Welcome to the API Reference for the Lightning game engine. This document hopefully provides you with everything you need to develop games using the engine and to utilise the engine to its maximum potential.

Any comments about the documentation should be directed to Starfrost#9088 on Discord.

**Contents  
1. General Overview**1.1. Lightning API 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 Window manually  
2.3. Window 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. The Local Settings API **6. Cameras**6.1. Camera System  
6.2. Camera Types **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. Animation**10.1. Animated Texture Overview  
10.2. Animation Cycles **11. Lighting**11.1. Drawing Lights  
11.2. Screenspace Map  
11.3. Environmental Light **12. Particle Effects**12.1. Creating & Loading Particle Effects  
12.2. Playing Particle Effects  
12.3. Particle Effect Modes  
12.4. Particle Effect Parameters **12. UI**13.1. UI System  
13.2. UI Events  
13.3. UI Gadgets  
13.3.1. Buttons  
13.3.2. ListBoxes  
13.3.3. ListBoxItem  
13.3.4. TextBoxes  
13.3.5. CheckBoxes **14. Text Rendering & Font Management**14.1. The Font Manager  
14.2. Loading Fonts  
14.3. Drawing Text  
14.4. Text Properties **15. Localisation**15.1. Localisation   
15.2. Localisation Settings in Engine.ini  
15.3. Localised Strings **16. Audio**16.1. The Audio Manager  
16.2. Loading Audio  
16.3. Playing & Managing Audio **17. System Information & Feature Detection**17.1.The SystemInfo class  
17.2. Enforcing System Requirements  
17.3. Detecting Features **18. Packaging**18.1. Packaging Your Game  
18.2. The MakePackage tool  
18.3. Package File Options  
18.4. How to Run Your Game Packaged **19. Debugging**19.1. Debugging Your Game  
19.2. The FPS Meter  
19.3. The Performance Profiler **20. Advanced Usage (How to Interface with SDL)**20.1. Interfacing with SDL  
20.2. Lightning + NativeAOT  
20.3. Extending the Engine **21. Error Message Reference  
  
1. General Overview**

**1.1. Lightning API Style**The Lightning API is based on the concept of windows, asset managers (see Chapter 1.5)) and, optionally, scenes.  
  
Each Window 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 Window 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.**

**1.2. Namespaces**

**LightningGL**Provides all of the primary rendering APIs.  
**LightningPackager**Provides all of the packaging APIs.  
**NuCore.SDL2**Provides access to unmanaged SDL2, SDL2\_image, SDL2\_mixer, SDL2\_ttf, and my fork of SDL2\_gfx APIs and functions.  
**NuCore.Utilities**ProAvides utility APIs.

**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>

**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(Window cWindow)  
{

PrimitiveRenderer.DrawText(cWindow, “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 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 fairly consistent API with similar method names and usage, although they do not conform to a specific base class yet.

**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. 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 Window Manually using WindowSettings**

Lightning also supports use 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** to the **[General]** section of the **Content\Engine.ini** in your project, and then delete the **Scenes** folder and any files in it.

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

Lightning2.Init(args);

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

Then you have to create a window:

Window window = new Window();

Then, configure the **WindowSettings** manually (see *Chapter 2.3: WindowSettings* for documentation on the **WindowSettings** class.:

WindowSettings windowSettings = new WindowSettings();

Then the window has to be started using the instance of **WindowSettings** you have created:

window.Start(windowSettings);

If you compile the code as it is now, the window 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 (Window.Run())

{

// rendering code  
window.Render();

}

The call to **Window::Run** allows Lightning to start the frame, and the call to **Window::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 **WindowSettings** can be accessed using the **Settings** property of the Window.

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);

Window window = new Window();

window.Start(new WindowSettings

{

Title = "Basic Lightning2 Demo"

}); // use default windowsettings except title

while (window.Run())

{

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

window.Render();

}

**2.3. WindowSettings**

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

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

**Vector2 Size**The window size. Must be between 192x48 and the current screen resolution. Default is (960, 640). Can be changed at runtime.

