Welcome to the Lightning game engine! View the release notes for the latest release in the **ReleaseNotes.docx** file.

This API reference will hopefully provide you, the developer, with everything you need to develop games using the engine and to utilise Lightning to its maximum potential in order to create awesome games.

Any comments about or issues with the documentation, the Lightning game engine, or feature requests should be directed to **Starfrost#9088** on Discord.

Underlined subchapters are new for version 1.1.  
**This version of the Lightning manual is a pre-release!**

**Contents  
1. General Overview**1.1. Lightning API & Manual Style  
1.2. Namespaces  
1.3. Standard .NET types used  
1.4. The Renderable Class  
1.5. Asset Managers **2. Getting Started**2.1. Setting Up using the Scene Manager  
2.2. If you don’t want to use the Scene Manager: Creating a Renderer manually  
2.3. Renderer Settings  
2.4. Delta Time & Framerate Management **3. Modules**3.1. LightningGL  
3.2. LightningPackager **4. Scenes**4.1. The Scene Manager  
4.2. Scenes Overview  
4.3. Scene Management  
4.4. Shutdown **5. Settings (Global & Local)**5.1. GlobalSettings Overview  
5.2. Global Settings Values  
5.2.1. General section  
5.2.2. Graphics section  
5.2.3. Localisation section  
5.2.4. Requirements section  
5.2.5. Scene section  
5.3. LocalSettings (and an introduction to INI files…) **6. Cameras**6.1. Camera Overview  
6.2. Camera Shake  
6.3. Camera Velocity **7. Input, Event Handling & Collision**7.1. Input & Event Handling Overview  
7.2. The Key class  
7.3. The MouseButton class  
7.4. Hit Testing **8. Primitives**8.1. Primitive Drawing Methods Overview  
8.1.1. Pixels  
8.1.2. Lines  
8.1.3. Circles  
8.1.4. Rectangles  
8.1.5. Rounded Rectangles  
8.1.6. Triangles  
8.1.7. Polygons  
8.1.8. Text using the Debug Font **9. Textures (and atlases)**9.1. Texture Loading & Drawing  
9.2. Texture API  
9.3. Texture Atlases **10. Animated Textures**10.1. Animated Texture Overview  
10.2. Animation Cycles  
**11. The Lightning Animation Editor**  
11.1. The Lightning Animation Engine  
11.2. Animation Editor Overview (AnimTool.exe)  
11.3. Animation Editor Tools **12. Lighting**12.1. Lighting Overview  
12.2. The Light Class  
12.3. Environmental Light & The Light Manager **13. Particle Effects**  
13.1. Particle System Overview  
13.1.1. The ParticleManager Class  
13.2. Particle Effect Properties  
13.3. Particle Effect Modes  
13.4. Playing Particle Effects **14. UI**14.1. UI System Overview  
14.2. UI Specific Events  
14.2.1. OnKeyPressed  
14.2.2. OnKeyReleased  
14.2.3. OnMousePressed  
14.2.4. OnMouseReleased  
14.2.5. OnMouseEnter  
14.2.6. OnMouseLeave  
14.2.7. OnFocusGained  
14.2.8. OnFocusLost  
14.2.9. OnMouseMove  
14.2.10. OnRender  
14.2.11. OnShutdown  
15.3. UI Gadgets  
15.3.1. Gadget Base Properties  
15.3.2. Buttons  
15.3.3. ListBoxes  
15.3.4. ListBoxItems  
15.3.5. TextBoxes  
15.3.6. CheckBoxes  
15.4. Expanding the UI System with Custom Gadgets **15. Text Rendering & Font Management**16.1. The Font Manager  
16.1.1. The FontManager Class  
16.2. Loading Fonts  
16.2.1. The Font Class  
16.3. Drawing Text  
16.4. Text Example Code **17. Localisation**17.1. Localisation Overview  
17.2. Localisation Settings in Engine.ini  
17.3. Localised Strings **18. Audio**18.1. Audio Manager Overview  
18.1.1. The AudioManager Class  
18.2. Audio Files  
18.2.1. The AudioFile Class  
18.3. Playing & Managing Audio **19. System Information & Feature Detection**19.1.The SystemInfo Class  
19.1.1. The SystemInfoCPU Class  
19.1.2. The SystemInfoCPUCapabilities Enum  
19.1.3. The SystemInfoOperatingSystem Enum  
19.2. Enforcing System Requirements  
19.3. Detecting Features **20. Packaging**20.1. Packaging Your Game & Running It Packaged  
20.1.1. Deleting Package Files on Exit.  
20.1.2. Package Debugging using the InitSettings Class and Command-Line Options  
20.2. The MakePackage tool  
20.3. Optional MakePackage Arguments **21. Debugging**21.1. Debugging Your Game  
21.2. The FPS Meter  
21.3. The Performance Profiler **22. Advanced Usage (How to Interface with SDL)**22.1. Interfacing with SDL  
22.2. Lightning + NativeAOT  
22.3. Extending the Engine **23. Error Message Reference**

**1. General Overview**

* 1. **Lightning API & Manual Style**

The Lightning API is based on the concept of Renderers, asset managers (see Chapter 1.5)) and, optionally, scenes.  
  
Each Renderer is a rendering context that allows you to use Managers.   
Managers are the various parts of Lightning that allow specific rendering operations, such as rendering lighting or input, to occur. It is generally required to pass the current Renderer being used to any rendering APIs that you call.

A scene is simply an “area” of a game and is an optional construct – the Scene Manager can be turned off with the **DontUseSceneManager** GlobalSetting. It provides ready-made basic handling of startup and shutdown as well as some very basic event handling.

Exceptions are implemented using a custom exception class, **NCException**.

Lightning uses British English, except where the standard library mandates American English. This may be changed based on feedback.

In this manual, chapter names other than the chapter currently being referred to are referred to using *italics*, and properties being introduced for the first time will use **bold** text*.* APIs will be referred to in this manual with the syntax **ClassName::API**. Any warnings the programmer may need to heed use **bold, red text.**

Any method without a return type specified is assumed to be **void**.

**1.2. Namespaces**

**LightningGL**Provides all of the primary rendering APIs.  
**LightningPackager**Provides all of the APIs for packaging and managing WAD files.  
**LightningBase**Provides access to unmanaged SDL2, SDL2\_image, SDL2\_mixer, SDL2\_ttf, and my fork of SDL2\_gfx APIs and functions, as well as the Global and Local Settings APIs.  
**NuCore.Utilities**Provides utility APIs for the engine. **Some of the classes here may be useful to developers, but they will not be documented and no support will be provided.**  
**NuCore.Utilities.Lightning**Private utility APIs for Lightning. Do not use.

**1.3. Standard .NET types used**

**System.Drawing.Color**Defines an ARGB-formatted color.  
**Documentation**: <https://docs.microsoft.com/en-us/dotnet/api/system.drawing.color?view=net-6.0>

**System,Numerics.Vector2**Defines a two-dimensional vector.  
**Documentation:** <https://docs.microsoft.com/en-us/dotnet/api/system.numerics.vector2?view=net-6.0>

**System.Runtime.InteropServices.Architecture**Defines valid .NET Core processor architectures.  
**Documentation:** <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.architecture?view=net-6.0>

**1.4. The Renderable class**

Any renderable object is expected to inherit from the Renderable class and override its Draw method – this provides the object with a few basic properties, such as position and size, that can be manipulated in order to make drawing the object easier for the programmer. UI elements are extended by inheriting from the Gadget class.

**Properties**

**Position**The position of this renderable. If **SnapToScreen** is false. Can be changed at runtime.

**Size**The Size of this Renderable.

**SnapToScreen**Determines if the Renderable will be drawn in world-relative space or camera-relative space.

**Methods**

**Draw**

Draws the Renderable.

The Renderable class can be extended to add your own properties (any .NET class is supported, including any custom classes) and drawing code, like so:

public class MyRenderable : Renderable  
{  
 public bool MyRenderableProperty { get; set; }

public override void Draw(Renderer cRenderer)  
{

PrimitiveRenderer.DrawText(cRenderer, “This is MyRenderable”, new(100, 100), Color.Yellow);

}

}

**new()** can be used as a shorthand to omit the class name and any required **using** statements for the namespace the class is in. References to assemblies are still required when using this syntax.

**1.5. Asset Managers**

Asset Managers are the third primary facet of Lightning. Put simply, asset managers load, unload, store, and manage a particular class of object. Lightning provides asset managers for scenes, textures (experimental), lighting, UI, audio, and various other aspects of the engine API. The asset managers provide a consistent API – to add an asset to an asset manager, call that asset manager’s **AddAsset** method with an asset object you wish to load. Lightning will automatically load the asset and add it to the Asset Manager’s internal asset list.

In a similar fashion, remove an asset from an asset manager, call that asset managers’ **RemoveAsset** method with the asset you wish to unload and remove, and Lightning will unload and remove the asset for you. As expected, the asset will also no longer be rendered.

**2. Getting Started**

**2.1. Setting Up using the Scene Manager**

The Lightning game engine includes Visual Studio templates to allow you to get started quickly. For more information, refer to the **Quickstart.docx** document included with your engine installation. It is assumed that you have basic C# knowledge before using the engine. If you don’t want to use the Scene Manager and want to start manually

**2.2. If you don’t want to use the Scene Manager: Creating a Renderer Manually using RendererSettings**

**WARNING: The ability to initialise Lightning without the Scene Manager will be removed in Lightning 1.2.0, and the Scene Manager will become mandatory.**

Lightning also supports being used without the Scene Manager, although using the Scene Manager is **highly recommended**. To do this, first create a Lightning Game Project as normal, add **DontUseSceneManager=true** (for further information on the settings that are available in this file, please refer to *Chapter 5: Settings)* to the **[General]** section of the **Content\Engine.ini** file in your project, and then delete the **Scenes** folder and any files that may be in it.

Then, in your Main function (usually in **Program.cs**, unless you renamed it), you first have to initialise the engine as normal:

Lightning.Init(args);

where **args** is the command-line arguments provided to the application.

Then you have to create a Renderer:

Renderer renderer = new Renderer();

Then, configure the **RendererSettings** manually (see *Chapter 2.3: RendererSettings* for documentation on the **RendererSettings** class):

RendererSettings RendererSettings = new RendererSettings();

Then the Renderer has to be started using the instance of **RendererSettings** you have created:

renderer.Start(RendererSettings);

If you compile the code as it is now, the Renderer will immediately exit. This is because you have not provided a main loop. The main loop in Lightning is simple, and merely consists of:

while (renderer.Run())

{

// rendering code  
renderer.Render();

}

The call to **Renderer::Run** allows Lightning to start the frame, and the call to **Renderer::Render** allows all of Lightning’s asset managers, including the Light Manager, the UI Manager, and several others, to perform rendering tasks.

At runtime, the **RendererSettings** can be accessed using the **Settings** property of the Renderer.

Here is code for the **basic** example, which uses this method of initialisation:

using LightningGL; // use lightninggl

using System.Drawing; // for color

using System.Numerics;

//Basic Lightning Example (Scene Manager NOT used)

//©2022 starfrost, August 12, 2022

// Initialise Lightning

Lightning.Init(args);

Renderer Renderer = new Renderer();

Renderer.Start(new RendererSettings

{

Title = "Basic Lightning2 Demo"

}); // use default Renderersettings except title

while (Renderer.Run())

{

PrimitiveRenderer.DrawText(Renderer, "Basic example", new Vector2(100, 100), Color.White); // no fonts loaded so we use the debug font

Renderer.Render();

}

**2.3. RendererSettings**

These values define settings for the Renderer. If you are using the Scene Manager, these settings can be acquired using equivalent GlobalSettings. For further information on GlobalSettings, refer to *Chapter 5.1. Global Settings*)

**string Title**The Renderer title of the Renderer. Can be modified at runtime. Optional.

**Vector2 Position**The position of the Renderer on the screen. The default value is the value of **GlobalSettings::PositionX** and **GlobalSettings::PositionY**, which is set based on the screen resolution in such a way as to position the Renderer in the middle of the screen

**Vector2 Size**The window size or resolution for this Renderer. The valid range of values is between 192x48 and the current screen resolution. Default is (960, 640). Can be changed at runtime.

**SDL\_RendererFlags RendererFlags**SDL\_RendererFlags for the Renderer. See [https://wiki.libsdl.org/SDL\_RendererFlags](https://wiki.libsdl.org/SDL_WindowFlags). Default is the content of **GlobalSettings::RendererFlags**, which is **SDL\_RENDERER\_SHOWN**.

**SDL\_RendererFlags RenderFlags**SDL\_RendererFlags for the renderer. See <https://wiki.libsdl.org/SDL_RendererFlags> Default is the content of **GlobalSettings::RenderFlags**, which is **0.**

**Int ID**Internal only value used for storing the SDL2 Window ID of the Renderer.   
**Does not matter for general use. Read-only.**

**IntPtr WindowHandle**Internal only pointer to unmanaged memory containing the SDL\_Window struct of the Renderer. **Can only be acquired from user code, not set.**

**IntPtr RendererHandle**Internal only pointer to unmanaged memory containing the SDL\_Renderer struct of the renderer. **Can only be acquired from user code, not set.**

**Camera Camera**Camera used for this Renderer. See chapter 6 for an overview of this camera – don’t use this property directly

**BackgroundColor**The background color of the Renderer.

**2.4. Delta Time & Framerate Management**

The current delta time used for calculations of velocity throughout the Lightning engine is stored in the **Renderer::DeltaTime** property. To use delta-time for velocity calculations, simply multiply the value you are using by it, like so:

double realVelocity = velocity \* Renderer.DeltaTime;

You must be using a method that receives a Renderer as a parameter currently in order to use delta time. To set a maximum framerate, use the **MaxFPS** global setting. A value of 0 or lower will turn off the frame limiter.

