Rising Tide : Engine Specification 2018-2019 (v1.5.0)

Created By The Rising Tide Team

December 5th, 2018

Abstract

This document is a very tentative overview of the engine by the Rising Tide team.

Overview

This document describes the API design of the Engine as a whole not on any specifics. Typically you will need to add more functionality than what is described here but this is the baseline so that all engine Modules can interact with each-other in a sane way. The member variables on any class is a suggestion while the **member functions are a requirement**. If you are contributing to the Engine layer of this project you must follow the Style Guide

---

General Information

Dependencies

Library	Explanation
C++ STL (C++11, C++14, C++17)	Provides template containers so that we don't need to reinvent the wheel.
C Runtime Library	Provides a good set of utility functions such as 'cos', and 'sin'.
glfw (w/ glad)	Provides windowing (w/ OpenGL context creation) and input to our game.
glm	Provides a very optimized and stable set of vector / matrix math operations.
OpenGL v4.4	Provides a way to interface with the GPU Pipeline.
FMOD	Provides audio output and other sound utilities.
DEAR ImGui	Provides an easy way to create GUIs for in engine editing of our game.
stb_image	Provides 2D asset loading utilities for use with OpenGL.
FreeType	Provides a way to render fonts into our game window.
SOL2 (w/ Lua v5.3.5)	Provides a clean, modern C++ way to interface with the Lua scripting language.

Engine Overview / Ideals

- Cmake 3.11 used for our build system.

  - "project/CmakeLists.txt"
- GitHub used for source control
  - engine/*
  - game/*
- Visual Studio for compiling and debugging although not strictly required by our setup.
  - (We also have a twin Qt Creator project for those who prefer that IDE)
- Clangformat is used for enforcing the styleguide.

Engine Overview / Ideals

- The Engine is Based and ECS model with dense arrays of data (components) to be processed by systems each frame.

Engine Document

Engine Sub-Systems

- Debug
- Editor
- Art Pipeline
- Asset IO
  - Area
  - File
  - Assets
  - JsonParser]
  - JsonValue
  - Resource
  - TileMap
- Animation
  - Animation
  - AnimationFrame
  - AnimationSet
- Artificial Intelligence
  - ActionList
  - IActionNode
- Audio
  - AudioStream
  - AudioSoundFx
  - audio_manip
- Core
  - Engine
  - IGameState
- Data Structures
  - Vector
  - HashTable
  - ForwardLinkedList
  - String
  - Variant
  - Any
  - SparseArray
- ECS

Asset IO

- The Asset System uses a reference count based system so that assets can be unloaded a reloaded pretty gracefully.
- Everything is handed out through an [AssetHandle][] that can be used exactly the same as a pointer to the specified resoruce.

TileMap (class)

- This class represents a grid of tiles that makes up an Area graphically.

Area (class)

- This is our main "Level" representation and what the Mjolnir Editor is designed to help create. It contains a list of all the active entities and components and is what is save to disk.
- Member Functions
  - `TileMap&  tilemap();`
  - `bool      load(const String& path);`
  - `bool      save(const String& path);`
    - Save only *really* needs to be implemented in the editor side most-likely but maybe this function is used for player save data?
  - `void      pushBackgroundTrack(AudioStream& track);`
  - `void      crossFadeTrack(unsigned old_track, AudioStream& new_track);`
  - `void      popBackgroundTrack();`

File (class)

- This is a utility wrapper around the "std::iostream" object allowing for finer control / utilities for writing out binary files in an Endian independent manner.

Assets (class)

- The Files have to come from somewhere so this class just manages the memory associated with Files. There should be a maximum of 16 Files opened at once.
- This class also manages the memory associated with Resources and makes sure a certain texture, or sound is only loaded once.
- All assets are referencce count thus unloaded when nothign references that asset, but if requested the engine will attempt to load it on demand.
- Member Functions
  - `File*           makeFile(const String& path);`
  - `Texture2D*      makeTexture(const String& name, std::uint32_t width, std::uint32_t height);`
  - `Texture2D*      loadTexture(const String& path);`
  - `RenderMaterial* makeMaterial(const String& name);`
  - `RenderMaterial* loadMaterial(const String& path);`
  - `...` pretty much the same pattern for the rest of the Resources types ...

JsonParser (class)

- The good old Json Parser. This class allows us to write out and read in Json formatted text documents for save data and other configuration.

