name has not been registered; note that 0 is still a valid layer.

Definition at line 864 of file World.cpp.

2.86.2.27 RenderLayers& World::GetLayers ( ) `[inline]`

Get the set of layers and their associated Renderables

Returns

The world's layers

Definition at line 319 of file World.h.

2.86.2.28 void World::RegisterConsole (Console ∗ *console*)

Register a new Console with the World. Only one Console can be activated at at time.

Parameters

console	The new Console that should accept input

Definition at line 1124 of file World.cpp.

2.86.2.29 Console ∗ World::GetConsole ( )

Get a pointer to the current registered console

Returns

The current console

Definition at line 1129 of file World.cpp.

2.86.2.30 b2World & World::GetPhysicsWorld ( )

Get a reference to the Box2D world that's handling all the Physics

Returns

Definition at line 898 of file World.cpp.

2.86.2.31 const bool World::IsPhysicsSetUp ( ) `[inline]`

Lets you know if physics have been appropriately initialized with the World::SetupPhysics function.

Returns

True if physics has been initialized

Definition at line 349 of file World.h.

2.86.2.32 void World::WakeAllPhysics ( )

Wakes up all physics bodies in the world. See the Box2D documentation for more information on what this means.

Definition at line 890 of file World.cpp.

2.86.2.33 void World::BeginContact (b2Contact ∗ *contact*) `[virtual]`

Implementation of the b2ContactListener::BeginContact function. We use it to manage collision notifications.

Definition at line 610 of file World.cpp.

2.86.2.34 void World::EndContact (b2Contact ∗ *contact*) `[virtual]`

Implementation of the b2ContactListener::EndContact function. We use it to manage collision notifications.

Definition at line 615 of file World.cpp.

2.86.2.35 void World::SetSideBlockers (bool *turnOn,* float *restitution* = `-1.0f`)

When working with physics, oftentimes you want to keep objects from going beyond the edge of the visible screen. This function sets up objects to block the edges so they can't.

Parameters

turnOn	If true, side blockers are enabled, if false, they're disabled
restitution	The restitution of the blockers (how bouncy they are)

Definition at line 903 of file World.cpp.

2.86.2.36 float World::GetCurrentTimeSeconds ( ) `[inline]`

Find out how much time has elapsed since the game started

Returns

Total elapsed time in seconds

Definition at line 385 of file World.h.

2.86.2.37 float World::GetTimeSinceSeconds (float *lastTime*) `[inline]`

Find out how much time has elapsed since a specific timestamp.

Parameters

lastTime	The time index you want to calculate the difference from

Returns

How much time has elapsed since lastTime

Definition at line 393 of file World.h.

2.86.2.38 void World::DrawDebugLine (const Vector2 & *a,* const Vector2 & *b,* float *time* = `5.f`, const Color & *color* = Color`(1.f, 0.f, 0.f)`)

Draw a line for a specified length of time.

Parameters

a	The starting point of the line
b	The ending point of the line
time	The length of time the line will be drawn (less than 0 will draw it permanently)
color	The color of the line

Definition at line 877 of file World.cpp.

2.86.2.39 void World::PurgeDebugDrawing ( )

Purge all debug drawing.

Definition at line 1066 of file World.cpp.

2.86.2.40 const bool World::IsSimulationOn ( )

Check whether the simulation is running. See also StartSimulation() and StopSimulation().

Returns

Whether the simulation is running

Definition at line 711 of file World.cpp.

2.86.2.41 void World::SetGameManager (GameManager ∗ *gameManager*)

Set a GameManager object to be your high-level coordinator.

Parameters

gameManager	The new manager

Definition at line 1089 of file World.cpp.

2.86.2.42 GameManager∗ World::GetGameManager ( ) [inline]

Get the current registered manager

Returns

The GameManager currently overseeing the game

Definition at line 430 of file World.h.

2.86.2.43 RenderableIterator World::GetFirstRenderable ( ) [inline]

Get an iterator to start cycling through all the Renderables that have been added to the World.

Returns

An iterator pointing to the first Renderable

Definition at line 438 of file World.h.

2.86.2.44 RenderableIterator World::GetLastRenderable ( ) [inline]

Get an iterator pointing to the last Renderable in the World.

Returns

An iterator at the end of the list

Definition at line 449 of file World.h.

2.86.2.45 void World::UnloadAll ( )

Removes all Actors from the world (pending a check to the GameManager::IsProtectedFromUnloadAll function).

Definition at line 1100 of file World.cpp.

2.86.2.46 void World::ReceiveMessage (**Message** ∗ *m*) `[virtual]`

Implementation of the MessageListener::ReceiveMessage function. Used to get notifications of camera changes so the side blockers can be updated if they're enabled.

Parameters

m	The message being delivered.

Implements MessageListener.

Definition at line 995 of file World.cpp.

2.86.2.47 const bool World::IsHighResScreen ( ) `[inline]`

Lets you know whether the current rendering device is high resolution (like the iPhone 4's "Retina Display").

Returns

Whether or not the screen is high resolution.

Definition at line 476 of file World.h.

2.86.2.48 void World::SetHighResolutionScreen (bool *highRes*) `[inline]`

INTERNAL: This function is called by the OS setup functions to let Angel know about the screen resolution. If you call in manually, expect weirdness.

Parameters

highRes	Whether or not we should be set up for high resolution rendering.

Definition at line 485 of file World.h.

2.86.2.49 const bool World::IsAntiAliased ( ) `[inline]`

Lets you know whether the current display is antialiased.

Returns

Whether or not we're running with antialiasing (multisampling).

Definition at line 492 of file World.h.

2.86.2.50 void World::TickAndRender ( )

INTERNAL: This function is used by various OS systems to run the Angel update loop. If you call it manually, expect weirdness.

Definition at line 620 of file World.cpp.

2.86.2.51 void World::Tick ( )

INTERNAL: This function is used by various OS systems to run the Angel update loop. If you call it manually, expect weirdness.

Definition at line 626 of file World.cpp.

2.86.2.52 void World::Render ( )

INTERNAL: This function is used by various OS systems to run the Angel update loop. If you call it manually, expect weirdness.

Definition at line 631 of file World.cpp.

2.86.2.53 void World::SetDT (float *dt*) `[inline]`

INTERNAL: This function is used by various OS systems to run the Angel update loop. If you call it manually, expect weirdness.

Definition at line 516 of file World.h.

The documentation for this class was generated from the following files:

- Infrastructure/World.h
- Infrastructure/World.cpp

2.87 XboxButtonBindRecord Struct Reference

Public Attributes

- int **HashKey**
- const bool(Controller::∗ **CheckFunc**)()

2.87.1 Detailed Description

Definition at line 198 of file InputManager.cpp.

The documentation for this struct was generated from the following file:

- Input/InputManager.cpp

3 File Documentation

3.1 Actors/Actor.h File Reference

```
#include "../Infrastructure/Renderable.h"
#include "../Infrastructure/Color.h"
#include "../Infrastructure/Interval.h"
#include "../Messaging/Message.h"
```

Classes

- class Actor

  Basic simulation element for Angel.

Macros

- #define **MAX_SPRITE_FRAMES** 64
- #define **CIRCLE_DRAW_SECTIONS** 32

Typedefs

- typedef std::vector< Actor ∗ > **ActorList**
- typedef std::set< Actor ∗ > **ActorSet**

Enumerations

- enum spriteAnimationType { **SAT_None**, **SAT_Loop**, **SAT_PingPong**, **SAT_OneShot** }
- enum actorDrawShape { **ADS_Square**, **ADS_Circle**, **ADS_CustomList** }

3.1.1 Enumeration Type Documentation

3.1.1.1 enum spriteAnimationType

An enumeration for the type of animations that can be given to an Actor.

Definition at line 44 of file Actor.h.

3.1.1.2 enum actorDrawShape

An enumeration for the shape of an Actor when it gets drawn to the screen. Note that at present, circular actors can't use textures.

Definition at line 56 of file Actor.h.