**3. Modules**

Lightning modules are independent sections of the engine that provide specific functionality. There are currently two modules, although more will be added in future releases. Documentation for all modules is contained within this manual.

**3.1. LightningGL:**

This module provides graphical rendering and event handling capabilities, and is the primary component of the Lightning Game Engine.

**3.2. LightningPackager:**

Provides WAD file packaging and extraction functions. For further information on packaging, refer to *Chapter 18: Packaging*.

**4. Scenes**

Scenes are an “area” of a game. An area of a game could be something such as the menu, the main game, options, mini-games, etc. A scene is defined in Lightning by adding a class that inherits from the **LightningGL::Scene** class.

**4.1. The Scene Manager**

The Scene Manager is the Lightning asset manager responsible for managing scenes. It provides utilities for sending events to scenes, handles the main loop of the scene Renderer (any scene can also handle events passed to the Renderer), and starting and shutting down scenes and the engine. Using the Scene Manager allows you to not have to write any boilerplate code in order to use the engine’s functionality, and will be required in version 1.2.0 and later.

To use the Scene Manager, it is required to set a startup scene, which is the scene that will be run at the moment your game is started. This is done by setting the value of **GlobalSettings::StartupScene** (see *Chapter 5: Settings (Global & Local)* for more information about Global Settings) to the class name (or Name property if it is set) of your scene. An error will be thrown if no startup scene is set or there are no defined scenes in the game assembly.

It is possible to access the current **Renderer** object from the Scene Manager using the read-only **SceneManager::Renderer** object. You can use this to perform rendering actions if the current **Renderer** object has not been passed to the method you are writing.

**4.2. Creating Scenes**

To create a scene, simply add a class to your game assembly that inherits from the Scene class. No file modification is required; the Scene Manager will automatically initialise every scene that inherits from Scene.

To create a scene, you must implement five methods:

**Start():** Called at the startup of the scene.

**Render(Renderer cRenderer):** Called each frame. The **cRenderer** property is the Renderer created by the Scene Manager and is used for calling asset managers for rendering.

**SwitchTo(Scene oldScene):** The scene is about to be switched to. **oldScene** is the scene that is about to be switched away from.

**SwitchAway(Scene newScene):** The scene is about to be switched away from. **newScene** is the scene that is about to be switched to.

**Shutdown():** Called at engine shutdown.

Optionally, you can also override the **Name** property and set a name for a scene that is used in SceneManager calls. If this is not done, the class name of the scene will be used for its name.

This is an example scene class, reproduced verbatim from the BasicScene example available in the Lightning2.Examples solution (which can be accessed from the Lightning Examples link in the Lightning SDK folder of the Start Menu):

using LightningGL;

using static LightningBase.SDL;

using System.Drawing;

using System.Numerics;

namespace BasicScene

{

public class MainScene : Scene

{

public override void Start()

{

}

public override void Shutdown()

{

}

public override void SwitchTo(Scene oldScene)

{

}

public override void SwitchAway(Scene newScene)

{

}

public override void Render(Renderer cRenderer)

{

PrimitiveRenderer.DrawText(cRenderer, "Hello from MainScene", new Vector2(300, 300), Color.Red);

// change the scene

if (cRenderer.EventWaiting)

{

if (cRenderer.LastEvent.type == SDL\_EventType.SDL\_KEYDOWN) SceneManager.SetCurrentScene("Scene2");

}

}

}

}

**4.3. Scene Management**

Scenes can be managed by calling public methods in **SceneManager**.

To acquire a **Scene** object, call **SceneManager::GetScene** with the name of the Scene you wish to acquire (by default, its class name). It will return the Scene object if a Scene exists with that name, or **null** if it is not present.

To set the current scene, call **SceneManager::GetCurrentScene** with either a Scene object obtained via **GetScene** or the name of the scene to switch to. This method will throw an error if the scene does not exist. If it does, it will call **SwitchAway** on the old scene, **SwitchTo** on the new scene, and then switch to the new scene.

**Example:**

SceneManager.SetCurrentScene(“Scene2”); // if scene2 does not exist will throw error

// not recommended  
Scene scene = SceneManager.GetScene(“Scene2”);

SceneManager

**4.4. Scene Shutdown**

On shutdown, any assets loaded in **Init** MUST be unloaded, or errors may occur on shutdown. This goes for all types of assets that are loaded using the Lightning asset managers and APIs.

**5. Settings (Global & Local)**

Lightning provides two types of Settings APIs, one for global, engine-level settings and one for local settings used for your game.

**5.1. Global Settings Overview  
GlobalSettings** is the class that implements engine settings in Lightning; it is loaded from an INI file, which should always be present in the **Content** folder, called **Engine.ini.** This file is not intended for game developer use, and should **NOT** be used to store game settings. It is only used for settings that affect the entire engine instead of specific games.

It stores various types of settings, such as settings governing package information, graphics settings, if the Scene Manager is to be used or not, and various other aspects of the engine.

It is highly recommended that you recommend to the users of your game that this file not be modified except in ways that are accessible through the game, such as graphics settings, as modifying the settings in an incorrect manner could make the game unbootable.

**5.2. Global Settings Values**

**5.2.1. General Section**

**string GLOBALSETTINGS\_PATH**

Internal static value used to store the path to the Global engine settings. Read-only for users.

**bool ProfilePerformance**

Determines if the **Performance Profiler** is enabled – see *Chapter* *18.3. Performance Profiler* for further information on the Performance Profiler.

**bool EngineAboutScreenOnShiftF9**

Determines if the Engine About screen will be available when **Shift+F9** is pressed. Defaults to **true.**

**bool DeleteUnpackedFilesOnExit**

Ignored if **PackageFile** is not set, or a package file was not passed using the **-packagefile** command line option. Deletes files unpacked using LightningPackager at start on exit. Defaults to **false.**

**string LocalSettingsPath**

Path to the Local Settings file. Defaults to **null,** which will disable the Local Settings API.

**string PackageFile**

A package (**.WAD**) file containing game content (except Engine.ini, which must always be uncompressed and is banned from being included in packages by the **MakePackage** tool) to extract on startup.

**string ContentFolder**

The content folder to extract game content to. Regardless of this setting, **Engine.ini** must always be in the **Content** folder under the game root. Ignored if **PackageFile** is not set.

**bool DontUseSceneManager**

Determines if the Scene Manager will be used or not. If it is not used (this value is set to **true**), you will not be able to use Scenes and will have to initialise the engine manually. See *Chapter 2.2: If you don’t want to use the Scene Manager: Creating a Renderer Manually using RendererSettings* for further information on this topic.

**bool DontSaveLocalSettingsOnShutdown**

Determines if Lightning will save local settings on shutdown (which it will do if the value of this GlobalSetting is set to **false**). Has no effect if the **LocalSettingsPath** GlobalSetting is not set or is set to **null**.

**5.2.2. Localisation section**

**string Language**

The language to use. Must correspond to a valid language INI file in the **Content\Localisation** folder, or the value of the **LocalisationFolder** GlobalSetting if that . See *Chapter 16: Localisation* for information on Lightning’s localisation system.

**string LocalisationFolder**

The folder to load localisation files from. Must

**5.2.3: Graphics section**

**int MaxFPS**

The maximum number of frames per second to render. The engine will use a stopwatch to delay until the frametime required for the framerate if the frame took under the required frametime. Set to 0 or lower to run as fast as possible.

**bool ShowDebugInfo**

Show the current framerate and frame number at the top-left corner of the screen, as well as red warning text shown when you are running below **MaxFPS.**

**SDL\_RendererFlags RendererFlags**

The **SDL\_RendererFlags** of the Renderer. See [https://wiki.libsdl.org/SDL\_RendererFlags](https://wiki.libsdl.org/SDL_WindowFlags) for more information on how to use this setting. The string values of the the enum can be used in Engine.ini, as well as the numeric values. The default is **SDL\_RENDERER\_SHOWN.**

To set the Renderer as fullscreen, use **SDL\_RENDERER\_FULLSCREEN\_DESKTOP**, not **SDL\_RENDERER\_FULLSCREEN**. This is because **SDL\_RENDERER\_FULLSCREEN** forces the operating system to give semi-exclusive control of the Renderer to your application, as well as forces a video mode to a non-native monitor resolution if the **ScreenResolutionX** and **ScreenResolutionY** GlobalSettings are less than the primary monitor’s native resolution, causing severe compatibility issues with ALT+TAB and some overlay programs.

**SDL\_RendererFlags RenderFlags**

The **SDL\_RendererFlags** of the Renderer. See <https://wiki.libsdl.org/SDL_RendererFlags> for more information on how to use this setting. The string values of the the enum can be used in Engine.ini, as well as the numeric values. The default is **0**, with no flags set**.**

**uint ResolutionX**

The initial width of the main game Renderer. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_RENDERER\_FULLSCREEN\_DESKTOP** flag. The default value is 960 pixels. Can be changed after startup by modifying the value of the Renderer’s **RendererSettings.Size.X.**

**uint ResolutionY**

The initial height of the main game Renderer. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_RENDERER\_FULLSCREEN\_DESKTOP** flag. The default value is 640 pixels. Can be changed after startup by modifying the value of the Renderer’s **RendererSettings.Size.Y.**

**uint PositionX**

The initial horizontal position of the main game Renderer on the screen. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_RENDERER\_FULLSCREEN** or **SDL\_RENDERER\_FULLSCREEN\_DESKTOP** flag. The default valued is the value of the mathematical expression **(SystemInfo.Resolution.X / 2) – (GlobalSettings.Resolution.X).** Can be changed after startup by modifying the value of the Renderer’s **RendererSettings.Position.X.**

**uint PositionY**

The initial vertical position of the main game Renderer on the screen. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_RENDERER\_FULLSCREEN** or **SDL\_RENDERER\_FULLSCREEN\_DESKTOP** flag. The default valued is the value of the mathematical expression **(SystemInfo.Resolution.Y / 2) – (GlobalSettings.Resolution.Y).** Can be changed after startup by modifying the value of the Renderer’s **RendererSettings.Position.Y.**

**string RendererTitle**

Optional. The initial title (also known as caption text) of the main Renderer. Can be changed after startup by modifying the value of the Renderer’s **RendererSettings.Title.**

**Renderer Renderer**

The rendering backend to use. The valid values for the **Renderer** enum are **direct3d** (either Direct3D 9 or Direct3D 11 – SDL uses the best that is supported)**, direct3d11, opengl, opengles, opengles2, metal,** or **software. Metal** is only supported on macOS. **Opengl** is currently only supported on Linux. If SDL fails to use its preferred renderer, it will pick the best supported one and Lightning will issue a warning to this effect.

**WARNING: Using the software renderer is not recommended as it is extremely slow!**

**5.2.4: Requirements section**

This system allows the enforcement of system requirements for your game. The engine will refuse to boot the game if the system requirements are not met.

**int MinimumSystemRam**

The minimum about of system RAM, in MiB.

**int MinimumLogicalProcessors**

The minimum number of logical processors. This value is **NOT** the number of CPU cores, but instead the number of logical processors, which, if the CPU supports hyperthreading, is usually (but not always) two times the number of CPU cores. It is the value marked as **“threads”** in CPU specifications, although that is not strictly accurate.

**SystemInfoCPUCapabilities MinimumCpuCapabilities**

The minimum CPU capabilities required for the game. See *Chapter 17.1.2: The SystemInfoCPUCapabilities Enum* for more information on this topic and the values that can be set.

**SystemInfoOperatingSystem MinimumOperatingSystem**

Enforces a minimum operating system for your game. The lowest version of Renderers supported by .NET 6.0 is Renderers 7, and macOS supported by .NET 6.0 is 10.13. The requirements check will only happen for the current operating system. The values for this setting are provided in the **LightningGL::SystemInfoOperatingSystem** enum.

This is not supported on Linux. For further information on the values of this enum, refer to *Chapter 17.1.3: The SystemInfoOperatingSystem Enum*.

**5.2.5: Scene section**

**string StartupScene**The name of the startup scene. Required to use the Scene Manager, and must correspond to the name of a valid scene in your main game assembly. By default, the name of a scene is the name of its class. All scenes must inherit from **LightningGL::Scene**, and will not be recognised by the Scene Manager if they do not.

**5.3: LocalSettings (and an introduction to INI files…)**

Lightning provides a local settings API through the **LocalSettings** class. Local settings are automatically loaded at engine start if the value of **GlobalSettings::LocalSettingsPath** is not **null.** If the file specified by the **LocalSettingsPath** GlobalSetting does not exist and it corresponds to a valid path in a folder extant on the system, Lightning will create it automatically during startup; Lightning will automatically save it at shutdown unless the value of the **DontSaveLocalSettingsOnShutdown** GlobalSetting is set to **true** (for more information on GlobalSettings, refer to *Chapter 5.2: Global Settings*).

Before working with the LocalSettings INI file, it is important to understand the basic concepts of an INI file. If you do understand the basic concepts of an INI file, this section will be extremely condescending for you and is best skipped.

For the purposes of this section:

* A **section** is a section of an INI file. A section is marked with the **[** character and ended with a **]** character.
* A **key** is an identifier for a property or variable.
* A **value** is the value of a **key**.

An INI file like this:

[Section1]

a=b

defines one section, **Section1**, containing a single key **a** with the value **b**.

To add a section to the settings, call **LocalSettings::AddSection** with the name of the section you wish to add, like so:

LocalSettings.AddSection(“Section”);

They can be removed again with a simple call to **LocalSettings::DeleteSection**:

LocalSettings.DeleteSection(“Section”);

To add a value to the settings, call the **LocalSettings::AddValue** method with the name of the section and the key and value of the value you wish to add, like so:

LocalSettings.AddValue(“Section”, “Key”, “Value”);

Added values can be modified with the **LocalSettings::SetValue** method, as long as the parameters correspond to a real value in a real section:

LocalSettings.SetValue(“Section”, “Key”, “Value2);

Finally, values can be defeated with a call to the **LocalSettings::DeleteKey** method:#

LocalSettings.DeleteKey(“Section, “Key”);

The settings can be saved via a call **LocalSettings::Save**, which will write out the content of **LocalSettings::LocalSettingsFile** to the value specified in **GlobalSettings::LocalSettingsPath**. Lightning will do this at shutdown for you unless the value the value of the **DontSaveLocalSettingsOnShutdown** GlobalSetting is set to **true**.