JsonValue (class)

- This class is specialized variant that contains:
  - HashMap<String, JsonValue>
  - Vector<JsonValue>
  - double
  - String
  - bool

AssetHandle (class)

- This class is a tagged union that will contain the data to various assets used by each respective subsystem.
- AnimationSet <- Animation
- Texture2D <- Graphics
- AudioStream <- Audio
- AudioSoundFx <- Audio
- Font <- User Interface
- Script <- Scripting
- [RenderMaterial][] <- Graphics

Animation

- This module is a very simple Sprite / Frame based animation framework. It supports variable times frames.
- *(v1.1.5):* Now contains an easing module to help with more procedurally generated animations.

AnimationFrame (class)

- Just a pairing of an index and how long the frame should last.

Animation (typedef)

- Just a simple list of animation frames.
- Vector<AnimationFrame>

AnimationSet (class)

- This class contains a list of animations associated with a certain texture.
- The data needed are:
  - The UV rectangles of the each frame.
  - The order of the frames for each animation. (Probably contained in a map)

AnimationComponent (class)

- This is the class actually used as a component for the ECS module. It represents an actual instance of an animation while sharing the important data that is not directly related to each particular entity.
- The data needed are:
  - The shared Animation Set data.
  - The current animation that is wanting to be played.
  - The current index of the animation to be used.
  - The current time left for the current frame.

Ease<T> (class)

- T value() const;
- T update();

Artificial Intelligence

- This module is for giving some external structure to the behavior system as to reduce code duplications.
- The bulk of the system comes from making specific nodes for the type of game we're making so the engine backend is *very* light with only two classes in this specification.

IActionNode (class)

- This class is a single unit of state update that an object would undergo.
- These are contained in an ActionList to compose more conplex behaviors by aggregating together a string of nodes.
  - EX: 'Move', 'WaitNSeconds', 'FollowPlayer', 'FindPath', 'PlaySound'

ActionList (class)

- This class is a 'pre-compiled' list of nodes that are ran in the same order every time.
- If you want an entity to change its set of actions it takes then just multiple action lists and switch which one is currently the active one.
  - EX: Have an action list for each stage of a boss battle.

Audio

- This is a simple module that is a simple wrapper around FMOD to provide audio output to our engine.
- This may be replaced by a custom audio engine at some point but is not a proprity.

AudioInfo (class)

- This class has public members for creating an an AudioStream or AudioSoundFx from memory.

AudioStream (class)

- This class is made for cases where larger audio files such as background music as the file is streamed from the hard-drive rather than having it loaded completely into ram.
- Member Functions
  - `void play(unsigned int start_pos_in_ms);`
  - `void stop(); // Cuts the music immediately`
  - `void setVolume(float value);`
  - `bool loadFromFile(const String& path);`
  - `bool loadFromData(const AudioInfo& data);`

AudioSoundFx (class)

- This class is for smaller sound effect sounds that can be easily kept in memory and is very short lived.
- Member Functions
  - `void play();`
  - `void randomizePitch();`
    - Allows the sound to be played at random pitches.
  - `void randomizeGain();`
    - Allows the sound to be played at random volumes.
  - `void makeInteresting();`
    - Calls 'randomizePitch' and 'randomizeGain'.
  - `bool loadFromData(const AudioInfo& data);`
  - `void setVolume(float value);`
  - `void matchToTrack(const AudioStream& track);`
  - `void addReverb(int amount);`

audio_manip (namespace)

- This namespace has functions that are shared between streams and soundfx

```
  // Handles fading the audio in or out depending on the parameters.
template<typename T>
void fade(T&           audio_source,
          unsigned int fade_time_in_ms,
          float        start_volume,
          float        end_volume,
          unsigned int start_pos);

template<typename T>
void fade_volume(T&           audio_source,
                 unsigned int fade_time_in_ms,
                 float        start_volume,
                 float        end_volume);

  // Adds a "tape stop" effect to the music, which essentially pitches the song down
  // and fades out at the same time.
  // Use this when player dies?
template<typename T>
void tape_stop(T&           audio_source,
```

```
                unsigned int fade_time_in_ms);

template<typename T>
T low_pass(T& audio_source, int frequency);

template<typename T>
T high_pass(T& audio_source, int frequency);
```

API Example

```
  // NOTE: This will enqueue an Ease into the current Area to be updated by the EasingSystem.
audio_manip::fade<AudioStream>(engine.currentArea()->backgroundTracks().at(1),
                                500,  // 0.5s
                                0.0f, // completely muted
                                1.0f, // full volume
                                0     // 0.0s
                               );
```

Core

- This is a basic module consolidates all the other modules of the engine together and helps coordinate a cohesive result.
- It also contains the GameState Manager along with the definition of a Game State.
  - The gamestates are organized as a stack with a push and pop interface.
  - A gamestate can decide if it wants to propagate events down to the lower gamestates by just blocking the event or not.
    - EX: This allows for a very flexible design by maybe for an in-game editor that can be enabled or disabled by just pushing a new editor gamestate onto the stack.

