Programmers Guide

1. License

BSD 2-Clause "Simplified" License

ToyGame

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

2. About ToyGame

ToyGame can be seen as a streamlined version of NGE2.

It serves as a foundational framework, encompassing essential functionalities such as window creation, input handling for keyboard, mouse, and gamepad, rendering, and audio playback.

Borrowing code directly from NGE2, it provides a user-friendly starting point for graphical applications, particularly video games.
ToyGame leverages Branimir Karadzic's bgfx library for rendering, supporting both DirectX11/12 and Vulkan. Additionally, it utilizes Jari Komppa's SoLoud for audio processing and GLFW3 for window management.

3. Framework

3.1 Types

The header file ***<ToyGame/Types.hpp>*** defines all types and math structs:

Name	Type
real6	64-bit floating point number
real3	32-bit floating point number
decimal	long double
int [8;64]	Integer
uint	unsigned Integer
strg	std::string
ulong	unsigned long
ushort	unsigned short
sshort	signed short
vec4	4-dimensional vector
vec3	3-dimensional vector
vec2	2-dimensional vector
mat4	matrix 4x4
mat3	matrix 3x3
mat2	matrix 2x2
quat	quaternion
vec4i	4-dimensional int vector
vec3i	3-dimensional int vector
vec2i	2-dimensional int vector
vec4d	4-dimensional double vector
vec3d	3-dimensional double vector
vec2d	2-dimensional double vector
mat4d	double matrix 4x4
mat3d	double matrix 3x3
mat2d	double matrix 2x2
quatd	double quaternion

This header file should always be included first in every header.

The *eDirection* struct is for handling 3D rotation:

```
struct eDirection
{
    real3 mYaw;
    real3 mPitch;
    real3 mRoll;
};
```

The *eTransformation2D* struct is for handling 2D Position, scale, rotation, and rotation pivot:

```
struct eTransformation2D
{
    vec2    mPosition;
    vec2    mScale;
    real3   mRotation;
    vec2    mRotationPivot;
};
```

The *eTransformation3D* struct is for handling position, rotation, and scale in a 3D space:

```
struct eTransformation3D
{
    vec3 mPosition;                 //1 unit = 1 centimeter
    vec3 mRotation;                 //In degrees
    vec3 mScale = vec3(1.0f);       //1 unit = 1 centimeter
};
```

Your scene does **not** have to follow the unit scaling displayed here;
some functions for 3D rendering expect the rotation to be in degrees though.

3.2 String

This custom string type can be included with ***<ToyGame/String.hpp>.***

It's a very basic string type, utilizing the standard *std* string type.
It is still better to work with this custom string type though, because working with *std::string* directly can get messy pretty fast.

Some special functions are **Set(),** for both a *c string* and *std::string*, as well as some other misc functions like **AddLine(), Clear(), Get()** and **GetStrg(), Add()**, **Compare()**, **Contains**(), **Size(), Length(), IsEmpty()** and basic operations.

3.3 eTask

The *eTask* class can be included with ***<ToyGame/Task.hpp>.***
You can interpret an *eTask* as something that controls and manages the main application loop.
When creating a new task, you are required to create four functions:

The **Initialize()** function should include all your initialization code like window creation and file loading. You should also call **SuperInit()** so that a *time*, *window*, *gamepad* and *audio* module are being initialized. This function returns a *Boolean*.

The **Update()** function includes your main application loop. When utilizing the time module you should call **mTime->Update().**

The **Render()** function includes all your rendering code.

And at last, the **Unload()** function is being called once the application quits. Make sure to call **SuperUnload()** to release all loaded modules.

3.4 eTime

To use this class, include ***<ToyGame/Time.hpp>.***

This class handles delta time.

You can receive the delta time by calling **DeltaTime()**, which returns a *real6*.

You can also manipulate your applications speed by pointing to a custom speed variable. When your application first starts you should create a speed variable and set it with **SetSpeed()**. The application will cause a runtime error if that value is a *nullptr*.

3.5 EDR

EDR can be included with ***<ToyGame/EDR.hpp>.***

The primary purpose of the *EDR* namespace is to handle file interactions.
This simple file format is like the ini file format.
You can look, change, and write values of tokens, or group them together.

The *EDR* namespace also includes functions for handling ZIP files.

You can create a *zip** object and open a archive with **OpenZIPArchive().**
Any changes made to the archive only apply once **CloseZIPArchive()** is called.

Have a look at the header file of *EDR* to learn more about it.