**6. Cameras**

**6.1: Camera Overview**

Lightning provides a basic camera system to allow for scrolling and similar game actions. A camera is created by creating an instance of the **Camera** class with a **CameraType**, like so:

Camera camera = new Camera(CameraType.Chase)

Currently there are only three camera types:

**Follow**No processing is done to the camera location.

**Chase**The camera is offset to the left, and slightly up, as if it was chasing its subject.

**Floor**The camera is offset to the bottom as if there was a rising floor.

Alternatively, you can change the **FocusDelta** property of the camera to change how the camera behaves. The default for the Follow camera type is **(0, 0),** whereas the default for the Chase camera type is **-RendererSettings::Size::X / 2**.

When cameras are in use (signified by the **Settings::Camera** value of the current Renderer being set to a Camera object, all rendering commands will have their positions transformed such that only objects in the camera’s render area are shown.

To set a camera as the current camera, call **Renderer::SetCurrentCamera** with a valid **Camera** object:

Camera camera = new Camera(CameraType.Chase);  
Renderer.SetCurrentCamera(camera);

**6.2: Camera Shake**

Cameras also have the ability to shake. This mechanism is controlled by the **Camera::ShakeAmount** property, which is a **Vector2** holding the maximum displacement of the camera from its Position. Correction is applied to prevent the camera drifting over time – this behaviour can be turned off by setting the value of the **Camera::AllowCameraMoveOnShake** Boolean property to **true**. A value of **(0,0)** indicates no shake.

**6.3: Camera Velocity**

Cameras can be given Velocity in Lightning 1.1 and later. This mechanism is controlled by the **Camera::Velocity** property. Setting the value of this property to **(0,0)** will not move the camera, otherwise the camera will move each frame by the value of this property.

**7. Input, Event Handling & Collision**

**7.1. Input and Event Handling overview**

Lightning currently primarily supports two forms of input: keyboard and mouse. SDL can be used to add support for numerous further types of input, such as touch, joysticks and controllers of various types, and native Lightning support for these will arrive in a future release. Event handling is handled using the SDL event system with some additional classes to make the handling more programmer- and C#- friendly. UI event handling is different and will be covered in *Chapter 12: UI.*

For all supported SDL events, see the documentation for **SDL\_EventType** at the SDL wiki:  
<https://wiki.libsdl.org/SDL_EventType>.

To handle events, the **Renderer::EventWaiting** should be checked:

if (Renderer.EventWaiting)

{

// event handling code…

}

The event information is contained in the **Renderer::LastEvent** property. An example event handler would be something like this:

if (Renderer.EventWaiting)

{

switch (Renderer.LastEvent.type)  
 {

case SDL\_EventType.SDL\_KEYDOWN:

// key handling code here…

}

}

To use this API, you must add using LightningBase; to the top of your code.

Lightning provides utility classes for easier usage of mouse and keyboard input. They are **Key** for keyboard input and **MouseButton** for mouse input respectively.

An instance of Key can be obtained by casting an **SDL\_KeyboardEvent** to it, whereas an instance of MouseButtoncan be obtained by casting an **SDL\_MouseButtonEvent** or **SDL\_MouseMotionEvent** to it.

**7.2. The Key Class**

**Properties:**

**SDL\_Keysym KeySym**The **SDL\_KeySym** of the key. The Lightning game engine uses KeySym sinstead of keycodes in order to provide an API that is more accurate to real keyboards and supports keyboards that have non-standard buttons. Processing is done to handle punctuation and various other characters. This design may be changed in future releases.

It is **not recommended** to use this to determine what key has been pressed as in some cases multiple keysyms are used for different keys. Instead, it is recommended to use the case-insensitive **Key::ToString** method, which converts all keys to uppercase and handles duplicate keys.   
However if you wish to use this method for determining which key has been pressed, the documentation for the SDL\_Keysym enum is here on the SDL Wiki: <https://wiki.libsdl.org/SDL_Keysym>

**SDL\_Keymod Modifiers**

The key modifier state. See the documentation for SDL\_Keymod at <https://wiki.libsdl.org/SDL_Keymod> for further information.

**WARNING: KMOD\_CTRL, KMOD\_SHIFT, KMOD\_ALT, and KMOD\_GUI do not refer to either one of their respective modifier keys being pressed, but to *both.* If you want to check for *either* modifier key being pressed, use the Key class utility methods EitherShiftPressed, EitherAltPressed, EitherCtrlPressed, and EitherGuiPressed.**

**bool Repeated**

Determines if the key is repeated or not.

**WARNING: Do not use this for continual movement in games. Instead, store if the key is being pressed and move based on that. There will be unbearable input delay (~0.2 seconds or more) if you simply check the repeat property.**

**Methods:**

**string ToString()**Returns a textual version of the Key, which is always converted to culture-invariant uppercase. This can be used, for example, with camera movement:  
 switch (curEvent.type)

{

case SDL.SDL\_EventType.SDL\_KEYDOWN: // Key is held down.

Key key = (Key)curEvent.key;

string keyString = key.ToString();

switch (keyString)

{

case "LEFT":

case "A":

camera.Position -= new Vector2(10, 0);

break;

case "RIGHT":

case "D":

camera.Position += new Vector2(10, 0);

break;

case "UP":

case "W":

camera.Position -= new Vector2(0, 10);

break;

case "DOWN":

case "S":

camera.Position += new Vector2(0, 10);

break;

}

break;

}

**bool EitherShiftPressed()**Returns true if either SHIFT key is held.

**bool EitherCtrlPressed()**Returns true if either CTRL key is held.

**bool EitherAltPressed()**Returns true if either ALT key is held.

**bool EitherGuiPressed()**Returns true if either GUI key (Renderers key, Apple key, etc) is held.

**7.3. The MouseButton class**

A class defining a mouse action.

**Properties**

**SDL\_MouseButton Button**The **SDL\_MouseButton** of the current mouse action. This is not a part of the SDL API and is a construct added by LightningBase. The valid buttons are Left, Right, Middle, X1 (forward button), and X2 (mouse button), and any combination of these.

**Vector2 Position**The position of the mouse action.

**Vector2 Velocity**The velocity of the mouse. This property is only filled when the MouseButton instance was obtained by a cast from **SDL\_MouseButtonEvent**.

**int ClickCount**The number of times in a row a mouse has been clicked.

**7.4. Hit Testing**

While Lightning does not implement a physics engine at the current time, a class (**AABB**) for hit testing is implemented, which allows you to test two axis aligned bounding boxes using the **Intersects** method. This method is demonstrated by the **Collision** example. There are several overloads for this method:

**bool Intersects(Renderable render1, Renderable render2):** Tests intersection between two Renderable objects.

**bool Intersects(Renderable render1, Vector2 render2):** Tests intersection between a Renderable and a position on the screen. The position is always located in world-relative space.

**8. Primitives**

**8.1. Primitive Drawing Methods Overview**

Lightning provides basic primitive drawing methods to draw simple shapes extremely quickly, quicker than SDL can provide. These use the **SDL2\_gfx** library with some custom additions and modifications at the C++ level, in order to correct limitations in the original library. There is also a simple text drawing method that does not require any fonts to be loaded and uses the debug font provided by SDL2\_gfx.

This is used for all debugging functionality in Lightning, and it is recommended that all debug functionality uses this font to increase debugging performance.

All of these methods are provided by the static **PrimitiveRenderer** class.

**8.1.1. Pixels**

public static void DrawPixel(Renderer cRenderer, Vector2 position, Color color, bool snapToScreen = false)

Draws a pixel to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw a pixel to  
**position:** The position to draw the pixel at.  
**color:** The color of the pixel to draw, in ARGB format (refer to *Chapter 1.3: Standard Types Used* for documentation of System.Drawing.Color)  
**snapToScreen:** Determines if the pixel will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.2. Lines**

public static void DrawLine(Renderer cRenderer, Vector2 start, Vector2 end, short thickness, Color color, bool antiAliased = true, bool snapToScreen = false)

Draws a line to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw the line to.  
**start:** The start position of the line.  
**end:** The end position of the line.  
**thickness:** The thickness of the line In pixels.  
**color:** The color of the line.  
**antiAliased:** Determines if the line will be anti-aliased. The default value is **true**.  
**snapToScreen:** Determines if the pixel will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.3. Circles**

public static void DrawCircle(Renderer cRenderer, Vector2 position, Vector2 size, Color color, bool filled = false, bool antiAliased = false, bool snapToScreen = false)

Draws a circle to the screen.

**Parameters:**  
  
**cRenderer:** The Renderer to draw the circle to.

**position:** The position of the **centre** of the circle.

**size:** The horizontal and vertical radius of the circle in pixels.  
**color:** The color of the circle.

**filled:** Determines if the circle is filled or not. The default value is **false**. **antiAliased:** Determines if the circle is anti-aliased. **Warning: Only works for unfilled circles in this release!**

**snapToScreen:** Determines if the pixel will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.  
  
**8.1.4. Rectangles**

public static void DrawRectangle(Renderer cRenderer, Vector2 position, Vector2 size, Color color, bool filled = false, Color borderColor = default(Color), Vector2 borderSize = default(Vector2), bool snapToScreen = false)

Draws a rectangle to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw the circle to.

**position:** The position of the rectangle.

**size:** The size of the rectangle.  
**color:** The color of the rectangle.   
**filled:** Determines if the rectangle will be filled or not. The default value is **false.**  
**borderColor:** The color of the border. The default value is **0,0,0,0** (**transparent**)  
**borderSize:** The size of the border in pixels. The default value is **0**.

**snapToScreen:** Determines if the rectangle will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.5: Rounded Rectangles**public static void DrawRoundedRectangle(Renderer cRenderer, Vector2 position, Vector2 size, Color color, int cornerRadius, bool filled = false, bool snapToScreen = false)

Draws a rounded rectangle to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw the rounded rectangle to.  
**position:** The position of the rounded rectangle.

**size:** The size of the rounded rectangle.  
**color:** The color of the rounded rectangle.  
**cornerRadius:** The radius of the corners of the rounded rectangle in pixels.  
**filled:** Determines if the rounded rectangle is filled or not. The default value is **false**.

**snapToScreen:** Determines if the rounded rectangle will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.6: Triangles**public static void DrawTriangle(Renderer cRenderer, Vector2 point1, Vector2 point2, Vector2 point3, Color color, bool filled = false, bool snapToScreen = false)

Draws a triangle to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw the triangle to.

**point1:** The position of the first point of the triangle.  
**point2:** The position of the second point of the triangle.

**point3:** The position of the third point of the triangle.

**color:** The color of the triangle.  
**filled:** Determines if the triangle is filled or not. The default value is **false**.  
**snapToScreen:** Determines if the triangle will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.7: Polygons**public static void DrawPolygon(Renderer cRenderer, List<Vector2> points, Color color, bool filled = false, bool antiAliased = false, bool snapToScreen = false)

Draws a polygon to the screen.

**Parameters:**

**cRenderer:** The Renderer to draw the polygon to.

**points:** A List of Vector2s containing the points of the polygon.|  
**color:** The color of the polygon.  
**filled:** Determines if the polygon is filled. The default value is **false**.  
**antiAliased:** Determines if the polygon is anti-aliased before drawing. The default value is **false**. **WARNING: This parameter is only supported for unfilled polygons in this release. Using filled as true will simply draw a non-antialiased polygon. This will be corrected in a future release.**

**snapToScreen:** Determines if the polygon will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**8.1.8: Text using the Debug Font**While Lightning provides a full-blown Font Manager leveraging SDL2\_ttf and FreeType-2 in order to allow many forms of text to be drawn, there is also a simpler – and faster – rendering API using a debug font provided by SDL2\_gfx. To use it, call this method in **PrimitiveRenderer:**public static void DrawText(Renderer cRenderer, string text, Vector2 position, Color color, bool localise = true, bool snapToScreen = false)

**Parameters:**

**cRenderer:** The Renderer to draw the text to.

**text:** The text to draw to the screen. **Warning: Multi-line text is not supported using this method! For multi-line text, you must use Font Manager (see *Chapter 14: Text Rendering and Font Management* for further information on this topic)**

**position:** The position to draw the text on the screen.  
**color:** The Color of the text. **Warning: Outline color, gradients, or any other kind of text styling is not supported. For this support, you must use Font Manager (see *Chapter 14: Text Rendering and Font Management* for further information on this topic)**

**localise:** Determines if the text will be localised or not. The default value is **true**. For further information on Lightning’s internationalisation support, see *Chapter 16: Localisation.*

**snapToScreen:** Determines if the polygon will be drawn in world-relative space (**false**) or screen-relative space (**true**). The default value is **false**.

**Example:** PrimitiveRenderer.DrawText(Renderer, "Basic example", new Vector2(100, 100), Color.White);

will produce



**9. Textures (and Atlases)**

Having a renderer is good enough, but you require things to draw to have any kind of game! To facilitate this, Lightning supports textures. The API is in an early state and is planned to be dramatically improved in future releases, but it is enough to perform game rendering tasks in its current state.