3.2 Actors/ParticleActor.h File Reference

```
#include "../Actors/Actor.h"
```

Classes

- class ParticleActor

  An Actor that draws and keeps track of drawing a particle system on screen.
- struct ParticleActor::Particle

3.3 Actors/TextActor.h File Reference

```
#include "../Actors/Actor.h"
```

Classes

- class TextActor

  An Actor for displaying text on the screen.

Enumerations

- enum TextAlignment { **TXT_Left**, **TXT_Center**, **TXT_Right** }

3.3.1 Enumeration Type Documentation

3.3.1.1 enum TextAlignment

An enumeration for the alignment of text within a TextActor

Definition at line 38 of file TextActor.h.

3.4 Infrastructure/Common.h File Reference

```
#include <GLFW/glfw3.h>
#include <math.h>
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
```

3.4.1 Detailed Description

This file handles the common definitions and includes that each platform uses, as well as the C++ libraries used frequently in Angel.

Definition in file Common.h.

3.5 Infrastructure/TextRendering.h File Reference

```
#include "../Util/StringUtil.h"
```

Functions

- const bool RegisterFont (const String &filename, int pointSize, const String &nickname)
- const bool IsFontRegistered (const String &nickname)
- const bool UnRegisterFont (const String &nickname)
- Vector2 DrawGameText (const String &text, const String &nickname, int pixelX, int pixelY, float angle=0.0f)
- Vector2 DrawGameTextRaw (const String &text, const String &nickname, int pixelX, int pixelY, float angle=0.-0f)
- Vector2 GetTextExtents (const String &text, const String &nickname)
- float GetTextAscenderHeight (const String &nickname)

3.5.1 Detailed Description

A low-level C-style interface for drawing text to the screen. Most of the functionality is wrapped in TextActor, which is where you should probably be focusing if you care about text. But if you want/need to wrangle your own text functionality, here you go.

Definition in file TextRendering.h.

3.5.2 Function Documentation

3.5.2.1 const bool RegisterFont (const String & *filename*, int *pointSize*, const String & *nickname*)

Register a font with our text rendering system.

Parameters

filename	The path to the font file you want to use. Must be a format readable by FreeType (`http-://www.freetype.org`). Most major font formats are acceptable.
pointSize	The size, in points, that you want the text to render.
nickname	How you want to refer to the font when telling it to draw

Returns

Whether or not it successfully registered (check the error log if this returns false)

Definition at line 48 of file TextRendering.cpp.

3.5.2.2 const bool IsFontRegistered (const String & *nickname*)

Tell whether there is already a font with a given name (to avoid repeat loading).

Returns

Whether or not there is currently a registered font by the given name.

Definition at line 79 of file TextRendering.cpp.

3.5.2.3 const bool UnRegisterFont (const String & *nickname*)

If you're done using a font and are concerned about memory usage, you can remove it from active use.

Parameters

nickname	The name you gave the font when you registered it

Returns

Whether it successfully unregistered. If false, check the error log.

Definition at line 92 of file TextRendering.cpp.

3.5.2.4 Vector2 DrawGameText (const String & *text,* const String & *nickname,* int *pixelX,* int *pixelY,* float *angle* = `0.0f`)

Draw text on the screen (but only once, so you need to call this every frame if you want the text to stay up).

Parameters

text	The string to write
nickname	The name you've assigned to the font you want to use
pixelX	The X-coordinate (in pixels) to start drawing
pixelY	The Y-coordinate (in pixels) to start drawing
angle	The angle at which to draw the text

Returns

The 2d point (in pixels) representing the bottom-most, right-most point where you could safely start drawing and not overwrite the string you just displayed.

Definition at line 106 of file TextRendering.cpp.

3.5.2.5 Vector2 DrawGameTextRaw (const String & *text,* const String & *nickname,* int *pixelX,* int *pixelY,* float *angle* = `0.0f`)

Renders text without first doing the transformation to screenspace. If you're drawing a lot of text and have already set up your projection, this can save you from needless matrix manipulation.

Parameters

text	The string to write
nickname	The name you've assigned to the font you want to use
pixelX	The X-coordinate (in pixels) to start drawing
pixelY	The Y-coordinate (in pixels) to start drawing
angle	The angle at which to draw the text

Returns

The 2d point (in pixels) representing the bottom-most, right-most point where you could safely start drawing and not overwrite the string you just displayed.

Definition at line 152 of file TextRendering.cpp.

3.5.2.6 Vector2 GetTextExtents (const String & *text*, const String & *nickname*)

If you're just interested in how much space a certain string will take up, but don't want to bother drawing it, this is the function for you.

Parameters

text	The string you care about
nickname	The font you would want to render it in

Returns

The 2d point (in pixels) representing the bottom-most, right-most point where you could safely start drawing and not overwrite the string if you had drawn it to the screen.

Definition at line 186 of file TextRendering.cpp.

3.5.2.7 float GetTextAscenderHeight (const String & *nickname*)

Gets the standard line height of this font – the number of pixels vertically occupied by a String containing capital letters. This does not include descenders like the bottom loop of a lowercase "g", only the height from the baseline. Useful for laying out multi-line text.

Parameters

nickname	The font whose height you want.

Returns

The number of vertical pixels used. Will return 0 if a font is not registered.

Definition at line 204 of file TextRendering.cpp.

3.6 Infrastructure/Textures.h File Reference

```
#include "../Util/StringUtil.h"
#include "../Infrastructure/Color.h"
```

Functions

- void InitializeTextureLoading ()
- void FinalizeTextureLoading ()
- void FlushTextureCache ()
- const int GetTextureReference (const String &name, bool optional=false)
- const int GetTextureReference (const String &filename, GLint clampmode, GLint filtermode, bool optional=false)
- const Vec2i GetTextureSize (const String &filename)
- bool PurgeTexture (const String &filename)
- bool GetRawImageData (const String &filename, std::vector< Color > &pixels)
- bool PixelsToPositions (const String &filename, Vector2List &positions, float gridSize, const Color &pixelColor, float tolerance=0.1f)

3.6.1 Detailed Description

A low-level C-style interface for managing textures. Most of the functionality is wrapped in the Actor classes, which is where you should probably be focusing. The function you might otherwise care about in here is FlushTextureCache.

Definition in file Textures.h.

3.6.2 Function Documentation

3.6.2.1 void InitializeTextureLoading ( )

Do whatever setup needs to be done at program start.

Definition at line 50 of file Textures.cpp.

3.6.2.2 void FinalizeTextureLoading ( )

Do whatever cleanup needs to be done at program end.

Definition at line 65 of file Textures.cpp.

3.6.2.3 void FlushTextureCache ( )

Goes through all loaded textures and marks them as dirty, so the next time they're referenced the file will be loaded up fresh from the disk. The effect of this is that you can update your texture files and see them in-game without having to restart.

Definition at line 82 of file Textures.cpp.

3.6.2.4 const int GetTextureReference (const String & *name*, bool *optional* = `false`)

Get the GL texture reference from a file path. If this is the first time a file has been referenced, it will be loaded from disk and formatted appropriately for OpenGL. Thus you use this function to both load your textures initially and to reference them afterwards.

Parameters

name	The path to the file to load. Must be readable by DevIL. (`http://openil.-sourceforge.net/`)
optional	If true, the engine won't complain if it can't load it.

Returns

The GLuint that OpenGL uses to reference the texture. If the number is negative, that means the texture couldn't be loaded or found.

Definition at line 93 of file Textures.cpp.

3.6.2.5 const int GetTextureReference (const String & *filename*, GLint *clampmode*, GLint *filtermode*, bool *optional* = `false`)

The same as the above function, but with some more available parameters if you want control over how to display this texture. See OpenGL docs for further explanation of how these parameters work.

Parameters

name	The path to the file to load. Must be readable by DevIL. (`http://openil.-sourceforge.net/`)
clampmode	Either GL_CLAMP or GL_REPEAT.
filtermode	One of: GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MI-PMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, or GL_LINEAR_MIPMAP_LINEAR
optional	If true, the engine won't complain if it can't load it.