3.6 eSystem

To use *eSystem* simply include ***<ToyGame/System.hpp>.***

The *eSystem* namespace includes functions for receiving system information.

```
namespace eSystem
{
    //CPU
    const uint GetProcessorCount();

    //RAM
    const uint64 GetRAMsize();
    const uint64 GetAvailableRAMsize();
    const uint64 GetVirtualRAMsize();
    const uint64 GetAvailableVirtualRAMsize();

    //Drive
    const uint64 GetSpaceOnDisk();
    const uint64 GetAvailableSpaceOnDisk();
};
```

3.7 eConsole

To use *eConsole* simply include ***<ToyGame/Console.hpp>.***

The *eConsole* class is an easy way to debug your application. You can send and receive messages and logs from it. You are also able to write the console log to a file called "CRASH.log".

```
class eConsole
{
public:
    static void Print(eString msg);
    static void PrintLine(eString msg);
    static void PrintLog(eString head, eString msg);
    static void Clear();
    static eString Get();
    static void WriteToDisk();
```

3.8 eWindow

To use *eWindow* simply include ***<ToyGame/Window.hpp>.***

The *eWindow* class manages a single window. You can resize it, set its resolution, aspect ratio, switch to Fullscreen, auto-size the window and even display a message box.

Each *eWindow* instance will hold their own *keyboard* and *mouse* module.

Have a look at the header file for more info.

4. Input

4.1 Keyboard

To use *eKeyboard* simply include ***<ToyGame/Keyboard.hpp>.***

The *eKeyboard* class handles key events for each individual *eWindow* instance.

You can check for a specific key input with the predefined keywords beginning with *NGE_INPUT_*.

Have a look at the header file for more info.

4.2 Mouse

To use *eMouse* simply include ***<ToyGame/Mouse.hpp>.***

The *eMouse* class handles mouse events for each individual *eWindow* instance.

For smooth mouse motion you should call **CheckRawMouseMotionSupport().**

Have a look at the header file for more info.

4.3 Gamepad

To use *eGamepad* simply include ***<ToyGame/Gamepad.hpp>.***

The *eGamepad* class handles gamepad events for each individual *eTask* instance.

Each gamepad has its own *ulong port*.

You can check if a gamepad is connected with **IsConnected().**

By default, an *Xbox One* controller is expected. You create custom gamepad layouts for different controller types by modifying the **eGamepadLayout** struct.

You can also load custom layouts saved in a "*gamepad.rd*" file using the *EDR* file format. Simply call **LoadLayout().**

Have a look at the header file for more info.

5. Audio

To use *eAudio* simply include ***<ToyGame/Audio.hpp>.***

The **eAudioSrc** struct holds your audio file data. You can load and manage your individual audio files using the **eAudio** module created by each *eTask* instance.

Calling **GetCore()** gives you access to the **eAudioCore** instance which gives you access to all *SoLoud* features.

Have a look at the header file for more info.

6. Graphics

6.1 eGraphics

To use *eGraphics* simply include ***<ToyGame/Graphics.hpp>.***

This class is created with every *eRenderer* instance.

You can draw basic 2D geometry with **Draw2D().**

With **Draw3D()** you can draw simple 3D primitives.

To draw a simple *Billboard* you can call **DrawBillboard().**

With the function **CompileShader()** you can compile a shader on the fly.

You can also print debug text with **Printf().**

To initialize *dear ImGui* simply call **InitDI().**

You can then render basic Gui windows using the *eDI* namespace.

To safely delete a bgfx handle simply **call eGraphics::Destroy().**

Have a look at the header file for more info.

6.2 eTexture

To use *eTexture* simply include ***<ToyGame/Texture.hpp>.***

The *eTexture* class is used for loading and managing a texture file.
You can adjust the textures quality by setting the global texture quality to either a lower, or higher value. If ***mSolidQuality*** is set to true, then this quality check will be ignored.

You can either load a texture from file or from memory. When loading a texture from memory you should use the **eTextureData** struct to define the textures properties:

```
struct eTextureData
{
    unsigned char*  mData;
    eString         mName;
    vec2i           mSize;
    int             mNbComponents;
};
```

6.3 eShader

To use *eShader* simply include ***<ToyGame/Shader.hpp>.***

The *eShader* class is used for loading and managing a shader file.

Each *uniform* must be initialized before being used. You can initialize a *uniform* with **InitUniform()**. The following uniform types are supported: **Sampler**, **Vec4**, **Mat3**, **Mat4**.