**9.1. Texture Loading & Drawing**

The simplest task that can be done with a texture is loading an image to it and then drawing that image.   
To do this, simply instantiate a new **Texture** with the current Renderer and the size of the image you are attempting to load (or the texture you wish to create):

Texture texture = new Texture(cRenderer, 512, 512);

Then load it by setting its **Path** property and calling **Texture::Load** on it with the only parameter being the current Renderer:

texture.Path = “Content/image.png”;

texture.Load(cRenderer);

**or** if you are using the scene manager:  
texture.Load(SceneManager.Renderer);

The texture can be positioned with its **Position** property, which is a Vector2:

texture.Position = new Vector2(100, 100)

The texture can then be drawn with:

texture.Draw(cRenderer);

**9.2. Texture API**

There is a texture API to allow the manipulation of textures. You do not have to load a texture to draw it – you can merely create it and then use the **Texture API** in order to manipulate textures. The Texture class inherits from **Renderable** (refer to *Chapter 1.4: The Renderable class* for further information on Renderables*)*

Textures must be locked to modify them and unlocked to facilitate drawing. This is done by calling the **Lock** and **Unlock** methods of the texture, but the pixel plotting functions will automatically lock the texture. It is only required to call **Unlock** once, right before **Draw**.

**Properties**

**string Path:** The path of the texture. If this is null (you are not loading an image), the value will be **<<<CREATED TEXTURE>>>.**

**IntPtr Handle:** Pointer to the SDL\_Texture\* unmanaged structure in memory.

**int\* Pixels:** Pointer to the texture’s pixels. **Warning: Only valid when the texture is locked for editing! Otherwise the value will be NULL and trying to dereference it will cause an access violation!**

**Vector2 Position:** The position of the **top-left** of this texture.

**Vector2 Size:** The size of this texture. If you are using this texture to draw an image, it is recommended to set this to the size of the image you are loading and use the **ViewportStart** and **ViewportEnd** properties in order to control what areas of the image are drawn.

**Vector2 Repeat:** If the value for both the X and Y components of the vector is more than one, determines the number of times the texture will be drawn. The **X** component of the vector determines the number of times the texture will be drawn in the positive X direction, whereas the Y component of the vector determines the number of times the texture will be drawn in the positive Y direction.

**Vector2 ViewportStart:** If the value for both the X and Y components of the vector is at leat zero and less than the size of the texture, this determines the point within the texture drawing of the texture will be started.

**Vector2 ViewportEnd:** If the value for both the X and Y components of the vector is at leat **ViewportStart** and less than the size of the texture, this determines the point within the texture drawing of the texture will be ended.

**bool Locked:** Determines if the texture is locked and therefore cannot be edited. You usually do not need to check this, as the plotting functions will automatically lock and unlock the texture in an optimal way.

**int Pitch:** The pitch of the texture. The pitch is the distance in bytes between two lines of the bitmap, and is usually (as 32-bit color is used in Lightning) four times the size of the image’s width.

**Methods**

**Texture(Renderer cRenderer, float sizeX, float sizeY, SDL\_TextureAccess textureAccess = SDL\_TextureAccess.SDL\_TEXTUREACCESS\_STREAMING):** Creates a new texture for the Renderer **cRenderer**, and then sets the **Size** to (**sizeX, sizeY**­). If the **textureAccess** parameter is provided, the

**Load(Renderer cRenderer):** Loads the texture. The texture must have a **Path** property corresponding to a valid image file in PNG, JPG, TIF, or WebP format, or an error will occur.. Throws an error if an invalid format is loaded or an error occurs during loading.

**Color GetPixel(int x, int y, bool unlockNow = false)** Returns a Color object containing the color of the pixel at **x,y.** An error will occur if either x or y are less than 0 or are more than the size of the image, minus one. **unlockNow** will unlock the texture for drawing after the pixel has been acquired, but it is not recommended to unlock when drawing every pixel for speed reasons. It is recommended to only use the unlockNowparameter if you only need to draw a single pixel.

**SetPixel(int x, int y, Color color, bool unlockNow = false)** Sets the pixel **x,y** of the texture to the Color **color.** An error will occur if either x or y are less than 0 or are more than the size of the image, minus one. **unlockNow** will unlock the texture for drawing after the pixel has been set, but it is not recommended to unlock when drawing every pixel for speed reasons. It is recommended to only use the unlockNowparameter if you only need to draw a single pixel.

**Lock():** Locks the texture so that its pixels can be manipulated. Results in the Pixels pointer pointing to valid memory, as well as the Pitch property being set to the pitch of the image (The concept of image pitch is defined in the description of the Pitchproperty). Automatically called by GetPixel and SetPixel if the texture is unlocked.

**Unlock():** Unlocks the texture so that it can be drawn. Results in the Pixelspointer being set to **NULL** and the Pitch property being set to zero. (The concept of image pitch is defined in the description of the Pitchproperty). Automatically called by GetPixel and SetPixel if the texture is locked and the **unlockNow** parameter is set to true.

**Draw(Renderer cRenderer):** Draws the texture to the Renderer **cRenderer.** Uses the properties of the texture that you have set.

**SetBlendMode(SDL\_BlendMode blendMode):** Sets the blend mode of the texture.

For further information on blend modes in SDL, refer to the SDL wiki documentation for **SDL\_BlendMode**:  
<https://wiki.libsdl.org/SDL_BlendMode>

And for information on composing custom blend modes, refer to the SDL wiki documentation for **SDL\_ComposeCustomBlendMode:**<https://wiki.libsdl.org/SDL_ComposeCustomBlendMode>

**Unload()**: Unloads the texture. Sets Handle and Pixels to **NULL** (or **IntPtr.Zero**), and sets the pitch to zero. **WARNING: The texture cannot be loaded or drawn after a call to Unload in this version!**

**9.3. Texture Atlases**Lightning also supports texture atlases. The API for texture atlases is based on the API for textures (to the point where **TextureAtlas** inherits from **Texture**). However, there are modifications to facilitate texture atlases.

Any properties or methods not discussed are identical in their usage to non-Atlas textures.

**Properties:**

**TextureAtlas(Renderer cRenderer, Vector2 frameSize, Vector2 textureCount):** Creates a texture atlas for the Renderer **cRenderer**. The size of an individual texture within the texture atlas is the **frameSize** property, whereas the **textureCount** property is the number of textures within the texture sheet.

**Example:**

TextureAtlas atlas = new TextureAtlas(cRenderer, new Vector2(64, 64), new Vector2(10, 10)

This would create a texture of size **(640, 640)**, as the size of an individual texture frame is multiplied by the number of textures in each axis to create the final image size.

**Vector2 FrameSize**: The size of one individual image within the texture atlas. **Can only be set at initialisation!**

**Vector2 TextureCount:** The number of textures in the image. Generally the image size divided by the size of one texture.

**Int Index:** The current index of the pixels used for drawing. The minimum index is 0 and the maximum index is **TextureCount.X \* TextureCount.Y**, minus one.

**GetPixel** and **SetPixel** have a **relative** parameter in addition to the parameters described in *Chapter 9.2: Texture API*. If this parameter is set to **true**, the **x** and **y** parameters will be interpreted as coordinates relative to the set index of the texture. If it is not, the coordinates will be interpreted as coordinates relative to the texture atlas.

**Draw** will draw the current **Index** property. To draw multiple indexes, call the Draw method again with a different index.

All other facets of operation are identical to normal textures.

**10. Animation**

**10.1. Animated Texture Overview**

Lightning implements a basic animation system to allow fluid movement of objects. This is primarily implemented through the **AnimatedTexture** class.

While **AnimatedTexture**‘s API is generally similar to that of Texture, and anything not mentioned here is the same as Texture (see *Chapter 9.2: Texture API* for further information on the Texture API), there are some differences in using this class when compared to the **Texture** and **AnimatedTexture** classes:

* The **Repeat** property is instead called **TextureRepeat** here. The **Repeat** property is instead used for determining how many times the animation repeats. To set an animation that always plays, set the value of Repeat to **zero**.
* Instead of a **Path** property, there is a **FramesPath** property containing a list of all of the frames that are to be loaded. ALL frames used in the animation **MUST** be added to this property using the **AnimatedTexture::AddFrame** method before loading.
* There is at this time no **ViewportStart** and **ViewportEnd** properties for AnimatedTextures, nor can **GetPixel** and **SetPixel** be used. This will be corrected in the next release and may be frame-specific.
* To set the blend mode of an AnimatedTexture, it is required to set the blend mode for every single frame of the animation. This can be done with a for loop fairly easily This will be corrected in the next-release.

To create an animated texture, the first step is to create its object:

AnimatedTexture animatedTexture = new AnimatedTexture(cRenderer, 512, 512);

It is **highly recommended** that all frames of the animation are the same size provided to the constructor of the **AnimatedTexture**, in order to prevent distortion of the texture.

Each frame of the image must be a different file on disk. To add a frame, call **AnimatedTexture::AddFrame** with the path of every frame of the animation that you wish to load:

animatedTexture1.AddFrame(@"Content\AnimTextureTest\AnimTextureTestF0.png");

animatedTexture1.AddFrame(@"Content\AnimTextureTest\AnimTextureTestF1.png");

animatedTexture1.AddFrame(@"Content\AnimTextureTest\AnimTextureTestF2.png");

animatedTexture1.AddFrame(@"Content\AnimTextureTest\AnimTextureTestF3.png");

Then, configure any properties of the AniamtedTextureas normal if you wish.

It is then required to set the AnimatedTexture’s **Cycle** property, an instance of the class **AnimationCycle** that contains information on how the animation will be cycled.

**10.2. Animation Cycles**

**Properties**

**Int StartFrame:** The start frame of the animation. The range of valid values is from zero to the number of frames in the animation, minus one.

**Int EndFrame:** The end frame of the animation. This **does not** have to be more than the StartFrame property. If it is less than the StartFrame property, the animation will simply play in reverse.

**uint FrameLength:** The number of frames one frame of the animation will play for before changing to the next frame.

**Methods**

**Constructor(int nStartFrame, int nEndFrame, uint nFrameLength):** Constructor for AnimationCycle. The **StartFrame** property is set to the value of the **nStartFrame** parameter, the **EndFrame** property is set to the value of the **nEndFrame** parameter, and the **FrameLength** property is set to the value of the **nFrameLength** parameter.

**12. Lighting**

**12.1. Lighting Overview**

Lightning uses a basic, screen-space environmental map system for rendering lights. To use them, Lightning provides an asset manager specifically for use with lighting – the Light Manager.

All lights are masks drawn on top of a texture filling the entire screen, set to the environmental light color. All lights are drawn on top of this when they are created, instead of once per frame, in order to improve the performance of the engine. Removing a light from the list of lights will stop drawing its mask.

Each light uses the midpoint circle algorithm modified to take into account the range and brightness of the light. The range of the light is the radius of the circle, and the brightness sets the minimum alpha that will be used to draw the light to (255 – brightness).

The first step in using the Light Manager is to set the environmental light color, as it is set to be fully transparent (ARGB **0,0,0,0**) at engine startup. To do this, call **LightManager::SetEnvironmentalLight**:

LightManager.SetEnvironmentalLight(Color.FromArgb(100, 50, 50, 50);

This line of code will set the environmental light to a dull red color.

It is then necessary to add lights. For further information and reference on the usage of the Light class, refer to *Chapter 11.2: The Light Class*. To add lights, create a newinstance of the Light class:

// See Ch11.2 for details on Light properties  
Light light = new Light();

light.Range = 8;

light.Brightness = 123;

light.Position = new Vector2(100, 100);

Note that all light positions are interpreted as screen-relative space instead of world-relative space.

Then, simply add the Light to the Light Manager using the **LightManager::AddLight** method:

LightManager.AddLight(light);

The light will then be drawn on each frame by the Light Manager.

There is no requirement to manually draw lights. The Light Manager will automatically draw all lights that have been added to its light list during the call to **Renderer::Render** in the engine’s main loop.

**12.2. The Light class**

**Properties**

**int Brightness:** The brightness of this texture. Used to set the lowest alpha of the screenspace map texture when drawing the light, which is set to 255, subtracted by the value of this property. The range is **0** to **255** (any value above 255 will be set to 255), and the default value is 255.

**int Range:** The range of the texture. Determines how far the light emanates from its position.

**Vector2 Position:** The position of the light.

**12.3. Environmental Light & The Light Manager**

**Properties**

**EnvironmentalLight:** Can be used to get the current color of the light manager’s environmental light color. The environmental light color is the base color of the screen-space lightmap before any lights are drawn to it. If the opacity of the environmental light is less than 255, the world will be visible through the lightmap.

**Methods**

**AddLight(Renderer Renderer, Light light):** Adds the Light object **light** to the Light Manager for rendering on the Renderer **Renderer**. (for more information on Light objects, see *Chapter 11.2: The Light class*).

**SetEnvironmentalLight(Color color):** Sets the environmental light color to the **System.Drawing.Color** object **color.** See the **EnvironmentalLight** property description for further information on environmental light.

**SetEnvironmentalLightBlendMode(SDL\_BlendMode blendMode):** Sets the blend mode for the screenspace lightmap. For more information on texture blending, see the description of the **Texture::SetBlendMode** method in *Chapter 9.2: The Texture API*.

**13. Particle Effects**

Lightning provides a particle effect system in order to advance the graphical capabilities of games. The particle effect system is quite flexible and supports numerous settings to control the emittance of particles and how they are rendered.

Individual particles are generated by *particle effects*, which are loaded and managed by the asset manager for particles, the *particle manager*. In Lightning, these are implemented by the **ParticleEffect** and **ParticleManager** classes respectively.

**13.1. Creating & Loading Particles**

To create a particle effect, an instance of **ParticleEffect** must be created and its various properties used to alter the particle effect’s behaviour and appearance.

All particle effects must be coupled with an image that they display when they are drawn, and then loaded using the **ParticleManager::AddEffect** method.