**SDL\_WindowFlags WindowFlags**SDL\_WindowFlags for the window. See <https://wiki.libsdl.org/SDL_WindowFlags>. Default is the content of **GlobalSettings::WindowFlags**, which is **SDL\_WINDOW\_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 window. **Can only be acquired from user code, not set.**

**IntPtr WindowHandle**Internal only pointer to unmanaged memory containing the SDL\_Window struct of the window. **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 window. See chapter 6 – don’t use this property direcl

**BackgroundColour**The background colour of the window.

**2.4. Delta Time & Framerate Management**

Not all of Lightning is currently set up to use delta time. This will be repaired in the next release. The delta time is stored in the **Window::DeltaTime** property. To use delta-time for velocity calculations, simply multiply the value you are using by it, like so:

double realVelocity = velocity \* window.DeltaTime;

You must be using a method that receives a Window 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 capabilities and is the primary area of the Lightning Game Engine.

**3.2. LightningPackager:**

Provides WAD file packaging functions.

**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 **LightningGL::Scene.**

**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 window (any scene can also handle events passed to the window), and starting and shutting down scenes and the engine. Using the Scene Manager allows you to not write most of the required boilerplate code to use Lightning..

To use the Scene Manager, it is required to set a startup scene. 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.

**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(Window cWindow):** Called each frame.

**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 **Name** 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 Start Menu):

using LightningGL;

using static NuCore.SDL2.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(Window cWindow)

{

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

// change the scene

if (cWindow.EventWaiting)

{

if (cWindow.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 will occur on shutdown. This goes for all types of assets that are loaded using the Lightning 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**

**GLOBALSETTINGS\_PATH**

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

**ProfilePerf**

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

**EngineAboutScreenOnShiftF9**

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

**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.**

**LocalSettingsPath**

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

**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.

**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.

**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 Window Manually using WindowSettings* for further information on this topic.

**5.2.2. Localisation section**

**Language**

The language to use. Must correspond to a valid language INI file in the **Content\Localisation** folder. See *Chapter 14: Localisation* for information on Lightning’s localisation system.

**5.2.3: Graphics section**

**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.

**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.**

**WindowFlags**

The **SDL\_WindowFlags** of the window. See <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\_WINDOW\_SHOWN.**

To set the window as fullscreen, use **SDL\_WINDOW\_FULLSCREEN\_DESKTOP**, not **SDL\_WINDOW\_FULLSCREEN**. This is because **SDL\_WINDOW\_FULLSCREEN** forces the operating system to give semi-exclusive control of the window 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.

**RenderFlags**

The **SDL\_RendererFlags** of the window. 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**.**

**ResolutionX**

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

**ResolutionY**

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

**PositionX**

The initial horizontal position of the main game window on the screen. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_WINDOW\_FULLSCREEN** or **SDL\_WINDOW\_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 window’s **WindowSettings.Position.X.**

**PositionY**

The initial vertical position of the main game window on the screen. Will be ignored if **SDL\_RendererFlags** contains the **SDL\_WINDOW\_FULLSCREEN** or **SDL\_WINDOW\_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 window’s **WindowSettings.Position.Y.**

**WindowTitle**

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

**Renderer**

The rendering backend to use. Valid values 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 not 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.

**MinimumSystemRam**

The minimum about of system RAM, in MiB.

**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.

**MinimumCpuCapabilities**

The minimum CPU capabilities required for the game. See *Chapter 16.3: Enforcing CPU Capabilities* for more information on this topic and the values that can be set.

**MinimumOperatingSystem**

Enforces a minimum operating system for your game. The lowest version of Windows supported by .NET 6.0 is Windows 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.