Returns

The GLuint that OpenGL uses to reference the texture. If the number is negative, that means the texture couldn't be loaded or found.

Definition at line 327 of file Textures.cpp.

3.6.2.6 const Vec2i GetTextureSize (const String & *filename*)

Gets the dimensions for a loaded texture.

Parameters

filename	The path to the texture's file.

Returns

The width and height of the texture; if no texture is currently loaded from the given filename, will return (0, 0)

Definition at line 416 of file Textures.cpp.

3.6.2.7 bool PurgeTexture (const String & *filename*)

Remove a texture from memory. On the desktop this usually isn't an issue unless your game is long-running and uses lots of images, but those issues crop up earlier on mobile platforms.

Parameters

filename	The filename of the texture to be removed

Returns

Whether the texture was previously loaded into memory or not

Definition at line 98 of file Textures.cpp.

3.6.2.8 bool GetRawImageData (const String & *filename,* std::vector< Color > & *pixels*)

Use this function to get the raw pixel data of an image as a vector of Color structures.

Parameters

filename	The path to the file to load
pixels	The vector of Colors to populate

Returns

Whether the image was found and processed.

Definition at line 429 of file Textures.cpp.

3.6.2.9 bool PixelsToPositions (const String & *filename,* Vector2List & *positions,* float *gridSize,* const Color & *pixelColor,* float *tolerance* = `0.1f`)

Use this function to process an image into positional data, in other words, use an image as map or level data. For every pixel within the tolerance range of the specified color, a Vector2 is added to positions. Resulting positions will be in range imageSizeX ∗ gridSize, imageSizeY ∗ gridSize, and be centered on 0,0.

Parameters

filename	The path to the file to load.
positions	The vector of Vector2s to populate
gridSize	The size of the grid, i.e. how much space one pixel occupies.
pixelColor	The color to search the image for (channels in 0.0-1.0 range). NOTE: Alpha component is unused.
tolerance	The amount RGB channels can deviate from pixelColor.

Returns

Whether the image was found and processed.

Definition at line 509 of file Textures.cpp.

3.7 Input/Input.h File Reference

Functions

- void **keyboardInput** (GLFWwindow ∗window, int key, int scancode, int state, int mods)
- void **charInput** (GLFWwindow ∗window, unsigned int key)

3.7.1 Detailed Description

This file contains the C-level interface for accepting keyboard input. Because our windowing/input toolkit (GLFW) is C-based, it needs these loose functions to operate.

Definition in file Input.h.

3.8 Util/DrawUtil.h File Reference

```
#include "../Infrastructure/Vector2.h"
```

Functions

- void DrawCross (const Vector2 &point, float size)
- void DrawPoint (const Vector2 &point, float size)
- void DrawLine (const Vector2 &from, const Vector2 &to)
- bool HandleGLErrors ()

3.8.1 Detailed Description

A set of C-style utility functions to draw things on the screen. Good for quick debug work.

Definition in file DrawUtil.h.

3.8.2 Function Documentation

3.8.2.1 void DrawCross (const Vector2 & *point*, float *size*)

Draws a cross on the screen. Will be in whatever color you last passed in to a glColor∗ function.

Parameters

point	The point for the center of the cross (in world space)
size	How long each arm should be (in GL units)

Definition at line 36 of file DrawUtil.cpp.

3.8.2.2 void DrawPoint (const Vector2 & *point*, float *size*)

Draws a point on the screen. Will be in whatever color you last passed in to a glColor∗ function.

Parameters

point	The point to draw (in world space)
size	How large the point should appear (in GL units)

Definition at line 55 of file DrawUtil.cpp.

3.8.2.3 void DrawLine (const Vector2 & *from*, const Vector2 & *to*)

Draws a line on the screen. Will be in whatever color you last passed in to a glColor∗ function.

Parameters

from	The starting point in world space
to	The ending point in world space

Definition at line 74 of file DrawUtil.cpp.

3.8.2.4 bool HandleGLErrors ( )

Checks for OpenGL errors and prints any found to the log.

Returns

Whether or not any errors were found.

Definition at line 85 of file DrawUtil.cpp.

3.9 Util/FileUtil.h File Reference

```
#include "../Util/StringUtil.h"
```

Functions

- bool GetLinesFromFile (const String &fileName, StringList &outList)
- bool WriteLinesToFile (const String &fileName, const StringList &strings)
- bool AppendLineToFile (const String &fileName, const String &line)
- bool MakeDirectories (const String &path)
- const String ReadWholeFile (const String &fileName)
- const String GetStorageDirectory ()
- const String GetDocumentsPath ()
- const String GetExeName ()
- const long GetModificationTime (const String &fileName)

3.9.1 Detailed Description

A set of C-style utility functions to handle reading and writing files.

Definition in file FileUtil.h.

3.9.2 Function Documentation

3.9.2.1 bool GetLinesFromFile (const String & *fileName,* StringList & *outList*)

Gets the entire contents of a file with each line separated for you. (Doesn't do smart streaming, so don't pass it bajigabyte sized files.)

Parameters

fileName	The path to the file to load
outList	A StringList to which the file's lines will be appended, with each line being its own value.

Returns

True if we could read the file, false if we couldn't

Definition at line 50 of file FileUtil.cpp.

3.9.2.2 bool WriteLinesToFile (const String & *fileName,* const StringList & *strings*)

Writes a set of lines to a file. Will wipe out any existing file contents.

Parameters

fileName	The path of the file to write to
strings	The StringList to be written (each string will get its own line)

Returns

True if we could write to the file, false if we couldn't

Definition at line 80 of file FileUtil.cpp.

3.9.2.3 bool AppendLineToFile (const String & *fileName,* const String & *line*)

Appends a set of lines to a file.

Parameters

fileName	The path of the file to append to
strings	The StringList to be append (each string will get its own line)

Returns

True if we could write to the file, false if we couldn't

Definition at line 95 of file FileUtil.cpp.

3.9.2.4 bool MakeDirectories (const String & *path*)

Ensures that a path of directories exists – returns true if it does, false if it doesn't (if, for example, the program doesn't have sufficient permissions to create the path).

Parameters

path	The path to create

Returns

Whether it actually exists at the end of this function

Definition at line 107 of file FileUtil.cpp.

3.9.2.5 const String ReadWholeFile (const String & *fileName*)

Convenience function for reading an entire file as a single string. If there is no such file, and empty string will be returned.

Parameters

fileName	The file to read

Returns

The complete text of the file

Definition at line 68 of file FileUtil.cpp.

3.9.2.6 const String GetStorageDirectory ( )

Gives a system-appropriate writable directory for the use of logs, preference files, etc.

Returns

The path to the defined writable directory.

Definition at line 157 of file FileUtil.cpp.

3.9.2.7 const String GetDocumentsPath ( )

Returns a path to the My Documents directory on Windows, or the ∼/Documents folder on Mac. On Linux? Who knows?!

Returns

Path to the user's documents

Definition at line 180 of file FileUtil.cpp.

3.9.2.8 const String GetExeName ( )

Gives the name of the current executable.

Returns

The name of the current executable

Definition at line 203 of file FileUtil.cpp.

3.9.2.9 const long GetModificationTime (const String & *fileName*)

Returns the modification time of a file. Defined as number of seconds after a system-specific epoch, so not portable between operating systems, but useful for comparisons within a single build.

Parameters

fileName	The file to check

Returns

The modification time of the file. Returns 0 if file could not be found. NOTE: this actually *is* a valid modification time, so apologies if you happen to have a file that was modified exactly on your epoch time.

Definition at line 240 of file FileUtil.cpp.