**13.1.1. The ParticleManager Class**

**Methods**

**AddEffect(Renderer cRenderer, ParticleEffect particle):** Adds a particle effect to the particle manager for rendering to Renderer **cRenderer**. A particle effect must have a **Path** parameter corresponding to a valid on-disk image file when **ParticleManager::AddEffect** is called, as this method automatically loads the particle effect. If the **NeedsManualTrigger** property of the **ParticleEffect** is set to **false**, the particle effect will automatically start playing. Otherwise, it will start playing when the **ParticleEffect::Play** method is called and stop playing when the **ParticleEffect::Stop** method is called.

**13.2. Particle Effect Properties**

Particle effects have numerous parameters that can be modified using properties of an instance of the **ParticleEffect** class. They allow you to modify the behaviour of the particle effect, such as how it plays (if it is or is not repeated, for example), how often particles are created, how many particles are created each time, the maximum number of articles, and many other aspects of particle behaviour. Combined, these allow for an extremely powerful particle system that truly enriches the look of your game.

**Properties**

**int Amount:** The maximum number of particles that can be on screen at any one time. The default value is **0.**

**int Lifetime:** The lifetime of each particle in frames. The particle will despawn after this time. The default value is **0.**

**double Variance:** The variance of each particle’s position in frames. The default value is **0**.

**Vector2 Velocity:** The velocity of each particle. The behaviour of this property is different dependent on the value of **Mode** property. The default value is **(0, 0)**.For more information on the **Mode** property, see *Chapter 12.3: Particle Effect Modes*.

**int MaxNumberCreatedEachFrame:** The maximum number of particles created each frame. The default value is **0.**

**ParticleMode Mode:** The mode of this particle. For more information on thisproperty and texture modes, see *Chapter 12.3: Particle Effect Modes*. The default value is **AbsoluteVelocity**.

**int FrameSkipBetweenCreatingParticles:** The number of frames to wait between creating a set of particles (the number of which will be created is governed by the **MaxNumberCreatedEachFrame** property). The default value is **0**, which will create a particle every frame. If the value is not zero, it will create a particle every (**value + 1**) frames.   
  
**bool NeedsManualTrigger:** Determines if the particle effect needs to be played to start generating particles or not. The default value of this property is **false**. If it is set to **true**, the particle effect will need to be controlled with the **Play** and **Stop** methods.

**13.3. Particle Effect Modes**

In addition to the aforementioned properties, ParticleEffects also have several modes that modify the behaviour of these parameters and how the particle effect is drawn to the screen. They are modified by using the **Mode** property of any **ParticleEffect** instance before adding it to the Particle Manager.

**Modes**

**AbsoluteVelocity:** Absolutely nothing is done to the particles’ velocity. The particles always move at the same velocity as dictated by their **Velocity** property, with their position offset from the centre of the ParticleEffect (dictated by the **Position** property) by their **Variance** property. They will only ever move in their assigned directions at their assigned speeds.

**SinCos:** The particles will move, from the centre of the ParticleEffect (dictated by the **Position** property), by the sine (X) and cosine (Y) of their velocities, multiplied by their lifetime in frames. The velocities are capped to the **Velocity** property. Thus, it is appropriate to say that the Velocity property is partially ignored in this mod.

**Explode:** The same as **SinCos,** but velocities are not capped. Therefore, the particles are seen to explode.

**13.4. Playing Particle Effects**

If the particle effect’s **NeedsManualTrigger** property is set to **false**, the particle effect will be automatically managed by the **ParticleManager**. However, if it is set to **true**, the playing and stopping of the particle effect needs to be handled by you, the developer. There are simple methods in Lightning to facilitate this functionality:

**Methods**

**Play():** Allows the creation of new particles. The particle effect will gradually return to normal speed.

**Stop():** Stops the creation of new particles. The particle effect will gradually “fizzle out” over time; all properties are maintained and the particle effect can be immediately played again, even before all particles have been deleted.

A method to erase all particles immediately will be implemented in the next release.

**14. UI**

**14.1. UI System Overview**

All games need a user interface. Therefore, Lightning provides a user interface subsystem to provide for efficient and rapid implementation of this functionality. The system aims to be flexible and powerful, but also easy to use for developers to quickly produce powerful games.

The UI system in Lightning is made up of two primary primitives: **events** and **gadgets**. A gadget is an individual UI control, such as a checkbox or listbox, that is placed on the screen. All gadgets implement **events** – methods triggered when a particular action is performed. The available events are

An event name is the property of **Gadget** through which its delegate can be accessed for subscribing. The event class name is the delegate within Lightning that holds the event data. All events are handled by the UIManager and automatically routed to all gadgets that have been added to it.

To add a Gadget to the UI Manager for rendering, use the **UIManager::AddGadget(Gadget gadget)** method and pass an instance of any object inheriting from Gadget to it. This will immediately render the object in the next frame.

A gadget can be removed using the **UIManager::RemoveGadget(Gadget gadget)** method. If it has not already been added using the AddGadget method, an error will be thrown (error 135).

**14.2. UI Events**

To implement an event, its **event handler** must be implemented using a method with the same parameters as the event. To do this, simply subscribe to its respective property in any class inheriting fROM **Gadget** (in this case a **TextBox**) with the **+=** keyword. It is recommended to prefix any method names with the name of your gadget variable:

TextBox textBox1 = new textBox();  
textBox1.OnKeyPressed += Tb1KeyPressed;

The OnKeyPressed event requires a single parameter, of type **Key**. So a new method needs to be created – all event methods need to return **void,** so the resulting code (with some text drawing and string interpolation to display the most recently pressed key):

public void Tb1KeyPressed(Key key)  
{  
 PrimitiveRenderer.DrawText(Renderer, $“Key pressed: {key}”, new Vector2(100, 100), Color.Yellow);   
}

This code will then be run every time a key is pressed.

A Gadget is **focused** when it has been clicked on by the user. This is controlled by the **Focused** property and the state determines if some events will be handled.

**14.2.1. OnKeyPressed**

A UI event fired when a key is first held down. **This event is only fired to a Gadget if it is currently focused!**

**Class Name: KeyPressedEvent**

**Parameters:  
Key key –** The key that has been pressed.

For further information on the Key class and its properties, refer to *Chapter 7.2: The Key Class*.

**14.2.2. OnKeyReleased**

A UI event fired when a key is released. **This event is only fired to a Gadget if it is currently focused!**

**Class Name: KeyReleasedEvent**

**Parameters:  
Key key –** The key that has been released.

For further information on the Key class and its properties, refer to *Chapter 7.2: The Key Class*.

**14.2.3. OnMousePressed**

A UI event fired when a mouse button is pressed down. **This event is only fired if the mouse click was within the bounds of the gadget!**

**Class Name: MousePressedEvent**

**Parameters:  
MouseButton button –** The mouse button that has been pressed down.

For further information on the MouseButton class and its properties, refer to *Chapter 7.3: The MouseButton Class*.

**14.2.4. OnMouseReleased**

A UI event fired when a mouse button is released. **This event is only fired if the mouse click was within the bounds of the gadget!**

**Class Name: MouseReleasedEvent**

**Parameters:  
MouseButton button –** The mouse button that has been released.

For further information on the MouseButton class and its properties, refer to *Chapter 7.3: The MouseButton Class*.

**14.2.5. OnMouseEnter**

A UI event fired when the mous=e enters the screen.

**Class Name: GenericEvent**

**Parameters:**This event has no parameters.

**14.2.6. OnMouseLeave**

A UI event fired when the mouse leaves the screen.

**Class Name: GenericEvent**

**Parameters:**This event has no parameters.

**14.2.7. OnFocusLost**

A UI event fired when the current Renderer used for rendering loses focus.

**Class Name: GenericEvent**

**Parameters:**This event has no parameters.

**14.2.8. OnFocusGained**

A UI event fired when the current Renderer used for rendering gains focus.

**Class Name: GenericEvent**

**Parameters:**This event has no parameters.

**14.2.9. OnMouseMove**

A UI event fired when a mouse button move is completed – i.e., when the mouse comes to rest again after it has been moved.

**Class Name: MouseMoveEvent**

**Parameters:  
MouseButton button –** The mouse button that has been released.

For further information on the MouseButton class and its properties, refer to *Chapter 7.3: The MouseButton Class*.

**14.2.10. OnRender**

A UI event fired on the render of each frame.

**Class Name: RenderEvent**

**Parameters:  
Renderer cRenderer – The Renderer that is being rendered to.**

**14.2.11. OnShutdown**

A UI event fired on the engine being shut down, just before all UI gadgets are destroyed by the **UIManager::Shutdown** method.

**Class Name: ShutdownEvent**

**Parameters:  
Renderer cRenderer –** The Renderer that is about to be shutdown.

**14.3. UI Gadgets**

**14.3.1. Gadget Base Properties**

The **Gadget** class defines certain base properties that are shared between **all Gadgets**. These are used for the base event handlers (described in *Chapter 13.1: UI Overview*) and are also used to allow a more expansive range of drawing options for those who wish to write custom gadgets (refer to *Chapter 13.4: Expanding the UI System with Custom Gadgets* for an overview of custom gadgets.)

**Gadget** inherits from **Renderable** and therefore has Position and Size properties (for further information on Renderables, refer to *Chapter 1.4: The Renderable Class*)

The **Gadget** class additionally provides basic event handling for the **OnMouseMove, OnMousePressed** and **OnMouseRelease** handler, through the **MouseMove**, **MousePressed,** and **MouseReleased** methods. It is recommended to call these methods at the start of any custom event handlers for these events you may create in order to allow for the handling of events. Additionally, most prebuilt Gadget types have code for handling various events that must be called if writing a Gadget that inherits from one of the inbuilt classes.

**Properties**

**Color BackgroundColor:** The background color of the Gadget. The default value is ARGB (**0,0,0,0**), or transparent.

**Color ForegroundColor:** The foreground color of this Gadget – what constitutes the foreground is context-dependent, but is usually any text that may exist within the gadget. The default value is ARGB (**0,0,0,0**), or transparent.

**Color HoverColor:** The background color of the Gadget when hovered. The default value is ARGB (**0,0,0,0**), or transparent. To not have a hover color, set the value of this property to the value of **BackgroundColor.**

**Color PressedColor:** The background color of the Gadget when it is being pressed. The default value is ARGB (**0,0,0,0**), or transparent. To not have a pressed color, set the value of this property to the value of **BackgroundColor.**

**Color BorderColor:** The color of the gadget’s border. The default value is ARGB (**0,0,0,0**), or transparent. To not have a pressed color, set the value of **BorderSize** to (**0,0**).

**Vector2 BorderSize:** A Vector2 value indicating the size of the border. The default value is **0,0,** or no border.

**bool Filled:** A Boolean determining if the background of this Gadget is filled or not.

**bool Pressed:** A Boolean determining if the Gadget currently has the mouse pressed on it.

**bool Focused:** A Boolean determining if the Gadget is currently focused. This indicates that the last mouse click was within the Gadget’s area and no other mouse clicks outside of the Gadget’s area have occurred. This makes the gadget eligible to receive key press events.

**string Font:** A string corresponding to a FriendlyName property of a font loaded using the Font Manager to be used in the rendering of the Gadget’s text. For more information on the Font Manager, refer to *Chapter 12: Fonts*.

**14.3.2. Buttons**

A button is a simple Gadget that simulates a push button. The object implements a render event handler by default, so if you wish to extend this Gadget the **Render** method must be called in your gadget’s **OnRender** event handling.

**Properties**

**string Text:** The text of this button. It is always drawn in the centre of the button.

**TTF\_FontStyle style:** A TTF\_FontStyle enum value of the button’s text. For further information on the values of this enum, refer to *Chapter 14.3: Drawing Text.*)

**14.3.3. ListBoxes**

A list box is a UI gadget providing a list of selectable items. The list box keeps track of the current selected item and will automatically display It when it is clicked. There are also methods for creating items and default event handlers (the **Render, ListBoxMousePressed, ListBoxMouseReleased,** and **ListBoxMouseMove** methods) for the **OnRender, OnMousePressed, OnMouseReleased,** and **OnMouseMove** events respectively.

**Properties**

**List<ListBoxItem> Items:** Read-only list of **ListBoxItem** objects making up the items of the ListBox. To add an item to this list, use the **AddItem** method.

**bool DontAlternateItemColors:** Determines if item colors will be alternated (**false**) or not (**true**). The default value is **false**, which will alternate item colors.

**int AlternateItemColorsAmount:** Determines how much item colors will be alternated between each item. This value is subtracted from the current color value (depending on if the ListBox has the mouse clicking on it, the mouse hovering over it or not), so a negative value will make the alternate item color brighter than the standard item color.

**Methods**

**AddItem(ListBoxItem item):** Adds a new list box item to the ListBox. For further information on ListBoxItems, refer to *Chapter 13.3.4: ListBoxItems.*

**14.3.4. ListBoxItems**

Define an Item in a ListBox. These can only be used when passed to a ListBox using the **ListBox::AddItem** method.

**Warning: As the events for ListBoxItems are passed through from their parent ListBox, only the OnMousePressed, OnMouseReleased, OnMouseMove, and OnRender events are received for ListBoxItems!**

**Other events will not be fired!**

**Properties**

**string Text:** The text of the event handler.

**Methods**

**Constructor(string text):** Constructor for ListBoxItem. Sets the **Text** property to the value of the **text** parameter.

**14.3.5. TextBoxes**

A single-line textbox. Multi-line textboxes are scheduled for the next release.

This Gadget has built-in event handlers for the **OnKeyPressed** and **OnRender** events (via the **Render** and **KeyPressed** methods respectively).

**Properties:**

**string Text:** The text the user has typed into the TextBox. The default value is **100**.

**bool HideCursor:** A Boolean value determining if the text boxes’ cursor is hidden or not.

**int Capacity:** Determines the maximum length of the text box in characters.

**short CursorThickness:** Determines the thickness of the text cursor in pixels. The default value is **2.**

**Color CursorColor:** A Color value determining the color of the text cursor. The default value is white (ARGB **255,255,255,255**).