**Windows values:**

**Win7** – Windows 7 (6.1.7600+)  
**Win8** – Windows 8 (6.2.9200+)  
**Win81** – Windows 8.1 (6.3.9600+)  
**Win10TH1** – Windows 10, original release (10.0.10240+)  
**Win10TH2** – Windows 10, Threshold 2 (1511 release) (10.0.10586+)  
**Win10RS1** – Windows 10, Redstone 1 (1607 release) (10.0.14393+)  
**Win10RS2 –** Windows 10, Redstone 2 (1703 release) (10.0.15063+)  
**Win10RS3** – Windows 10, Redstone 3 (1709 release) (10.0.16299+)  
**Win10RS4 –** Windows 10, Redstone 4 (1803 release) (10.0.17134+)  
**Win10RS5** – Windows 10, Redstone 5 (1809 release) (10.0.17763+)  
**Win1019H1 –** Windows 10, 19H1 release (10.0.18362+)  
**Win1019H2 –** Windows 10, 19H2 release (10.0.18363+)  
**Win1020H1 –** Windows 10, Vibranium release (10.0.19041+)  
**Win1020H2 –** Windows 10, 20H2 release (10.0.19042+)  
**Win1021H1 –** Windows 10, 21H1 release (10.0.19043+)  
**Win1021H2 –** Windows 10, 21H2 release (10.0.19044+)  
**Win1022H2 –** Windows 10, 22H2 release (10.0.19045+)  
**Manganese –** Windows 10, Insider Dev Channel 2019-20 (10.0.19480-10.0.19645)  
**Iron –** Windows Server 2022 (10.0.20348+)  
**Win11 –** Windows 11, Cobalt release (10.0.22000+)  
**Nickel –** Windows 11, 22H2 release (10.0.22621+)  
**Copper –** Windows 11, Dev Channel (10.0.25054+)

**MacOS values:**

**MacOS1013** – macOS 10.13.x (High Sierra)  
**MacOS1014** – macOS 10.14.x (Mojave)  
**MacOS1015** – macOS 10.15.x (Catalina)  
**MacOS11** – macOS 11.x (Big Sur)  
**MacOS12** – macOS 12.x (Monterey)  
**MacOS13** – macOS 13.x (Ventura)

**Linux values;**

**Linux** –any Linux.

**5.2.5: Scene section**

**StartupScene**The name of the startup scene. Required to use the Scene Manager. By default, the name of a scene is its class name.

**5.3: LocalSettings**

There is an experimental local settings API in the **LocalSettings** class. Local settings are automatically loaded at engine start if the value of **GlobalSettings::LocalSettingsPath** is not **null.** An error will be thrown if the value of the aforementioned LocalSettingsPath does not correspond to a file that exists.

**WARNING: This is experimental and not well tested functionality and the usage of this API is almost certain to change in newer versions, and there may be bugs in the current implementation. If you use it, expect your code to break in future versions of Lightning.**

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”);

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

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

To save the settings, call **LocalSettings::Save**, which will write out the content of **LocalSettings::LocalSettingsFile** to the value specified in **GlobalSettings::LocalSettingsPath**.

**6. Cameras**

Lightning provides a basic camera system to aid in game development. 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 two 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.

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 **-WindowSettings::Size::X / 2**.

When cameras are in use (signified by the **Settings::Camera** value of the current Window 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 **Window::SetCurrentCamera** with a valid **Camera** object:

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

**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 **Window::EventWaiting** should be checked:

if (window.EventWaiting)

{

// event handling code…

}

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

if (window.EventWaiting)

{

switch (window.LastEvent.type)  
 {

case SDL\_EventType.SDL\_KEYDOWN:

// key handling code here…

}

}

To use this API, you must add using NuCore.SDL2; 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. Lightning uses Keysym instead of keycodes as it is more accurate 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 it the documentation for SDL\_Keysym is here: <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 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 (Windows 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 NuCore.SDL2. 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.

**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(Window cWindow, Vector2 position, Color colour, bool snapToScreen = false)

Draws a pixel to the screen.