3.10 Util/StringUtil.h File Reference

```
#include "../Infrastructure/Common.h"
#include "../Infrastructure/Vector2.h"
#include <set>
```

Typedefs

- typedef std::string **String**
- typedef std::vector< String > **StringList**
- typedef std::set< String > **StringSet**

Functions

- int StringToInt (const String &s)
- float StringToFloat (const String &s)
- bool StringToBool (const String &s)
- Vector2 StringToVector2 (const String &s)
- String IntToString (int val)
- String ULLIntToString (unsigned long long val)
- String FloatToString (float val)
- String BoolToString (bool val)
- String Vector2ToString (const Vector2 &val)
- String ToUpper (const String &s)
- String ToLower (const String &s)
- String JoinString (const StringList &list, const String &joinString="")
- StringList SplitString (const String &splitMe, const String &splitChars, bool bRemoveEmptyEntries=true)
- StringList SplitString (const String &splitMe)
- String TrimString (const String &trimMe, const String &trimChars)
- String TrimString (const String &trimMe)

3.10.1 Detailed Description

A set of C-style utility functions for manipulating strings.

Definition in file StringUtil.h.

3.10.2 Function Documentation

3.10.2.1 int StringToInt (const String & *s*)

Convert a string to an integer.

Parameters

s	The string to convert

Returns

The int representation thereof (0 if the string couldn't be parsed)

Definition at line 65 of file StringUtil.cpp.

3.10.2.2 float StringToFloat (const String & *s*)

Convert a string to a float.

Parameters

s	The string to convert

Returns

The float representation thereof (0.0f if the string couldn't be parsed)

Definition at line 72 of file StringUtil.cpp.

3.10.2.3 bool StringToBool (const String & *s*)

Convert a string to a bool. First looks to see if the string is "true" or "false" with any capitalization. If not, then it tries to convert it to an integer and uses that as the result (0 being false; any other value being true).

Parameters

s	The string to convert

Returns

The bool representation thereof

Definition at line 79 of file StringUtil.cpp.

3.10.2.4 Vector2 StringToVector2 (const String & *s*)

Converts a string of two numbers separated by a whitespace character to a Vector2. Returns a zero-length vector if the string is invalid. (0, 0)

Parameters

s	The string to convert

Returns

The Vector2 representation thereof

Definition at line 89 of file StringUtil.cpp.

3.10.2.5 String IntToString (int *val*)

Converts an integer to a string.

Parameters

val	The integer to convert

Returns

The string representation of the integer

Definition at line 104 of file StringUtil.cpp.

3.10.2.6 String ULLIntToString (unsigned long long *val*)

Converts extra long integers to a string.

Parameters

val	The unsigned long long integer to convert.

Returns

The string representation of the number.

Definition at line 114 of file StringUtil.cpp.

3.10.2.7 String FloatToString (float *val*)

Converts a float to a string.

Parameters

val	The float to convert

Returns

The string representation of the float

Definition at line 109 of file StringUtil.cpp.

3.10.2.8 String BoolToString (bool *val*)

Converts a bool to a string.

Parameters

val	The bool to convert

Returns

The string representation of the bool (in numeric form)

Definition at line 119 of file StringUtil.cpp.

3.10.2.9 String Vector2ToString (const Vector2 & *val*)

Converts a Vector2 to a string.

Parameters

val	The Vector2 to convert

Returns

The string representation of the Vector2 ("X Y")

Definition at line 124 of file StringUtil.cpp.

3.10.2.10 String ToUpper (const String & *s*)

Converts a string to all uppercase

Parameters

s	The string to convert

Returns

The transformed string

Definition at line 130 of file StringUtil.cpp.

3.10.2.11 String ToLower (const String & *s*)

Converts a string to all lowercase

Parameters

s	The string to convert

Returns

The transformed string

Definition at line 140 of file StringUtil.cpp.

3.10.2.12 String JoinString (const StringList & *list,* const String & *joinString* = " ")

Joins a list of strings into one, with a specified separator.

Parameters

The	string list to join
The	string to put between each entry in the final string.

Returns

The joined string.

Definition at line 185 of file StringUtil.cpp.

3.10.2.13 StringList SplitString (const String & *splitMe,* const String & *splitChars,* bool *bRemoveEmptyEntries* = `true`)

Splits a long string into a StringList.

Parameters

splitMe	The string to split up
splitChars	The set of delimiter characters to use for the splitting
bRemoveEmpty-Entries	Whether or not empty strings (resulting from multiple delimiters) should be added to the return list

Returns

The list of strings derived from the split

Definition at line 151 of file StringUtil.cpp.

3.10.2.14 StringList SplitString (const String & *splitMe*)

Splits a long string into a StringList wherever newlines or tabs appear.

Parameters

splitMe	The string to split

Returns

The list of strings derived from the split

Definition at line 180 of file StringUtil.cpp.

3.10.2.15 String TrimString (const String & *trimMe,* const String & *trimChars*)

Remove a series of duplicated characters from a string.

Parameters

trimMe	The string to trim
trimChars	The characters to remove

Returns

The trimmed string

Definition at line 201 of file StringUtil.cpp.

3.10.2.16 String TrimString (const String & *trimMe*)

Remove whitespace from the beginning or end of a string

Parameters

trimMe	The string to trim

Returns

The trimmed string

Definition at line 236 of file StringUtil.cpp.

Index

~Actor
    Actor, 19
~Console
    Console, 47
~ControllerManager
    ControllerManager, 59
~DeveloperLog
    DeveloperLog, 61
~GameManager
    GameManager, 67
~Message
    Message, 95
~MouseListener
    MouseListener, 99
~ParticleActor
    ParticleActor, 103
~PhysicsActor
    PhysicsActor, 109
~Renderable
    Renderable, 120
~TouchListener
    TouchListener, 143
_squareVertices
    Actor, 31

A
    Color, 44
AIBrain, 31
AIBrainState, 32
AIEvent, 33
AStarSearch< UserState >, 33
AStarSearch< UserState >::HeapCompare_f, 74
AStarSearch< UserState >::Node, 101
Abs
    MathUtil, 83
Accelerometer, 16
    GetData, 17
    GetInstance, 17
    NewAcceleration, 17
AcceptAutocomplete
    Console, 50
Actor, 17
    ~Actor, 19
    _squareVertices, 31
    Actor, 19
    AnimCallback, 26
    ChangeColorTo, 23
    ChangeSizeTo, 24
    ClearSpriteInfo, 25
    Create, 30
    GetAlpha, 22
    GetBoundingBox, 20
    GetClassName, 31
    GetColor, 22
    GetDrawShape, 22
    GetName, 28
    GetNamed, 28
    GetPosition, 21
    GetRotation, 21
    GetSelf, 30
    GetSize, 20
    GetSpriteFrame, 26
    GetSpriteTexture, 24
    GetTags, 28
    GetUVs, 27
    IsSpriteAnimPlaying, 26
    IsTagged, 27
    LevelUnloaded, 30
    LoadSpriteFrames, 25
    MoveTo, 23
    PlaySpriteAnimation, 25
    ReceiveMessage, 29
    Render, 30
    RotateTo, 23
    SetAlpha, 22
    SetColor, 21
    SetDrawShape, 22
    SetLayer, 29
    SetName, 28
    SetPosition, 20, 21
    SetRotation, 21
    SetScriptCreatedActor, 31
    SetSize, 20
    SetSprite, 25
    SetSpriteFrame, 26
    SetUVs, 26
    Tag, 27
    Untag, 27
    Update, 29
    UseDisplayList, 22
Actor.h
    actorDrawShape, 179
    spriteAnimationType, 179
actorDrawShape
    Actor.h, 179
Actors/Actor.h, 178
Actors/ParticleActor.h, 179
Actors/TextActor.h, 179
Add
    World, 172
AddButton
    UserInterface, 151
AddLog
    CompoundLog, 44
AddObjToTagList
    TagCollection, 134
AddToRuntimeTuningList
    Tuning, 147
AdjustWindow
    World, 169

AdvanceInputHistory
    Console, 49
AngleFromVector
    MathUtil, 87
AngleFromVectors
    MathUtil, 87
AnimCallback
    Actor, 26
AppendLineToFile
    FileUtil.h, 187
ApplyAngularImpulse
    PhysicsActor, 112
ApplyForce
    PhysicsActor, 111
ApplyLinearImpulse
    PhysicsActor, 112
ApplyLocalForce
    PhysicsActor, 111
ApplyTorque
    PhysicsActor, 111