**int CursorBlinkFrequency:** An integer value determining the number of frames between cursor blinks. The default value is **30**, or a 30-frame blink frequency.

**int CursorBlinkLength:** An integer value determining the number of frames the cursor will blink for. The default value is **50**, or a 50-frame blink length.

**Methods:**

**Constructor(int capacity):** Constructor for TextBox. Sets the **Capacity** property to the value of the **capacity** parameter.

**14.3.6. CheckBoxes**

Implements a simple checkbox. This gadget provides built-in event handlers for the **OnRender** and **OnMousePressed** events (using the **Render** and **MousePressed** methods, respectively).

**Properties**

**Checked:** Determines if this CheckBox is checked. If it is, a checkbox will be drawn using the PrimitiveRenderer. For further information on the Primitive Renderer, refer to *Chapter 8: The Primitive Renderer*.

**Thickness:** Determines the thickness of the lines used for drawing the checkmark in the CheckBox, in pixels. The default value is **2**, for a 2-pixel line.

**14.4. Expanding the UI System with Custom Gadgets**

Lightning’s UI system is designed to be expandable. To do this, it is as simple as creating a new class in your game assembly that inherits from the Lightning **Gadget** class. However, there are a few considerations that the developer needs to take into account when developing custom Gadget class:

* It is highly recommended to give your event handlers unique names, and to call the default Gadget event handlers, if they exist, before using your custom event handlers. If you do not do this, pressed and hover colors will break.
* This is also the case when extending Lightning’s inbuilt gadgets – call their event handlers before your own, which will also call the base Gadget event handlers.
* You must always call **UIManager::AddGadget** before the Gadget will be rendered.
* **All** constructors must inherit from the base Gadget constructor by using the keyword **: base()** after the constructor class name.

**15. Text Rendering & Font Management**

**15.1. Font Manager Overview**

The basic text rendering API provided by the **PrimitiveRenderer** (see *Chapter 8.1.8: Text* for further information on the debug text rendering API) is not nearly enough for use in a game.

To solve this problem, a **FontManager** asset manager has been implemented to take care of rendering text to the screen. This API uses SDL\_ttf and supports anti-aliasing (of several types), text styling, background and foreground colors, outlines, and numerous other functions. The FontManager operates on one primary primitive – a **Font**. A font defines a single font and provides methods and properties for manipulating, loading, and unloading that font.

To draw text, the method **FontManager::DrawText** is called with the font and properties of the text you wish to draw (for further information on how to draw text using this method, refer to *Chapter 14.3: Drawing Text*)

**Warning: One Font object only corresponds to a single size of a font face. If you wish to load a font that has already been loaded, but with a different size, you must load a new font!**

**Only TrueType (.ttf) fonts are supported in this release! This will be corrected in the next release.**

**15.1.1. The FontManager Class**

**Methods**

**GetFont(string friendlyName):** Returns the font in the font manager’s internal font list with the **FriendlyName** property, if it is present. If it is not present, **null** is returned.

**LoadFont(string name, int size, string friendlyName, string path = null, int index = 0)**Loads a font.

If the **path** parameter is set, the **name** parameter is the **actual name of the font to be loaded** (i.e. in the name you would see in a font picker}  
If the **path** parameter is **not provided** or set to **null**, the **name** parameter the **file name of the font inside the system font directory.**

The **size** parameter is the size of the font in the industry-standard point measurement.

The optional **path** parameter is the path to the font, **excluding the .ttf extension**. If it is not set, or is set to **null**, the font will be loaded from the system font directory.

The **index** parameter is only useful when dealing with font files that have multiple indexes (multiple font faces within the same font, which almost no TrueType fonts “in the wild” actually do). If this is the case and there is more than one font fafce, this parameter can be used to select the loaded font face index. The default value is **0. Warning: If this parameter is set to a non-zero value when loading a font with only one index, the font will fail to load!**

**UnloadFont(Font font):** If the Font object **font** is present within the Font Manager’s internal font list, the Font Manager will remove this object from the internal font list and free its resources. **Do not** do anything further with a Font object that has been unloaded unless you load it again.

**UnloadFont(string friendlyName):** If a Font object with a **FriendlyName** property equal to the value of the **friendlyName** parameter is present in the within the Font Manager’s internal font list, the Font Manager will remove this font from the internal font list and free its resources. **Do not** do anything further with a Font object that has been unloaded unless you load it again.

**Vector2 GetTextSize(Font font, string text):** Returns a Vector2 corresponding to the size of the text **text** in the Font **font** when drawn on the screen. Extremely useful for positioning text. Works with multi-line text!

**Vector2 GetTextSize(string font, string text):** Returns a Vector2 corresponding to the size of the text **text** in a font with a **FriendlyName** property corresponding to a Font in the Font Manager’s internal font listwithwhen drawn on the screen. Extremely useful for positioning text. Works with multi-line text!

**DrawText:** Please refer to *Chapter 14.3: Drawing Text* for further information on this method and its use.

**15.2. Loading Fonts**

Fonts are loaded by calling the **FontManager::LoadFont** method (see *Chapter 14.1.1: The FontManager Class* for further documentation on the usage of this method).

A new Font must be created and loaded for every individual font size of a font you wish to load; the **FriendlyName** provided to LoadFont (accessible after the font has been loaded via using the **FriendlyName** property of any Font instance) is used to identify the font when calling FontManager methods after it has been loaded.

**15.2.1. The Font class**

**Properties**

**FriendlyName –** A name used to identify the font in FontManager method calls. **Must be unique for every size of every font that is loaded!**

**Name –** If a **path** parameter was supplied to thecall to **FontManager::LoadFont** that loaded this font, the **Name** property is the **actual name of the font to be loaded** (i.e. in the name you would see in a font picker}  
If the **path** parameter is **not provided** or set to **null**, the **Name** property is the **file name of the font inside the system font directory.**

**Size –** The size of the font in the industry-standard point measurement.

**SmoothingType – Obsolete! Do not use!**

**Handle –** Read-only pointer to the unmanaged memory containing the TTF\_Font structure of this font.

**15.3. Drawing Text**

Text drawing is handled using the **FontManager::DrawText** method:

public static void DrawText(Renderer cRenderer, string text, string font, Vector2 position, Color foreground, Color background = default(Color), TTF\_FontStyle style = TTF\_FontStyle.Normal, int resizeFont = -1, int outlineSize = -1, int lineLength = -1, FontSmoothingType smoothing = FontSmoothingType.Default, bool snapToScreen = false)

The parameters of this method control the appearance of the text; there are numerous ways to modify how the text appears on-screen.

**Required Parameters**

**Renderer cRenderer –** The Renderer to draw the text to.   
  
**string text –** The text to draw to the screen. Multi-line text is supported and **\n** characters are automatically interpreted as line breaks and are used to flow the text.

**string font –** A string corresponding to a **FriendlyName** property of a font loaded using the Font Manager. If a font with a corresponding FriendlyName is not found, an error is thrown with detailed information about the font.

**Vector2 position –** A Vector2 instance indicating the position that the **top-left** of the text will

be drawn at on the screen.

**Color4 foreground –** A Color4 instance indicating the color of the text that is to be drawn.

**Optional Parameters**

**Color4 background –** A Color4 instance indicating a background color to be drawn behind the text.

**TTF\_FontStyle style –** A **TTF­­\_FontStyle** enum value, enumerating style of font styling to use when rendering the text. The values of this enum are **Normal** (indicating no font styling), **Bold, Italic, Underline,** and **Strikeout.** The default value is **Normal**.

**resizeFont –** Resize the font before drawing. The default value is **-1**, which turns off this feature. **Warning: Using this parameter may cause your text to look pixelated or blocky. It is highly recommended to load multiple font sizes for your needs.  
Using this parameter will also change the original value of the Font’s Size property and is slow as all cached glyphs are annihilated!**

**outlineSize –** The size of the font’s outline in pixels. Range is 1 to the font’s size in point. The default value is **-1**, which will turn off outlined text.

**lineLength –** This will cut off the current line and generate a new line of text after a number of pixels equivalent to the value of this parameter. The default value is **-1**, which will simply treat **/n** characters as line breaks.

**smoothing –** Determines what type of font smoothing to use. The valid values are:

**Blended –** The text will be anti-aliased before being rendered to the screen.  
**Shaded –** A basic shading will be done to the font before it is rendered to the screen.  
**None –** No processing will be done to the font before it is rendered to the screen.

**snapToScreen –** Determines if the pixel will be drawn in world-relative space (false) or screen-relative space (true). The default value is **true.**

**15.4. Text Example**

The **FontAndText** example provides an excellent demonstration of the Font Manager and font APIs, and should allow developers to understand the concepts of the API. To use it, simply navigate to the Lightning Examples link in the Lightning Software Development Kit folder of the Start Menu after installing the SDK, open **Lightning2.Examples.sln**, and then compile the **FontAndText** example by opening the FontAndText project, selecting **FontAndText** on the project dropdown, and then pressing the **Start** button, which will automatically compile and run the project.

**16. Localisation**  
  
**16.1.** **Localisation Overview**

Lightning provides a localisation engine in order to allow games to be internationalised for launch in different markets. To facilitate this, a localisation asset manager is provided as well as the ability to load an arbitrary number of localisation files. While the .NET standard localisation feature is supported, it is not recommended.

A localisation file is a **.ini** file in the **Content/Localisation** folder (this folder path is required by the design) containing a **Metadata** section containing basic metadata about the language:

**Metadata Section Properties**

**string Name:** A friendly name of the language. Optional, but highly recommended.

**string Description:** A descrption of the language. Optional.

**string Version:** Version information of the language. Optional.

It also contains a **Strings** section, containing an arbitrary number of localisation strings, with the key being a string ID used to identify the string in code and the value the actual string content. It is recommended as a matter of convetnion, but not strictly required, that localisation string IDs are in all caps and start with the prefix **STRING\_**,



*Example localisation file. This file defines one string, STRING\_TEST.*

**16.2. Localisation Settings in Engine.ini**

To set a language (which is mandatory for engine startup) to use, the **Language** value in the **Localisation** section of **Engine.ini** must be set to a valid .ini file in the **Content/Localisation** folder. The .ini extension **must** be omitted in order for it to load. An error will be issued and the engine will shutdown if an invalid localisation file is defined. The user may modify this setting at any time.

**16.3. Localising Text**

In order to localise text, the ID of the localisation string that you wish to use must be prefixed with the **#[** character and suffixed with the **]** character. In line with the localisation file example from *Chapter 15.1: Localisation Overview*, the string

#[STRING\_TEST]

becomes the string Test string when localised.

All text rendering APIs, such as **PrimitiveRenderer::DrawText** and **FontManager::DrawText** localise text by default. They also additionally have a **localiseText** parameter that can be used in order to prevent the text from being localised.

An example of drawing a localised string from a test program;

PrimitiveRenderer.DrawText(cRenderer, "Loc string test: #[STRING\_TEST] aaaaaa #[STRING\_TEST] #[STRING\_TEST] bbbbbb", new Vector2(500, 150), Color.FromArgb(255, 0, 0, 255));

results in the text “Loc string test: Test string aaaaaa Test string test string bbbbbb” being drawn in the ARGB color **255, 0, 0, 255** (deep red), at the position **500, 150**:



While the text is nonsensical and additionally obscured by other rendered objects, it is a good demonstration of the Lightning localisation system’s ease of use and capabilities.

**17. Audio**

**17.1. Audio Manager Overview**

Lightning provides an audio engine in order to facilitate playing in-game audio (of MP3, FLAC, ModTracker, MIDI, and Ogg Vorbis format) and sound effects. Positional audio, repeat control and volume settings are also supported. The asset manager for audio is the **AudioManager**, which contains a list of **AudioFile** instances relating to each loaded audio file. An audio file is loaded using the **AudioManager::LoadFile** method – an optional friendly name can be provided for use in later method calls. They can also be acquired using the **AudioFile::GetFileWIthName** and **AudioFile::GetFileWithPath** method.

**17.1.1. The AudioManager class**

The AudioManager class, the asset manager for audio, provides utilities for loading and managing audio files, as well as acquiring previously loaded audio files for future use.

**Methods**

**LoadFile(string path, string name = null):** Loads an audio file from **path** and adds it to the audio manager’s internal audio file list. The friendly name is provided by the optional **name** parameter. If a name is not provided, it will be set to the name of the file, including its extension and without a path. An error will be thrown if the **path** parameter does not correspond to a file on disk.

**Warning: The audio file must be of either MP3, FLAC, MOD, MIDI, or OGG format. No other file format is supported!**

**UnloadFile(AudioFile file):** Unloads an audio file and removes it from the audio file list if it has previously been loaded.

**AudioFile GetFileWithName(string name):** Returns an AudioFile in the AudioManager’s internal file list with the friendly name property **name,** if it exists. If it does not exist, **null** is returned.

**AudioFile GetFileWithPath(string name):** Returns an AudioFile in the AudioManager’s internal file list with Path property **path,** if it exists. If it does not exist, **null** is returned.

**17.2. Audio Files**

An **AudioFile** is the class that represents a specific audio file in Lightning. It provides methods and properties for managing audio. Audio files use SDL\_mixer’s Chunk functionality in order to allow the playing of an arbitrary number of audio files at the same time. Positional audio is also supported as well as basic effects.

**17.2.1. The AudioFile Class**

**Properties**

**IntPtr AudioHandle**: Read-only pointer to the unmanaged MIX\_Chunk structure containing information about the audio file.

**string Name:** A name used to describe the audio file. If not provided at the time of the file being loaded, the value is the file name (including extension) that the audio file was loaded from.

**bool Playing:** Read-only Boolean property determining if the current audio file is playing.

**int Repeat:** Sets the number of times the audio file will repeat. If this property is set to **0**, the file will not repeat and only play once. If it is set to **-1**, it will endlessly repeat. Otherwise, it will repeat (**Repeat – 1)** times.