Have a look at the header file for more info.

6.3.1 Writing shaders

The shader language of bgfx is like GLSL.

All shaders must be in the *development/shaders/* folder.

The folder, vertex and fragment shader must be named all the same, with the vertex shader having the extension *.vs* and the fragment shader having the extension *.fs.*

Your input and output variables must be declared inside *varying.def.sc*.

You can simply copy one of the existing shaders as a starting point or modify an existing one. Please note that if your going to use the **Draw2D()** function that you have to use the predefined **eQuadVertex** structure. Your input texture file also must be named **s_texColor**.

You can of course write your own rendering code and don't have to rely on the **Draw2D()** function.

6.3.2 Compiling shaders

The *eGraphics* class includes a function called **CompileShader().**
The argument is the name of the shader you want to load. This is why the shaders folder, vertex and fragment name must be the same.
This function returns true if the compilation was successful.

By default, it will compile the shader for booth spirv (Vulkan) and HLSL (D3D).
For Direct3D it will use the shader model 5.

Once compilation is done the binaries can be found in the folders *shaders/d3d/* and *shaders/spirv/*.

6.4 Misc

6.4.1 eRenderAPI

To use *eRenderAPI* simply include ***<ToyGame/RenderAPI.hpp>.***

This is a simple enum used by *eGraphics* to define the target rendering API you want to use. The following rendering APIs are supported: **Direct3D11**, **Direct3D12** and **Vulkan**.

You can of course modify the source code of **ToyGame** to utilize different APIs.
This is possible thanks to bgfx supporting a wide variety of different rendering APIs.

6.4.2 eQuality

To use *eQuality* simply include ***<ToyGame/Quality.hpp>.***

This is used to define the *texture* quality when loading a *texture*. You don't have to utilize this method for other aspects of your application, it does however give you a continuity and it is very convenient to use.

6.5 Layer

To use *eLayer* simply include ***<ToyGame/Layer.hpp>.***

A layer can be seen as a drawing canvas. You draw your scene into a layer, and the layer manages the rest, like resizing the canvas or managing a framebuffer.

There are a bunch of important properties for a layer that define its behavior:

The **mViewId** is the layers order. A higher value will cause the layer to be rendered on top of others. It also means it will be drawn last.

The **mPosition** defines the canvases position on your window.

The **mResolution** defines the rendering resolution and canvas size.

The **mAspectRatio** defines the rendering aspect ratio.

If **mTopMost** is set to true, then it will always render on top of every other layer.

The **mClearColor** defines the layers background color.

If **mTransparent** is set to true, then the layers background will be transparent showing the layer behind it.

If **mUpdateSize** is set to true, then the canvas will automatically be resized according to the window size.

mName is the layers name.

mIs2D should be set to true if you only intend to render 2D sprites.

If **mDepthOnly** is set to true, then there will be only a depth buffer rendered. This is useful for shadow mapping and other things that only require a depth map with no color buffer.

When you call the constructor of *eLayer* you can chose to create a framebuffer which you can then access with **GetFramebuffer().**

6.6 Viewports

6.6.1 Viewport2D

To use *eViewport2D* simply include ***<ToyGame/Viewport2D.hpp>.***

A viewport can be seen as a virtual camera. Your scene is being rendered from its perspective.
A 2D viewport is used for rendering 2D scenes. You can set its position and its scale.

Have a look at the header file for more info.

6.6.2 Viewport3D

To use *eViewport3D* simply include ***<ToyGame/Viewport3D.hpp>.***

A 3D viewport is used to render a 3D scene from its perspective.
By default, it utilizes a first person – free camera type movement and view.
By calling **MouseRotate()** you can rotate the viewport with the mouse.
There is also a simple orbit rotation mode, but when using this mode it does not return a correct front vector.

Have a look at the header file for more info.

6.7 eRenderer

To use *eRenderer* simply include ***<ToyGame/Renderer.hpp>.***

The *eRenderer* class manages *layers* and *shaders* for you. It also creates an *eGraphics* module so all you must do is create a simple *eRenderer* derived.

With **PushLayer()** you can create a new layer; with **PushShader()** you can create a new shader.

Have a look at the header file for more info.

6.8 RenderDoc

RenderDoc is a very useful tool for debugging your rendering code.
By simply pressing *F11* when in D3D mode you can create a frame capture which is in the *temp/* folder.

I highly recommend working with *RenderDoc* as it can help you find issues with your code.