B
    Color, 43
BeginContact
    World, 174
BoolToString
    StringUtil.h, 191
BoundingBox, 34
Broadcast
    Switchboard, 131

Camera, 35
    Destroy, 36
    GetClassName, 41
    GetFarClipDist, 40
    GetInstance, 36
    GetLockedActor, 37
    GetNearClipDist, 40
    GetPosition, 39
    GetViewCenter, 41
    GetViewRadius, 38
    GetWindowHeight, 37
    GetWindowWidth, 37
    GetWorldMaxVertex, 38
    GetWorldMinVertex, 38
    GetZ, 39
    GetZForViewRadius, 40
    LockTo, 37
    MoveTo, 39
    Render, 37
    Reset, 37
    ResizeCallback, 36
    SetFarClipDist, 41
    SetNearClipDist, 40
    SetPosition, 38, 39
    SetRotation, 40
    SetViewCenter, 41
    SetZByViewRadius, 40
    Update, 37
ChangeColorTo
    Actor, 23
ChangeSizeTo
    Actor, 24
    PhysicsActor, 113
Clamp
    MathUtil, 85, 86
    Vector2, 159
    Vector3, 165
ClearSpriteInfo
    Actor, 25
Color, 42
    A, 44
    B, 43
    Color, 42
    FromHexString, 43
    FromInts, 43
    G, 43
    R, 43
CompoundLog, 44
    AddLog, 44
    GetSystemLog, 45
    Log, 45
ConfigUpdater, 45
    ReceiveMessage, 46
Console, 46
    ~Console, 47
    AcceptAutocomplete, 50
    AdvanceInputHistory, 49
    Console, 47
    Enable, 48
    Execute, 50
    GetCompletions, 50
    GetInput, 48
    GetSpecialInputDown, 48
    GetTabWidth, 49
    GetToggleConsoleKey, 49
    IsEnabled, 48
    Render, 47
    SetPrompt, 49
    SetTabWidth, 49
    ToggleConsole, 48
    Update, 47
    WriteToOutput, 49
ConsoleLog, 50
    Log, 51
Controller, 51
    GetControllerID, 57
    GetLeftThumbVec2, 52
    GetLeftThumbstick, 53
    GetLeftTrigger, 53
    GetLeftVibration, 57
    GetRightThumbVec2, 53
    GetRightThumbstick, 53
    GetRightTrigger, 53
    GetRightVibration, 57
    GetState, 52
    IsAButtonDown, 54

IsBButtonDown, 54
IsBackButtonDown, 55
IsButtonDown, 53
IsConnected, 57
IsLeftBumperDown, 55
IsLeftThumbstickButtonDown, 54
IsLeftTriggerPressed, 55
IsRightBumperDown, 55
IsRightThumbstickButtonDown, 55
IsRightTriggerPressed, 56
IsStartButtonDown, 55
IsXButtonDown, 54
IsYButtonDown, 54
SetLeftVibration, 56
SetLeftVibrationRaw, 56
SetRightVibration, 56
SetRightVibrationRaw, 56
Setup, 52
UpdateState, 52
ControllerInput, 57
ControllerManager, 58
∼ControllerManager, 59
GetController, 59
GetInstance, 59
Setup, 59
UpdateState, 59
Create
Actor, 30
Cross
Vector2, 158
CustomInitPhysics
PhysicsActor, 111

DebugDrawBase, 59
DebugLine, 60
DeferredBroadcast
Switchboard, 131
DeltaAngle
MathUtil, 93
Destroy
Camera, 36
Renderable, 120
World, 170
DeveloperLog, 61
∼DeveloperLog, 61
Log, 61
Printf, 61
Distance
MathUtil, 84
Vector2, 157
Vector3, 163
DistanceSquared
Vector2, 157
Vector3, 163
Dot
Vector2, 158
Vector3, 164
DrawCross
DrawUtil.h, 185
DrawDebugLine
World, 175
DrawGameText
TextRendering.h, 181
DrawGameTextRaw
TextRendering.h, 181
DrawLine
DrawUtil.h, 186
DrawPoint
DrawUtil.h, 186
DrawUtil.h
DrawCross, 185
DrawLine, 186
DrawPoint, 186
HandleGLErrors, 186
DumpStack
LuaScriptingModule, 82

E
MathUtil, 93
eShapeType
PhysicsActor, 109
Enable
Console, 48
EndContact
World, 175
Epsilon
MathUtil, 94
EventHandler, 62
Execute
Console, 50
LuaConsole, 80
TestConsole, 136
ExecuteInScript
LuaScriptingModule, 81

FileLog, 62
FileLog, 63
FileLog, 63
Log, 63
MakeLogFileName, 63
FileUtil.h
AppendLineToFile, 187
GetDocumentsPath, 188
GetExeName, 188
GetLinesFromFile, 187
GetModificationTime, 188
GetStorageDirectory, 188
MakeDirectories, 187
ReadWholeFile, 188
WriteLinesToFile, 187
Finalize
LuaScriptingModule, 81
FinalizeTextureLoading
Textures.h, 183
FindNextMoveState, 64
FloatToString
StringUtil.h, 191
FlushTextureCache

Textures.h, 183
FollowMoveState, 64
FromHexString
Color, 43
FromInts
Color, 43
FullScreenActor, 65
FullScreenActor, 65
FullScreenActor, 65
GetClassName, 66
IsLocked, 66
ReceiveMessage, 66
SetLock, 66
FuzzyEquals
MathUtil, 90

G
Color, 43
GameManager, 66
∼GameManager, 67
GameManager, 67
GameManager, 67
IsProtectedFromUnloadAll, 68
ReceiveMessage, 68
Render, 67
SoundEnded, 68
Update, 67
GestureData, 68
GestureMagnitude, 69
Velocity, 69
GestureMagnitude
GestureData, 69
GetAlignment
TextActor, 138
GetAlpha
Actor, 22
GetAxisColor
GridActor, 72
GetBody
PhysicsActor, 112
GetBoundingBox
Actor, 20
TextActor, 140
GetClassName
Actor, 31
Camera, 41
FullScreenActor, 66
HUDActor, 75
ParticleActor, 106
PhysicsActor, 113
TextActor, 140
GetColor
Actor, 22
GetCompletions
Console, 50
LuaConsole, 80
TestConsole, 136
GetConsole
World, 174
GetController
ControllerManager, 59
GetControllerID
Controller, 57
GetCurrent
Interval, 78
GetCurrentTimeSeconds
World, 175
GetDT
World, 171
GetData
Accelerometer, 17
GetDefaultPath
Preferences, 117
GetDisplayString
TextActor, 138
GetDocumentsPath
FileUtil.h, 188
GetDrawShape
Actor, 22
GetExeName
FileUtil.h, 188
GetFarClipDist
Camera, 40
GetFirstRenderable
World, 176
GetFloat
Preferences, 115
Tuning, 146
GetFont
TextActor, 138
GetGameManager
World, 176
GetInput
Console, 48
GetInstance
Accelerometer, 17
Camera, 36
ControllerManager, 59
Preferences, 114
SoundDevice, 124
Switchboard, 130
TagCollection, 134
Tuning, 145
UserInterface, 150
World, 169
GetInt
Preferences, 115
Tuning, 146
GetInterval
GridActor, 72
GetLastRenderable
World, 176
GetLayer
Renderable, 120
GetLayerByName
World, 173
GetLayers