**Parameters:**

**cWindow:** The window to draw a pixel to  
**position:** The position to draw the pixel at.  
**colour:** The colour 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(Window cWindow, Vector2 start, Vector2 end, short thickness, Color colour, bool antiAliased = true, bool snapToScreen = false)

Draws a line to the screen.

**Parameters:**

**cWindow:** The window 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.  
**colour:** The colour 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(Window cWindow, Vector2 position, Vector2 size, Color colour, bool filled = false, bool antiAliased = false, bool snapToScreen = false)

Draws a circle to the screen.

**Parameters:**  
  
**cWindow:** The window 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.  
**colour:** The colour 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(Window cWindow, Vector2 position, Vector2 size, Color colour, bool filled = false, Color borderColour = default(Color), Vector2 borderSize = default(Vector2), bool snapToScreen = false)

Draws a rectangle to the screen.

**Parameters:**

**cWindow:** The window to draw the circle to.

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

**size:** The size of the rectangle.  
**colour:** The colour of the rectangle.   
**filled:** Determines if the rectangle will be filled or not. The default value is **false.**  
**borderColour:** The colour 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(Window cWindow, Vector2 position, Vector2 size, Color colour, int cornerRadius, bool filled = false, bool snapToScreen = false)

Draws a rounded rectangle to the screen.

**Parameters:**

**cWindow:** The window to draw the rounded rectangle to.  
**position:** The position of the rounded rectangle.

**size:** The size of the rounded rectangle.  
**colour:** The colour 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(Window cWindow, Vector2 point1, Vector2 point2, Vector2 point3, Color colour, bool filled = false, bool snapToScreen = false)

Draws a triangle to the screen.

**Parameters:**

**cWindow:** The window 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.

**colour:** The colour 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(Window cWindow, List<Vector2> points, Color colour, bool filled = false, bool antiAliased = false, bool snapToScreen = false)

Draws a polygon to the screen.

**Parameters:**

**cWindow:** The window to draw the polygon to.

**points:** A List of Vector2s containing the points of the polygon.|  
**colour:** The colour 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(Window cWindow, string text, Vector2 position, Color colour, bool localise = true, bool snapToScreen = false)

**Parameters:**

**cWindow:** The window 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.  
**colour:** The Color of the text. **Warning: Outline colour, 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 15: 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(window, "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. Loading Textures**

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

Texture texture = new Texture(512, 512);

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

texture.Path = “Content/image.png”;

texture.Load(cWindow);

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

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(cWindow);

**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.

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 (Texture class, inherits from Renderable)**

**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 colour is used in Lightning) four times the size of the image’s width.

**Methods**

**Texture(Window cWindow, float sizeX, float sizeY):** Creates a new texture for the Window **cWindow**, and then sets the **Size** to (**sizeX, sizeY**­)

**Load(Window cWindow):** 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 colour 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 colour, bool unlockNow = false)** Sets the pixel **x,y** of the texture to the Colour **colour.** 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(Window cWindow):** Draws the texture to the Window **cWindow.** 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(Window cWindow, Vector2 frameSize, Vector2 textureCount):** Creates a texture atlas for the window **cWindow**. 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(cWindow, 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

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

AnimatedTexture animatedTexture = new AnimatedTexture(cWindow, 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.

**10.3. Animation Blending**

**11. Lighting**

**12. Particle Effects**

**13. UI**

**14. Text Rendering & Font Management**

**15. Localisation**

**16. Audio**

**17. System Information & Enforcing System Requirements**

**18. Packaging**

**19. Debugging**

**20. Advanced Usage**

**21. 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 WindowSettings was passed to Window::Start.

**Error Code 8**

An SDL error occurred during window creation in Window::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:\windows\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. Add one.

**Error Code 44**

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

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

**Error Code 118**

Attempted to move a window 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 Window when using the Scene Manager. Set the DontUseSceneManager GlobalSetting to true if you wish to do thi.

**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 window before initialising the engine. Call Lightning::Init before using any Lightning methods.

**Error Code 57005 (0xDEAD)**

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