**string Path:** Read-only string property determining the path to the audio file on disk that the instance of **AudioFile** was loaded from. To set this property, load the audio file using **AudioManager::LoadFile.** Refer to *Chapter 17.1.1: The AudioManager class* for further information on the Audio Manager.

**bool PositionalSound:** Determines if positional sound is enabled for this audio file. If it is enabled, the sound volume as set using the **volume** parameter of the **SetVolume** method will be multiplied utilising the formula (RealVolume / (magnitude / 12) \* 128); where **RealVolume** is the volume multipled by 128 and the **magnitude** is the absolute distance between the **Position** property of the audio and the position of the current Renderer’s camera.

If this value is zero, or the current Renderer does not have a **Camera** property, the calculation will not be applied.

**Vector2 Position:** The position of this audio file. Ignored if the **PositionalSound** property is not set to **true**.

**Methods**

**Play():** Starts playing the audio using the properties that the user has set.

**Pause():** Pauses the audio. The audio will resume from the point it was paused when resumed using the Play method.

**Stop():** Stops the audio. This is identical to Pause, except the audio will resume from the start when Play is called instead of the point it was stopped from.

**SetVolume(double volume):** Sets the volume (or maximum volume if the **PositionalSound** property is set to true). The range of valid values is between zero and one.

**17.3. Playing & Managing Audio**

To use an audio file it first must be loaded using the **AudioManager::LoadFile** method. It is recommended to provide a friendly name for the audio when loading it, so that Audio Manager calls better elaborate the purpose to other programmers who may be working on your project. For our example, we will use a basic sound effect for a nearby in-game item as an example scenario. It should be positional, as we want the player to notice it nearby and be drawn towards the sound, and we want a very high volume when the player is over it – so we want it to be maximum volume. Using Lightning, this is very simple:

**Warning: The audio file must be of either MP3, FLAC, MOD, MIDI, or OGG format. No other file format is supported!**

AudioManager.LoadFile(“audio.ogg”, “audioOgg”);

However, just because we have loaded an audio file does not mean we can immediately manipulate it. This will be corrected in the next release. In order to actually use the audio file, we must use either **AudioManager::GetFileWithPath** or **AudioManager::GetFileWithName**. As we have specified a name using the **name** parameter (**audioOgg**), we will use the GetFileWithName method:

AudioFile audioOgg = AudioManager.GetFileWithName(“audioOgg”);

This method will throw an error if **audioOgg** does not correspond to an audio file in the internal audio file list of the AudioManager.

Then, we can set various properties of the Audio file. In our case, we do want positional sound, so we need to set a position:

audioOgg.Position = new Vector2(100, 100);

We do *not* want it to repeat, so we set the **Repeat** property to 0. As the default for a **System::Int32** in C# is zero, we do not need to set it.

We want maximum volume – the default value – and do not wish to set any other properties, so we can simply play the audio – which is as simple as calling the **Play** method.

audioOgg.Play();

**18. System Information & Enforcing System Requirements**

**18.1. The SystemInfo class**

Information about the system that a Lightning game is currently running on can be obtained via the **SystemInfo** class. This information is acquired using various APIs at engine initialisation, and can be queried by the user at any time. The engine also uses the values stored within this class in order to enforce system requirements for the game.

**Properties**

**uint ScreenResolutionX**: Holds the horizontal screen resolution of the first monitor in pixels.

**uint ScreenResolutionY**: Holds the vertical screen resolution of the first monitor in pixels.

**int SystemRam**: Holds the amount of system RAM in MiB.

**SystemInfoCPU Cpu**: Holds information about the user’s CPU. For further information on the SystemInfoCPU class, refer to *Section 17.1.1: The SystemInfoCPU class*.

**SystemInfoOperatingSystem OperatingSystem:** Holds information about the current operating system. For further information about the values of the SystemInfoOperatingSystem enum, prefer to *Section 17.1.3: The SystemInfoOperatingSystem Enum.*

**18.1.1. The SystemInfoCPU Class**

**Properties**

**Threads:** The number of logical processors in the system. **Warning: This is not the same as the number of CPU cores, and is usually the value referred to in most mainstream publications as “threads”, hence its name. If hyperthreading is enabled, the value may be up to twice as much as the real number of CPU cores!**

**Architecture SystemArchitecture:** The architecture of the system Lightning is running on. Valid values are **x86 (0), AMD64 (1), Arm32 (2), Arm64 (3), Wasm (4), and S390x (5)**.

S390 is used for Linux on IBM Z, and Wasm is used for WebAssembly. For further information, refer to the documentation for **System.Runtime.InteropServices.Architecture** on MSDN:

<https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.architecture?view=net-6.0>

**Architecture ProcessArchitecture:** The architecture of Lightning itself. This value may be different to the processor architecture (such as when running x64 Lightning using ARM64 x64 Emulation in Renderers 11 and later – x86 and arm32 are not officially supported architectures). Valid values are the same as for **SystemArchitecture.**

**SystemInfoCPUCapabilities Capabilities:** Enumerates CPU capabilities, such as supported instruction sets. For further information on enumerated CPU capabilities, refer to *Chapter 17.1.2: The SystemInfoCPUCapabilities Enum*.

**18.1.2. The SystemInfoCPUCapabilities Enum**

**Flags**

**MMX (0x1):** This system supports the MMX instruction set. Should always be set on any modern x86, 0 on ARM.  
**ThreeDNow (0x2):** This system supports the 3DNow! Instruction set. Should always be set on any modern AMD x64 processor between the K6-2 and Phenom II. **Note: PREFETCH and PREFETCHW are supported on models all AMD and Intel processors.  
RDTSC (0x4):** This system supports the RDTSC instruction. If it is at least a Pentium, Cyrix 6x86, or AMD K5, which I hope your CPU is, then this is the case. (if it isn’t, how are you running this)  
**AltiVec (0x8):** This should always be 0. I presume you do not have a PowerPC CPU with AltiVec extensions and have single-handedly ported .NET Core, Lightning, and all of its dependencies to it.  
**SSE (0x10):** This system supports the SSE instruction set. Therefore, it is at least a Pentium III. Which it should be.  
**SSE2 (0x20):** This system supports the SSE2 instruction set. Therefore, it is at least a 2000-era, Williamette Pentium 4. Which it should be.  
**SSE3 (0x40):** This system supports the SSE3 instruction set. Therefore, it is at least a Prescott Pentium 4. Which it, again, should be.  
**SSE41 (0x80):** This system supports the SSE4/4.1 instruction set (this is how SDL implements it). **Warning: No, SSE4a DOES NOT COUNT!  
SSE42 (0x100):** This system supports the SSE 4.2 instruction set. Therefore, it is at least a Nehalem first-generation Core CPU. Which it should be, but there’s a small chance of it actually not being. If so, I am sorry for your loss.  
**AVX (0x200):** This system supports the AVX instruction set.  
**AVX2 (0x400):** This system supports the AVX2 instruction set.  
**AVX512 (0x800):** This system supports the AVX512 instruction set. **Warning: This instruction set is only supported on AMD Zen 4 and newer, and on Intel CPUs between Knights Landing (server) / Skylake (client) and Tiger Lake (by default) / Alder Lake (with firmware and BIOS modifications)!  
NEON (0x1000):** This system supports the ARM NEON instruction set. This is present in most, but not all, ARMv8.x and ARMv9 CPUs. Will never be set on x86.  
**ARMSIMD (0x2000):** This system supports the ARM SIMD / Scalable Vector Extensions instruction set.

**18.1.3. The SystemInfoOperatingSystem Enum**

The SystemInfoOperatingSystem Enum holds the current operating system and version (for Renderers and macOS) that is running Lighjtning. One enum value represents all Linux versions, as there are too many variables in a Linux system to allow for appropriate version testing. System requirements will only test for versions of the user’s installed operating system.

**Values:**

**Renderers:**

**Win7 (0):** Renderers 7   
**Win8 (1):** Renderers 8   
**Win8.1 (2):** Renderers 8.   
**Win10TH1 (3):** Renderers 10, Threshold 1 (1507 release) / LTSB 2015   
**Win10TH2 (4):** Renderers 10, Threshold 2 (1511 release)   
**Win10RS1 (5):** Renderers 10, Redstone 1 (1607 release) / LTSB 2016  
**Win10RS2 (6):** Renderers 10, Redstone 2 (1703 release)   
**Win10RS3 (7):** Renderers 10, Redstone 3 (1709 release)  
**Win10RS4 (8):** Renderers 10, Redstone 4 (1803 release)  
**Win10RS5 (9):** Renderers 10, Redstone 5 (1809 release) / LTSC 2019  
**Win1019H1 (10):** Renderers 10, 19H1 release (1903 release)   
**Win1019H2 (11):** Renderers 10, 19H2 release (1909 release) – Cumulative Update for 19H1   
**Win1020H1 (12):** Renderers 10, Vibranium (20H1 release)  
**Win1020H2 (13):** Renderers 10, 20H2 – Cumulative Update for 20H1  
**Win1021H1 (14):** Renderers 10, 21H1 – Cumulative Update for 20H2  
**Win1021H2 (15):** Renderers 10, 21H2 – Cumulative Update for 21H1  
**Win1022H2 (16):** Renderers 10, 22H2 – Cumulative Update for 21H2 / LTSC 2021  
**Manganese (20):** Renderers 10 Dev Channel, 2019-2020 – Build 19480-19645  
**Iron (21):** Renderers Server 2022, version 21H2 (Build 20348)  
**Win11 (22):** Renderers 11, 21H2 release (Build 22000)  
**Nickel (23):** Renderers 11, 22H2 release (Build 22621)  
**Copper (24):** Renderers 11, Dev Channel (Build 25054)

**macOS:**

**MacOS1013 (50):** macOS 10.13.x (High Sierra)  
**MacOS1014 (51):** macOS 10.14.x (Mojave)  
**MacOS1015 (52):** macOS 10.15.x (Catalina)  
**MacOS11 (53):** macOS 11.x (Big Sur)  
**MacOS12 (54):** macOS 12.x (Monterey)  
**MacOS13 (55):** macOS 13.x (Ventura)

**Linux:**

**Linux (100):** Linux (of any version supported by .NET 6.0)

**18.2. Enforcing System Requirements**

The values used in the **SystemInfo** class can be used to enforce system requirements in-game. System requirements can be enforced on hardware threads, RAM, CPU features, and operating system (only the user’s current operating system will be tested).

The engine tests the system capabilities in **SystemInfo** against the system requirements if they are set, and will throw a fatal error during engine initialisation and shut down if the requirements are not satisfied.

All GlobalSettings relating to system requirements are optional.

These are controlled using the **MinimumSystemRam**, **MinimumLogicalProcessors**, **MinimumCpuCapabilities,** and **MinimumOperatingSystem** GlobalSettings (for more information on GlobalSettings, refer to *Chapter 5: GlobalSettings*).

The **MinimumSystemRam** GlobalSetting specifies the minimum amount of RAM that the game requires to run. This value is in MiB.

The **MinimumLogicalProcessors** GlobalSetting specifies the minimum number of logical processors that the game requires to run. This value is an integer. **Warning: This is not the same as the number of CPU cores, and is usually the value referred to in most mainstream publications as “threads”, hence its name. If hyperthreading is enabled, the value may be up to twice as much as the real number of CPU cores!**

**The MinimumCpuCapabilities** GlobalSetting specifies the minimum CPU capabilities that the game requires to run. For further information on the values for this GlobalSetting, refer to *Chapter 17.1.2: The SystemInfoCPUCapabilities Enum*.

**19. Packaging**

Lightning supports a packaging engine in order to aid in file obfuscation and game packaging. Compression will be implemented in a future release. It is recommended that you develop your game with this functionality off and turn it on when it is time for your game to release in order to allow for easier iteration and a better game development workflow.

The packaging system is a fairly basic system using WADfiles. These do not have any similarity to the classic WAD file format used in DOOM (nor does it have any relationship with Quake’s WAD2 or Half-Life’s WAD3), I just like the “Where’s All the Data” acronym :P.

**19.1. Packaging Your Game**

Packaging your game is a simple affair. A package is generated from your game content files (**Engine.ini** is excluded, and must always be included in your game’s distribution in the **Content** folder).

The GlobalSetting **PackageFile** in **Engine.ini** is then changed to a relative path to your package file, and the GlobalSetting **ContentFolder** is changed to a relative path to the folder your game’s content will be extracted to. This is then extracted by the Lightning engine the first time the game is run. If you wish to change this behaviour, refer to *Chapter 18.1.1: Deleting Extracted Files on Exit*.

**19.1.1. Deleting Extracted Files on Exit**

If you wish to extract all files every single time the game is run, all extracted files will optionally be cleaned up at exit if the **DeleteUnpackedFilesOnExit** GlobalSetting is set to true (for further information on the available GlobalSettings, refer to *Chapter 5: Settings, Global & Local*).

**19.1.2.** **Package Debugging using the InitSettings Class and Command-Line Options**

Lightning provides command-line options for package debugging purposes only, allowing you to deploy multiple packages and test them if necessary, They are implemented by the **InitSettings** class.

**PackageFile (-packagefile):** The package file to load game content from.

**ContentFolder (-contentfolder):** The content folder to extract game content to. Always relative to the game directory. Optional if **-packagefile** is provided – the default value is a **Content** subfolder of the game directory. If the specified folder does not exist, it will be created.

**Warning: These command-line options override the values that have been stored in the game’s Engine.ini file!**

**19.2. The MakePackage Tool**

The Lightning SDK provides a basic command-line tool for generating game packages, called **MakePackage,** located in the root of the SDK install directory.This will automatically generate a package compatible with the Lightning game engine:



There are two sets of required arguments depending on if the user wishes to extract or generate a package file.