Engine (class)

- The Engine class serves 3 main purposes:
  - This class is what's instantiated from main when the program started up. It's job is to coordinate all the subsystems and making sure everything gets called in the correct order.
  - This class manages the IGameState stack and makes sure to call all the necessary virtual methods on the gamestate to make sure restarting works correctly.
  - This class also manages the the total memory used the the application as a whole.

IGameState (interface)

- This is an interface for what makes a game state. It provides some utilities for manipulating the gamestate stack.

Data Structures

Vector (typedef)

- std::vector<T>

HashTable (class)

- This data-strucuture is for storing key-value mappings can be used in an almost identical way as "std::unnordered_map".
- This is implemented using an open-addressing, linear probing technique.

ForwardLinkedList (typedef)

- std::forward_list<T>

String (typedef)

- std::string

Variant (class)

- This class represents a tagged union of a limited set of values you specify.
- An example of the use is the MaterialVariable typedef.
- The size of this class is the sizeof the largest class type in the template argument list + sizeof(std::size_t).

Any (class)

- This class is similar to Variant except it can hold any value. This comes at a performance cost (dynamic memory allocation + virtual function call overhead) so Variant is prefered.
- The size of this class is the sizeof(void*) + sizeof(void*)*. (This is because it used dynamic memory allocation to support any type)

SparseArray (class)

- The SparseArray is used for fast addition and removal of elements while keeping a cache coherent array of objects.
- Made for faster (frequent) Insert(s) and Remove(s) relative to Vector while keeping a cache coherent dense array.

ECS

Entity (class)

- This is the GameObject representation of the Engine that contains handles to each of it's components.
- All Entities are owned by a parent Area along with it's component data.

Behavior (class)

-

Graphics

- The Graphics backend we will be uisng is OpenGL 3.2.
- The Graphics Module will be largely based off of the concept of materials that contain whole pipeline states for easy batch management.
- The Renderer should be separated into 3 Layers of abstraction the highest layer being what the rest of the Modules will be interfacting with.
  - High Level
    - Will contain code that will be used to interface with the Graphics Module from other Engine Modules.
    - This layer will be the most abstracted from the hardware and will use concepts such as Sprites and Materials.
  - Mid Level
    - Will contain code that will make it easier for the graphics programmers to work on adding new functionality to the engine.
    - This layer is based off of the concept of DrawItems and this is where we do all of the culling and sorting.
  - Low Level
    - Will contain code that is just a thin wrapper around the underlying graphics API (OpenGL, DirectX, etc...) that we will be using.
    - This layer manages the GPU memory making sure everything is layed out in a way so that efficient rendering is possible.

High Level Graphics

BlendMode (enum class)

- These are the Blendmodes the Engine will support. 'NONE' is an optimization since blending takes GPU cycles otherwise a normal 'ALPHA_BLENDING' works. 'ADDITIVE_BLENDING' is typically used for the way lights will be applies to the scene.
  - `NONE`
  - `ALPHA_BLENDING`
  - `ADDITIVE_BLENDING`

DrawMode (enum class)

- Supporting all the DrawModes because it's such a trivial feature to have and may be useful for debug drawing type contexts.
  - `POINT_LIST` `(GL_POINTS)`
  - `LINE_LIST` `(GL_LINES)`
  - `LINE_STRIP` `(GL_LINE_STRIP)`
  - `LINE_LOOP` `(GL_LINE_LOOP)`
  - `TRIANGLE_LIST` `(GL_TRIANGLES)`
  - `TRIANGLE_STRIP` `(GL_TRIANGLE_STRIP)`
  - `TRIANGLE_FAN` `(GL_TRIANGLE_FAN)`

MaterialVariable (typedef)

- Variant<bool, float, int, Vec2, Vec3, Vec4, Mat2x2, Mat3x3, Mat4x4, std::pair<int, Texture2D*»

RenderMaterial (class)

- The material class contains the WHOLE pipeline state of the renderer, this makes sorting for batch rendering super easy as you just have to sort based of the pointers of the material each DrawItem has.
- Member Functions
  - `BlendMode setBlendMode(BlendMode mode);`
  - `DrawMode setDrawMode(DrawMode mode);`
  - `MaterialVariable value(const char* value_name);`
  - `void setValue(const char* value_name, T value);`
  - `Shader* shader(void);`
  - `void setShader(Shader* shader);`