World, 174
GetLeftThumbVec2
Controller, 52
GetLeftThumbstick
Controller, 53
GetLeftTrigger
Controller, 53
GetLeftVibration
Controller, 57
GetLineColor
GridActor, 72
GetLineSpacing
TextActor, 139
GetLinesFromFile
FileUtil.h, 187
GetLockedActor
Camera, 37
GetLuaState
LuaScriptingModule, 82
GetMainWindow
World, 170
GetMajorAxis
MathUtil, 92
GetMaxCoord
GridActor, 73
GetMessageName
Message, 96
GetMinCoord
GridActor, 73
GetModificationTime
FileUtil.h, 188
GetName
Actor, 28
GetNamed
Actor, 28
GetNearClipDist
Camera, 40
GetObjectsTagged
TagCollection, 134
GetPhysicsWorld
World, 174
GetPosition
Actor, 21
Camera, 39
GetRawImageData
Textures.h, 184
GetRightThumbVec2
Controller, 53
GetRightThumbstick
Controller, 53
GetRightTrigger
Controller, 53
GetRightVibration
Controller, 57
GetRotation
Actor, 21
GetSelf
Actor, 30
GetSender
Message, 96
GetSize
Actor, 20
GetSpecialInputDown
Console, 48
GetSpriteFrame
Actor, 26
GetSpriteTexture
Actor, 24
GetState
Controller, 52
GetStorageDirectory
FileUtil.h, 188
GetString
Preferences, 115
Tuning, 146
GetSubscribersTo
Switchboard, 132
GetSubscriptionsFor
Switchboard, 132
GetSystemLog
CompoundLog, 45
GetTabWidth
Console, 49
GetTable
Preferences, 115
GetTagList
TagCollection, 134
GetTags
Actor, 28
GetTextAscenderHeight
TextRendering.h, 182
GetTextExtents
TextRendering.h, 182
GetTextureReference
Textures.h, 183
GetTextureSize
Textures.h, 184
GetTimeSinceSeconds
World, 175
GetToggleConsoleKey
Console, 49
GetTouchList
TouchListener, 143
GetUVs
Actor, 27
GetUserPrefsPath
Preferences, 117
GetValue
TypedMessage, 149
GetVariables
Tuning, 146
GetVector
Tuning, 146
GetVideoModes
World, 169
GetViewCenter

Camera, 41
GetViewRadius
Camera, 38
GetWindowHeight
Camera, 37
GetWindowWidth
Camera, 37
GetWorldDimensions
MathUtil, 92
GetWorldMaxVertex
Camera, 38
GetWorldMinVertex
Camera, 38
GetZ
Camera, 39
GetZForViewRadius
Camera, 40
GotoAIEvent, 69
GotoTargetAIEvent, 70
GridActor, 70
GetAxisColor, 72
GetInterval, 72
GetLineColor, 72
GetMaxCoord, 73
GetMinCoord, 73
GridActor, 71
GridActor, 71
Render, 73
SetAxisColor, 72
SetInterval, 72
SetLineColor, 71
SetMaxCoord, 73
SetMinCoord, 72
Update, 73
GwenRenderer, 73

HUDActor, 75
GetClassName, 75
Render, 75
HandleCharacter
UserInterface, 151
HandleGLErrors
DrawUtil.h, 186
HandleKey
UserInterface, 151

Infrastructure/Common.h, 180
Infrastructure/TextRendering.h, 180
Infrastructure/Textures.h, 182
InitPhysics
PhysicsActor, 111
Initialize
LuaScriptingModule, 81
World, 169
InitializeTextureLoading
Textures.h, 183
Input/Input.h, 185
InputBinding, 76
InputManager, 76
IsKeyDown, 77
IntToString
StringUtil.h, 190
Interval
GetCurrent, 78
Interval, 78
ShouldStep, 78
Step, 78
Interval$<$ T $>$, 77
IsAButtonDown
Controller, 54
IsAntiAliased
World, 177
IsBButtonDown
Controller, 54
IsBackButtonDown
Controller, 55
IsButtonDown
Controller, 53
IsConnected
Controller, 57
IsDestroyed
Renderable, 120
IsEnabled
Console, 48
IsFontRegistered
TextRendering.h, 181
IsHighResScreen
World, 177
IsKeyDown
InputManager, 77
IsLeftBumperDown
Controller, 55
IsLeftThumbstickButtonDown
Controller, 54
IsLeftTriggerPressed
Controller, 55
IsLocked
FullScreenActor, 66
IsPaused
SoundDevice, 125
IsPhysicsSetUp
World, 174
IsPlaying
SoundDevice, 125
IsProtectedFromUnloadAll
GameManager, 68
IsRightBumperDown
Controller, 55
IsRightThumbstickButtonDown
Controller, 55
IsRightTriggerPressed
Controller, 56
IsRuntimeTuned
Tuning, 148
IsSimulationOn
World, 176
IsSpriteAnimPlaying

Actor, 26
IsStartButtonDown
Controller, 55
IsTagged
Actor, 27
IsXButtonDown
Controller, 54
IsYButtonDown
Controller, 54

JoinString
StringUtil.h, 192

Length
Vector2, 157
Vector3, 163
LengthSquared
Vector2, 157
Vector3, 163
Lerp
MathUtil, 84
Vector2, 159
Vector3, 165
LevelUnloaded
Actor, 30
LoadLevel
World, 171
LoadSample
SoundDevice, 124
LoadSpriteFrames
Actor, 25
LoadedVariable, 79
LockTo
Camera, 37
Log
CompoundLog, 45
ConsoleLog, 51
DeveloperLog, 61
FileLog, 63
SystemLog, 133
Log10E
MathUtil, 93
Log2E
MathUtil, 94
LuaConsole, 79
Execute, 80
GetCompletions, 80
LuaConsole, 80
LuaConsole, 80
LuaScriptingModule, 81
DumpStack, 82
ExecuteInScript, 81
Finalize, 81
GetLuaState, 82
Initialize, 81
Prep, 81

MakeDirectories
FileUtil.h, 187
MakeLogFileName
FileLog, 63
MathUtil, 82
Abs, 83
AngleFromVector, 87
AngleFromVectors, 87
Clamp, 85, 86
DeltaAngle, 93
Distance, 84
E, 93
Epsilon, 94
FuzzyEquals, 90
GetMajorAxis, 92
GetWorldDimensions, 92
Lerp, 84
Log10E, 93
Log2E, 94
Max, 84
MaxFloat, 94
Min, 84
MinFloat, 94
Pi, 94
PiOver2, 94
PiOver4, 94
PixelsToWorldUnits, 92
RandomBool, 89
RandomFloat, 88
RandomFloatInRange, 88
RandomFloatWithError, 89
RandomInt, 87
RandomIntInRange, 88
RandomIntWithError, 88
RandomPointField, 90
RandomVector, 89
RoundToInt, 87
ScreenToWorld, 91
SmoothStep, 85
SplitBoundingBox, 93
ToDegrees, 86
ToRadians, 86
TwoPi, 94
VectorDeltaAngle, 93
VectorFromAngle, 86
WorldToScreen, 91
WorldUnitsToPixels, 92
Max
MathUtil, 84
Vector2, 159
Vector3, 165
MaxFloat
MathUtil, 94
Message, 94
∼Message, 95
GetMessageName, 96
GetSender, 96
Message, 95
MessageListener, 96
ReceiveMessage, 97

Min
  MathUtil, 84
  Vector2, 159
  Vector3, 164
MinFloat
  MathUtil, 94
MobileSimulator, 97
  MouseDownEvent, 98
  MouseMotionEvent, 98
  MouseUpEvent, 98
  Update, 97
MouseDownEvent
  MobileSimulator, 98
  MouseListener, 99
  UserInterface, 151
MouseListener, 99
  ∼MouseListener, 99
  MouseDownEvent, 99
  MouseListener, 99
  MouseMotionEvent, 99
  MouseUpEvent, 100
  MouseWheelEvent, 100
  MouseListener, 99
MouseMotionEvent
  MobileSimulator, 98
  MouseListener, 99
  UserInterface, 150
MouseUpEvent
  MobileSimulator, 98
  MouseListener, 100
  UserInterface, 151
MouseWheelEvent
  MouseListener, 100
  UserInterface, 151
MoveTo
  Actor, 23
  Camera, 39
  PhysicsActor, 113
MoveWindow
  World, 170

NameLayer
  World, 173
NamedEventAIEvent, 100
Negate
  Vector2, 160
  Vector3, 165
NewAcceleration
  Accelerometer, 17
Normalize
  Vector2, 158
  Vector3, 164

One
  Vector2, 161
  Vector3, 166
OverrideFloat
  Preferences, 117
OverrideInt
  Preferences, 116
OverrideString
  Preferences, 117