**Required arguments for package mode:**

**-infolder:** Input folder to generate a package from.  
**-outfile:** Output game package file. It is recommended, but not strictly required, that this file have an extension of **.wad.**

**Required arguments for extract mode:**

**-infile:** Input file to extract files from.  
**-outfolder:** Output folder to extract files to.

**19.3. Optional MakePackage Arguments**

MakePackage also provides some additional, entirely optional arguments when generating a package file. These affect the behaviour of the package file and some of the metadata inside of the package file.

**Optional arguments:**

**-gamename [-name]:** An optional game name to place in the metadata of the package file.  
**-gameversion:** An optional game version to provide in the metadata of the package file.  
**-engineversion:** An optional engine version to specify the game is compatible. **Must** be in the format **major.minor.revision.build.** **Warning: The engine will not allow games that do not have the same major and minor versions from being extracted and run!  
-allowbinaries:** This allows binary files (files with a **.exe, .dll, .sys, .ocx, .scr, .cpl, .winmd, and .rll** extension, case-sensitive, as well as the **Engine.ini** file) to be added to the wad file. **Warning: This will cause issues with the game once it has been run and is not recommended!.**   
**-packagemode:** This sets the packaging mode of the package. There are currently two packaging modes available:

**None –** Simply stores the file into the package unmodified.  
**XOR –** XORs the package files with a key and then increments all bytes by three in order to obfuscate the package. **This is not secure or an even remotely passable encryption!**

**20. Debugging**

**20.1. Debugging Your Game**

Lightning game assemblies are fully compatible with all standard Visual Studio debugging tools. If you wish to debug the engine itself, Lightning is also compatible with debugging using dnSpy or ILSpy. The engine code (contained in **LightningGL.dll** and **LightningPackager.dll**)is presently not obfuscated, so it is fairly easy to debug in order to find crashes or bugs in the engine.

**20.2. The FPS Meter**

Lightning provides an FPS meter for measuring the performance of your game. It can be toggled using the GlobalSetting **ShowDebugInfo**. When this GlobalSetting is set to **true**, debugging information will be displayed at the top-left corner of the screen similar to this:



The first line is the current rate of frames per second and the frametime in milliseconds.  
The second line is the number of frames rendered since the engine was started.

If the **MaxFPS** setting is set to limit the maximum number of frames rendered per second, and the engine is running below its target framerate, a red line of text under the frame count will be drawn to indicate this.

**20.3. The Performance Profiler**

In addition to being compatible with the Visual Studio performance profiler, Lightning incorporates basic performance profiling functionality in order to evaluate the performance of a game using the engine.

To use the performance profiler, set the **ProfilePerformance** GlobalSetting to **true**. Lightning will then record the FPS and frame time (in milliseconds) for each frame the engine is running. When the engine is shut down, all of this information is dumped to a CSV file ion the game directory starting with **Lightning-Perf-** and then the date and time the engine was started:



This CSV file has three columns – **Frame, FrametimeMs, and Fps**.

The **Frame** column holds the frame number.  
The **FrametimeMs** column holds the time, in milliseconds, it took for each frame to render.  
The **Fps** column holds the frames per second rate at the time of the frame **Frame**.



*Example data from a test project.*

At the bottom of the CSV file, some basic performance metadata is recorded:  


The **Average** value is the average framerate recorded during profiling.  
The **99th%ile** value is the 99th percentile of the framerates reached – i.e. the 1% High value.  
The **95th%ile** value is the 95th percentile of the framerates reached – i.e. the 5% High value.  
The **5th%ile** value is the 5th percentile of the framerates reached – i.e. the 5% Low value.  
The **1st%ile** value is the 1st percentile of the framerates reached – i.e. the 1% Low value.  
The **0.1st%ile** value is the 0.1st percentile of the framerates reached – i.e. the 0.1% Low value.

The data in the CSV file can be used, for example, for graphing performance over time:  
  
*An example graph of Lightning performance from a test project that uses all of Lightning’s functionality.*

**21. Advanced Usage**

**21.1. Interfacing with SDL**

Lightning uses C# bindings based on a modified version of SDL2-CS, using the **LightningBase** namespace for calling SDL, SDL\_image, SDL\_ttf, and SDL\_mixer APIs.

The source code of SDL2-CS can be found here:  
<https://github.com/flibitijibibo/SDL2-CS>

The modifications used by Lightning’s fork of SDL2-CS are primarily done for easier interfacing with C# and Lightning specifically, and are not intended for general-purpose use for non-Lightning projects. Upstream changes are merged into the fork when they occur.

The source code to Lightning’s fork of SDL2-CS can be found in the **Examples** folder (accessible through the “Lightning Examples” link on the start menu after installing the Software Development Kit), in a 7-Zip archive called **LightningBase-CS.7z.** This is strictly intended for understanding how Lightning’s SDL2-CS differs from the base SDL2-CS, as well as Lightning’s custom SDL2\_gfx fork, and is **not intended** for general use!

**Warning: No support will be provided for using LightningBase-CS outside of Lightning.**

**Note:** Lightning currently uses SDL 2.0.22 (the latest at feature freeze for 1.0), SDL2\_image 2.0.5, SDL2\_ttf 2.0.18, and SDL2\_mixer 2.0.4. and APIs that were added in later versions of the SDL libraries (such as SDL version 2.24.0 and SDL\_mixer version 2.6.0) are not supported for use, but are intended to be usable in the next release of Lightning version 1.1.

It is additionally intended that the SDL2\_ttf bindings are replaced with custom FreeType bindings in order to facilitate wider use of FreeType features.

**21.2. Lightning + NativeAOT**

Lightning is specifically designed to be NativeAOT compatible. It does not intend to use any .NET functionality that is not compatible with NativeAOT. However, owing to the fact that NativeAOT is a preview feature intended for general release in .NET 7.0, a NativeAOT build of Lightning has **not been tested!** Therefore, **no guarantee of compatibility can be provided!**

**21.3. Extending the Engine**

While there is currently no official plugin functionality to extend the capabilities of the Lightning game engine, it is intended and encouraged to write plugins to extend the engine. They may be included as standard in future versions of the game engine. To write an engine extension, it is as simple as writing a .NET Assembly that depends on **LightningGL** and then use it in your game.

There are no limitations on what functionality may be added or replaced, however it is **strongly recommended** to clearly name your plugin classes with a prefix so that users know which API is standard and which API is for a plugin.

**22. Error Message Reference**

**Error Code 0**

An error occurred initialising SDL2.

**Error Code 1**

An error occurred initialising SDL2\_image.

**Error Code 2**

An error occurred initialising SDL2\_ttf.

**Error Code 3**

An error occurred initialising SDL2\_mixer.

**Error Code 5**

NCLogging::NCLoggingSettings was null when calling NCLogging::Init. This is almost certainly an engine bug.

**Error Code 6**

NCLogging::NCLoggingSettings::LogFileName was null when calling NCLogging::Init. This is almost certainly an engine bug.

**Error Code 7**

A null RendererSettings was passed to Renderer::Start.

**Error Code 8**

An SDL error occurred during Renderer creation in Renderer::Start.

**Error Code 9**

Attempted to load a Texture with a Path property corresponding to an invalid file.

**Error Code 10**

An SDL error occurred while loading a Texture. Refer to the error cause for more detailed error information.

**Error Code 11**

There was an internal error locking pixels for modification. This may be an engine bug.

**Error Code 12, 14**

Texture::GetPixel was called with an invalid x and y position. Modify the call to GetPixel such that the x and y positions are at least 0 and less than the texture size.

**Error Code 13**

An SDL error occurred while allocating a texture format for a Texture. Refer to the error cause for more detailed error information.

**Error Code 15, 17**

Texture::SetPixel was called with an invalid x and y position. Modify the call to SetPixel such that the x and y positions are at least 0 and less than the texture size.

**Error Code 18**

Attempted to draw a line with a thickness of below 1. Change the call to PrimitiveRenderer::DrawText.

**Error Code 19**

An SDL error occurred in NCMessageBox::Show.

**Error Code 21**

Attempted to load a nonexistent INI file.

**Error Code 22**

An unknown error occurred loading an INI file. View the error cause for exception information.

**Error Code 23**

An INI item does not have a value. Add a value to parse correctly.

**Error Code 24**

An INI section name was not terminated with a ] character.

**Error Code 25**

An invalid INI section entry was found. INI Section entries start with [ and are ended with ].

**Error Code 26**

An INI value was not within an INI section.

**Error Code 28**

Engine.ini failed to load.

**Error Code 29**

Engine.ini does not have a Localisation section.

**Error Code 30**

Engine.ini’s Localisation section does not have a valid Language value.

**Error Code 31**

An attempt to load an invalid localisation file occurred. There is likely an error in the localisation file.

**Error Code 32**

A loaded localisation file does not have a Metadata section.

**Error Code 33**

A loaded localisation file does not have a Strings section.

**Error Code 34**

Attempted to load a nonexistent font file. Adjust your call to Font::Load to specify an actually extant font or font file. Please note that system fonts must use the font names in c:\Renderers\fonts.

**Error Code 35**

Attempted to load an invalid localisation string.

**Error Code 36**

Attempted to load a non-TrueType font. Only TrueType fonts are supported.

**Error Code 37**

Attempted to load an invalid font size. Font size must be at least 1.

**Error Code 38**

An SDL2\_ttf error occurred during font loading.

**Error Code 39**

Attempted to draw text with an invalid font parameter. Change your call to DrawText or load the font you need to use.

**Error Code 41**

Engine.ini does not have a General section. One must be added.

**Error Code 44**

Attempted to load an AnimatedTexture without a set Size property. Set the Size property to at least 1 x 1 pixels.

**Error Code 45**

Attempted to load a TextureAtlas without a valid FrameSize property. Set it before calling TextureAtlas::Load.

**Error Code 46**

Attempted to load a TextureAtlas with a frame count under 1 in the X or Y direction. Set it before calling TextureAtlas::Load.

**Error Code 47**

Attempted to draw an invalid Index for a TextureAtlas. The index must be between zero and (TextureCount.X \* TextureCount.Y).

**Error Code 50**

You attempted to load an audio file with a path that does not exist.

**Error Code 51**

An SDL\_mixer error occurred while loading an audio file. See error description for the actual error.

**Error Code 54**

Attempted to load an AnimatedTexture property with an invalid Cycle. The cycle must have at least one frame.

**Error Code 56**

An error occurred initialising the audio device required for SDL2\_mixer.

**Error Code 61, 62, 124, 125**

Attempted to use the Light Manager without initialising it. The error code determines the called method.

**Error Code 70**

An exception occurred while initialising the Performance Profiler. No profiling will be performed.

**Error Code 80**

An SDL\_ttf error occurred while sizing text.

**Error Code 81, 82**

Attempted to call GetTextSize (80) or GetLargestTextSize (81) specifying an invalid Font parameter.

**Error Code 83**

Attempted to select an invalid ListBox SelectedIndex. SelectedIndex must be between 0 and the number of items in the list box – 1.

**Error Code 94**

A Texture Path was set, but the texture was not loaded.

**Error Code 95**

You attempted to shutdown Lightning without initialising it.

**Error Code 96**

Attempted to add a non-existent file to a PackageFileCatalog

**Error Code 98, 100**

Attempted to read a package file that does not exist.

**Error Code 99**

Attempted to read a non-package file as it does not have a package file magic.

**Error Code 100**

Attempted to read a package file with an incorrect version. Please update Lightning.

**Error Code 102**

A package file has an invalid file catalog magic and therefore is likely corrupt.

**Error Code 103**

Invalid command-line arguments were provided to the engine.

**Error Code 104, 128**

A fatal error occurred extracting a package. 104 indicates the error occurred during command-line option parsing, 128 indicates the error occurred during GlobalSettings parsing.

**Error Code 105**

A package file has an invalid header. It may be corrupted.

**Error Code 106**

Attempted to move a textbox cursor beyond the text length.

**Error Code 109**

An error occurred cleaning up the content directory when the GlobalSetting DeleteUnpackedFilesOnExit is set to true.

**Error Code 110**

An error occurred while writing to an INI.

**Error Code 111, 112, 113, 114**

These errors indicate the system requirements tests failed.

**Error Code 117**

Attempted to set a Renderer to a Renderer size of 191x47 or lower. The Renderer width must be at least 192 pixels and the Renderer height must be at least 48 pixels.

**Error Code 118**

Attempted to move a Renderer to a position not on the current monitor.

**Error Code 119**

An SDL error occurred during texture creation. See error cause for more detailed error cause information.

**Error Code 120**

Attempted to draw a particle effect without loading it.

**Error Code 122**

Attempted to draw a UI element with no size. You will not be able to see it.

**Error Code 123**

Specified an SDL rendering backend that is not supported. SDL chose a different rendering backend listed in the error information.

**Error Code 127**

An attempt was made to initialise a Renderer when using the Scene Manager. Set the DontUseSceneManager GlobalSetting to true if you wish to do this.

**Error Code 130**

Failed to initialise a scene. The specific scene that failed to initialise is in the error cause.

**Error Code 131**

No startup scene was set. Set the StartupScene GlobalSetting to set startup scenes.

**Error Code 132**

There are no scenes. Please set DontUseSceneManager to true in order to not use the Scene Manager, or define a scene by adding a class to your game assembly that inherits from the Scene class.

**Error Code 133**

Attempted to set a current scene name that does not correspond to a valid scene.

**Error Code 134**

An attempt was made to initialise a Renderer before initialising the engine. Call **LightningGL::Init** before using any Lightning methods.

**Error Code 135**

Called UIManager::RemoveElement with a gadget property that does not correspond to a Gadget loaded by the UI Manager.

**Error Code 57005 (0xDEAD)**

An unknown fatal error occurred during engine initialisation. This may indicate an error during your engine installation.