ParticleActor, 102
  ∼ParticleActor, 103
  GetClassName, 106
  ParticleActor, 103
  ParticleActor, 103
  Render, 103
  SetAttractor, 105
  SetAttractorStrength, 106
  SetEndColor, 104
  SetEndScale, 104
  SetGravity, 105
  SetMaxParticles, 106
  SetMaxSpeed, 105
  SetMinSpeed, 105
  SetParticleLifetime, 104
  SetParticlesPerSecond, 104
  SetSpeedRange, 105
  SetSpread, 104
  SetSystemLifetime, 104
  Update, 103
ParticleActor::Particle, 101
PathFinder, 106
PathFinderMove, 107
PausePhysics
  World, 172
PauseSimulation
  World, 171
PauseSound
  SoundDevice, 125
PhysicsActor, 107
  ∼PhysicsActor, 109
  ApplyAngularImpulse, 112
  ApplyForce, 111
  ApplyLinearImpulse, 112
  ApplyLocalForce, 111
  ApplyTorque, 111
  ChangeSizeTo, 113
  CustomInitPhysics, 111
  eShapeType, 109
  GetBody, 112
  GetClassName, 113
  InitPhysics, 111
  MoveTo, 113
  PhysicsActor, 109
  PhysicsActor, 109
  ResetBody, 112
  RotateTo, 113
  SetDensity, 109
  SetDrawSize, 112
  SetFixedRotation, 110
  SetFriction, 109
  SetGroupIndex, 110
  SetIsSensor, 110
  SetPosition, 113
  SetRestitution, 110

SetRotation, 113
SetShapeType, 110
SetSize, 112
Pi
MathUtil, 94
PiOver2
MathUtil, 94
PiOver4
MathUtil, 94
PixelsToPositions
Textures.h, 184
PixelsToWorldUnits
MathUtil, 92
PlaySound
SoundDevice, 124
PlaySpriteAnimation
Actor, 25
PreDestroy
Renderable, 121
Preferences, 114
GetDefaultPath, 117
GetFloat, 115
GetInstance, 114
GetInt, 115
GetString, 115
GetTable, 115
GetUserPrefsPath, 117
OverrideFloat, 117
OverrideInt, 116
OverrideString, 117
SavePreferences, 114
SetFloat, 116
SetInt, 116
SetString, 116
Prep
LuaScriptingModule, 81
Printf
DeveloperLog, 61
PurgeDebugDrawing
World, 175
PurgeTexture
Textures.h, 184

R
Color, 43
RandomBool
MathUtil, 89
RandomFloat
MathUtil, 88
RandomFloatInRange
MathUtil, 88
RandomFloatWithError
MathUtil, 89
RandomInt
MathUtil, 87
RandomIntInRange
MathUtil, 88
RandomIntWithError
MathUtil, 88
RandomPointField
MathUtil, 90
RandomVector
MathUtil, 89
Ray2, 118
ReadWholeFile
FileUtil.h, 188
ReceiveMessage
Actor, 29
ConfigUpdater, 46
FullScreenActor, 66
GameManager, 68
MessageListener, 97
TextActor, 139
World, 176
RecoverMoveState, 118
Reflect
Vector2, 159
Vector3, 164
RegisterConsole
World, 174
RegisterFont
TextRendering.h, 180
Remove
World, 173
RemoveObjFromTagList
TagCollection, 135
RemoveUIElement
UserInterface, 152
Render
Actor, 30
Camera, 37
Console, 47
GameManager, 67
GridActor, 73
HUDActor, 75
ParticleActor, 103
Renderable, 120
Sentient, 123
TextActor, 137
UserInterface, 150
World, 177
Renderable, 119
∼Renderable, 120
Destroy, 120
GetLayer, 120
IsDestroyed, 120
PreDestroy, 121
Render, 120
Renderable, 119
Update, 120
RenderableIterator, 121
Reset
Camera, 37
ResetBody
PhysicsActor, 112
ResetWorld
World, 170

ResizeCallback
    Camera, 36
ResumePhysics
    World, 172
ResumeSimulation
    World, 171
Rotate
    Vector2, 160
RotateTo
    Actor, 23
    PhysicsActor, 113
RoundToInt
    MathUtil, 87

SavePreferences
    Preferences, 114
ScreenToWorld
    MathUtil, 91
ScriptExec
    World, 171
SearchInterface, 121
SendAllMessages
    Switchboard, 132
Sentient, 122
    Render, 123
    Update, 123
SetAlignment
    TextActor, 138
SetAlpha
    Actor, 22
SetAttractor
    ParticleActor, 105
SetAttractorStrength
    ParticleActor, 106
SetAxisColor
    GridActor, 72
SetBackgroundColor
    World, 172
SetColor
    Actor, 21
SetDT
    World, 177
SetDensity
    PhysicsActor, 109
SetDisplayString
    TextActor, 138
SetDrawShape
    Actor, 22
SetDrawSize
    PhysicsActor, 112
SetEndColor
    ParticleActor, 104
SetEndScale
    ParticleActor, 104
SetFarClipDist
    Camera, 41
SetFixedRotation
    PhysicsActor, 110
SetFloat
    Preferences, 116
    Tuning, 147
SetFont
    TextActor, 138
SetFriction
    PhysicsActor, 109
SetGameManager
    World, 176
SetGravity
    ParticleActor, 105
SetGroupIndex
    PhysicsActor, 110
SetHighResolutionScreen
    World, 177
SetInt
    Preferences, 116
    Tuning, 147
SetInterval
    GridActor, 72
SetIsSensor
    PhysicsActor, 110
SetLayer
    Actor, 29
SetLeftVibration
    Controller, 56
SetLeftVibrationRaw
    Controller, 56
SetLineColor
    GridActor, 71
SetLineSpacing
    TextActor, 139
SetLock
    FullScreenActor, 66
SetMaxCoord
    GridActor, 73
SetMaxParticles
    ParticleActor, 106
SetMaxSpeed
    ParticleActor, 105
SetMinCoord
    GridActor, 72
SetMinSpeed
    ParticleActor, 105
SetName
    Actor, 28
SetNearClipDist
    Camera, 40
SetPan
    SoundDevice, 126
SetParticleLifetime
    ParticleActor, 104
SetParticlesPerSecond
    ParticleActor, 104
SetPosition
    Actor, 20, 21
    Camera, 38, 39
    PhysicsActor, 113
    TextActor, 139

SetPrompt
  Console, 49
SetRestitution
  PhysicsActor, 110
SetRightVibration
  Controller, 56
SetRightVibrationRaw
  Controller, 56
SetRotation
  Actor, 21
  Camera, 40
  PhysicsActor, 113
  TextActor, 139
SetScriptCreatedActor
  Actor, 31
SetShapeType
  PhysicsActor, 110
SetSideBlockers
  World, 175
SetSize
  Actor, 20
  PhysicsActor, 112
SetSoundCallback
  SoundDevice, 126
SetSpeedRange
  ParticleActor, 105
SetSpread
  ParticleActor, 104
SetSprite
  Actor, 25
SetSpriteFrame
  Actor, 26
SetString
  Preferences, 116
  Tuning, 147
SetSystemLifetime
  ParticleActor, 104
SetTabWidth
  Console, 49
SetUVs
  Actor, 26
SetVector
  Tuning, 147
SetViewCenter
  Camera, 41
SetVolume
  SoundDevice, 126
SetZByViewRadius
  Camera, 40
Setup
  Controller, 52
  ControllerManager, 59
SetupPhysics
  World, 170
ShouldStep
  Interval, 78
ShowChoiceBox
  UserInterface, 152
Shutdown
  SoundDevice, 126
  UserInterface, 151
SmoothStep
  MathUtil, 85
SoundDevice, 123
  GetInstance, 124
  IsPaused, 125
  IsPlaying, 125
  LoadSample, 124
  PauseSound, 125
  PlaySound, 124
  SetPan, 126
  SetSoundCallback, 126
  SetVolume, 126
  Shutdown, 126
  StopSound, 125
  Update, 126
SoundEnded
  GameManager, 68
SpatialGraph, 127
SpatialGraphKDNode, 127
SpatialGraphManager, 128
SplitBoundingBox
  MathUtil, 93
SplitString
  StringUtil.h, 192, 193
spriteAnimationType
  Actor.h, 179
StartGame
  World, 170
StartMoveState, 129
StartRecoverMoveState, 129
Step
  Interval, 78
StopGame
  World, 171
StopSound
  SoundDevice, 125
StringToBool
  StringUtil.h, 190
StringToFloat
  StringUtil.h, 190
StringToInt
  StringUtil.h, 189
StringToVector2
  StringUtil.h, 190
StringUtil.h
  BoolToString, 191
  FloatToString, 191
  IntToString, 190
  JoinString, 192
  SplitString, 192, 193
  StringToBool, 190
  StringToFloat, 190
  StringToInt, 189
  StringToVector2, 190
  ToLower, 192

ToUpper, 192
TrimString, 193
ULLIntToString, 191
Vector2ToString, 191
SubscribeTo
Switchboard, 131
Switchboard, 130
Broadcast, 131
DeferredBroadcast, 131
GetInstance, 130
GetSubscribersTo, 132
GetSubscriptionsFor, 132
SendAllMessages, 132
SubscribeTo, 131
UnsubscribeFrom, 131
Update, 131
SystemLog, 133
Log, 133

TGenericCallback< ClassInstance, ParamType >, 141
Tag
Actor, 27
TagCollection, 133
AddObjToTagList, 134
GetInstance, 134
GetObjectsTagged, 134
GetTagList, 134
RemoveObjFromTagList, 135
TestConsole, 135
Execute, 136
GetCompletions, 136
TestConsole, 136
TestConsole, 136
TextActor, 136
GetAlignment, 138
GetBoundingBox, 140
GetClassName, 140
GetDisplayString, 138
GetFont, 138
GetLineSpacing, 139
ReceiveMessage, 139
Render, 137
SetAlignment, 138
SetDisplayString, 138
SetFont, 138
SetLineSpacing, 139
SetPosition, 139
SetRotation, 139
TextActor, 137
TextActor, 137
TextActor.h
TextAlignment, 179
TextAlignment
TextActor.h, 179
TextRendering.h
DrawGameText, 181
DrawGameTextRaw, 181
GetTextAscenderHeight, 182
GetTextExtents, 182
IsFontRegistered, 181
RegisterFont, 180
UnRegisterFont, 181
TextureCacheEntry, 140
Textures.h
FinalizeTextureLoading, 183
FlushTextureCache, 183
GetRawImageData, 184
GetTextureReference, 183
GetTextureSize, 184
InitializeTextureLoading, 183
PixelsToPositions, 184
PurgeTexture, 184
Tick
World, 177
TickAndRender
World, 177
TimerAIEvent, 141
ToDegrees
MathUtil, 86
ToLower
StringUtil.h, 192
ToRadians
MathUtil, 86
ToUpper
StringUtil.h, 192
ToggleConsole
Console, 48
Touch, 142
TouchEndEvent
TouchListener, 143
TouchListener, 142
~TouchListener, 143
GetTouchList, 143
TouchEndEvent, 143
TouchListener, 143
TouchMotionEvent, 143
TouchStartEvent, 143
TouchListener, 143
TouchMotionEvent
TouchListener, 143
TouchStartEvent
TouchListener, 143
Traversal, 144
TraversalAIEvent, 144
TrimString
StringUtil.h, 193
Tuning, 145
AddToRuntimeTuningList, 147
GetFloat, 146
GetInstance, 145
GetInt, 146
GetString, 146
GetVariables, 146
GetVector, 146
IsRuntimeTuned, 148
SetFloat, 147
SetInt, 147

SetString, 147
SetVector, 147
TwoPi
MathUtil, 94
TypedMessage
GetValue, 149
TypedMessage, 149
TypedMessage, 149
TypedMessage< T >, 148

ULLIntToString
StringUtil.h, 191
UnRegisterFont
TextRendering.h, 181
UnitX
Vector2, 161
Vector3, 166
UnitY
Vector2, 161
Vector3, 166
UnitZ
Vector3, 166
UnloadAll
World, 176
UnsubscribeFrom
Switchboard, 131
Untag
Actor, 27
Update
Actor, 29
Camera, 37
Console, 47
GameManager, 67
GridActor, 73
MobileSimulator, 97
ParticleActor, 103
Renderable, 120
Sentient, 123
SoundDevice, 126
Switchboard, 131
UpdateLayer
World, 173
UpdateState
Controller, 52
ControllerManager, 59
UseDisplayList
Actor, 22
UserInterface, 149
AddButton, 151
GetInstance, 150
HandleCharacter, 151
HandleKey, 151
MouseDownEvent, 151
MouseMotionEvent, 150
MouseUpEvent, 151
MouseWheelEvent, 151
RemoveUIElement, 152
Render, 150
ShowChoiceBox, 152
Shutdown, 151
Util/DrawUtil.h, 185
Util/FileUtil.h, 186
Util/StringUtil.h, 189

ValidateMoveState, 153
Vec2i, 153
Vec2ui, 154
Vec3i, 154
Vec3ui, 155
Vector2, 155
Clamp, 159
Cross, 158
Distance, 157
DistanceSquared, 157
Dot, 158
Length, 157
LengthSquared, 157
Lerp, 159
Max, 159
Min, 159
Negate, 160
Normalize, 158
One, 161
Reflect, 159
Rotate, 160
UnitX, 161
UnitY, 161
Vector2, 156, 157
X, 160
Y, 160
Zero, 160
Vector2ToString
StringUtil.h, 191
Vector3, 161
Clamp, 165
Distance, 163
DistanceSquared, 163
Dot, 164
Length, 163
LengthSquared, 163
Lerp, 165
Max, 165
Min, 164
Negate, 165
Normalize, 164
One, 166
Reflect, 164
UnitX, 166
UnitY, 166
UnitZ, 166
Vector3, 162, 163
X, 166
Y, 166
Z, 166
Zero, 166
VectorDeltaAngle
MathUtil, 93
VectorFromAngle

MathUtil, 86
Velocity
GestureData, 69

WakeAllPhysics
World, 174
World, 167
Add, 172
AdjustWindow, 169
BeginContact, 174
Destroy, 170
DrawDebugLine, 175
EndContact, 175
GetConsole, 174
GetCurrentTimeSeconds, 175
GetDT, 171
GetFirstRenderable, 176
GetGameManager, 176
GetInstance, 169
GetLastRenderable, 176
GetLayerByName, 173
GetLayers, 174
GetMainWindow, 170
GetPhysicsWorld, 174
GetTimeSinceSeconds, 175
GetVideoModes, 169
Initialize, 169
IsAntiAliased, 177
IsHighResScreen, 177
IsPhysicsSetUp, 174
IsSimulationOn, 176
LoadLevel, 171
MoveWindow, 170
NameLayer, 173
PausePhysics, 172
PauseSimulation, 171
PurgeDebugDrawing, 175
ReceiveMessage, 176
RegisterConsole, 174
Remove, 173
Render, 177
ResetWorld, 170
ResumePhysics, 172
ResumeSimulation, 171
ScriptExec, 171
SetBackgroundColor, 172
SetDT, 177
SetGameManager, 176
SetHighResolutionScreen, 177
SetSideBlockers, 175
SetupPhysics, 170
StartGame, 170
StopGame, 171
Tick, 177
TickAndRender, 177
UnloadAll, 176
UpdateLayer, 173
WakeAllPhysics, 174
WorldToScreen
MathUtil, 91
WorldUnitsToPixels
MathUtil, 92
WriteLinesToFile
FileUtil.h, 187
WriteToOutput
Console, 49

X
Vector2, 160
Vector3, 166
XboxButtonBindRecord, 178

Y
Vector2, 160
Vector3, 166

Z
Vector3, 166
Zero
Vector2, 160
Vector3, 166