Color (class)

- A Color is used in the Vertex class. It takes up only four bytes for all channel together by making use of bit shifts and masks for a very efficient memory layout. This is much better than using four floats as the representation of colors (although we do lose some precision but all the color depth that matters comes from textures anyway).
- Member Variables
  - `std::uint8_t m_R;`
  - `std::uint8_t m_G;`
  - `std::uint8_t m_B;`
  - `std::uint8_t m_A;`
- Member Functions
  - `std::uint8_t r() const;`
  - `std::uint8_t g() const;`
  - `std::uint8_t b() const;`
  - `std::uint8_t a() const;`
  - `void setR(const std::uint8_t value);`
  - `void setG(const std::uint8_t value);`
  - `void setB(const std::uint8_t value);`
  - `void setA(const std::uint8_t value);`

DirectionalLight (class)

- This light is used for things such as the Sun as it just has a direction on which to affect objects.

PointLight (class)

- This light is used for when to want light a specific part of the scene in a circular shape.

SpotLight (class)

- This light is used for lighting a specific part of the scene in a cone shaped manner (by using an angle to decide the area of effect).

Vertex (class)

- This class is the Uber vertex class used for the drawing of any graphics on screen.

Transform (class)

- Member Functions (You should have more functions than this for manipulating the transform)
  - `inline float rotation() const { return m_Rotation; }`
  - `inline const Vec3& position() const { return m_Position; }`
  - `inline const Vec3& scale() const { return m_Scale; }`
  - `inline const Vec3& size() const { return m_Size; }`
  - `inline void setPosition(const Vec3& value) { m_Position = value; m_IsDirty = true; }`
  - `inline void setScale(const Vec3& value) { m_Scale = value; m_IsDirty = true; }`
  - `inline void setSize(const Vec3& value) { m_Size = value; m_IsDirty = true; }`
  - `inline bool isDirty() const { return m_IsDirty; }`
  - `const Mat4x4& matrix() const;`

SpriteComponent (class)

- This is the main component used to display a flat 2D image on screen.

Low Level Graphics

ShaderProgram

FrameBuffer

Texture2D (class)

- A texture is an image where the data lives on the GPU, pretty obvious what this class is.
- Member Functions
  - `std::int32_t handle() const;`
  - `std::uint32_t width() const;`
  - `std::uint32_t height() const;`

BufferUsage (enum class)

- (0x0) READBACK - Buffer is used to store data from device operations such as screenshots, occlusion depth buffer *readback*, etc.
- (0x1) UPLOAD - Buffer is used to upload dynamic geometry (textures, uniforms, etc) as a staging buffer for *static* **DEVICE_LOCAL** buffers.
- (0x2) DEVICE_LOCAL - Buffer is used to store data that needs to be efficiently fetched by the device: rendertargets, textures, (static) buffers.

BufferType (enum class, flags)

- (0x1) VERTEX - This type of buffer is used for storing vertex data for geometry.
- (0x2) INDEX - This type of buffer is used for storing indices for which to do an indexed draw.
- (0x4) UNIFORM - This type of buffer is used for setting constants / uniforms to the shader.
- (0x8) STORAGE - This type of buffer is used for storing SSBO content.

BufferCreateParams (class)

- BufferUsage usage;
- BufferType type;
- std::uint32_t size;

Buffer (class)

- std::uint32_t size() const;
- std::int32_t handle() const;
- BufferUsage usage() const;
- BufferType type() const;
- void* map();
- void unmap();

DrawItem API Example

- A **DrawItem** contains all of the state except for Depth / Stencil and Render Targets.

```
RenderMaterial* const material = engine.fileSystem().makeMaterial("CustomMaterial");
material->setValue("uTextureDiffuse",  std::make_pair<int, Texture2D*>(0, textureDiffuse));
material->setValue("uTextureSpecular", std::make_pair<int, Texture2D*>(1, textureSpecular));
material->setValue("uTextureNormal",   std::make_pair<int, Texture2D*>(2, textureNormal));
material->setBlend(BlendMode::ALPHA_BLENDING);
material->setShader(coolShader420);

  // NOTE: 'material' inherits whatever state 'default_material' defines.
RenderMaterial* const material_inheritance[] = {default_material, material};

DrawItem* const item = engine.renderer().makeDrawItem(material_inheritance, 2)
item->bindTransform(Mat4x4*);
item->bindBuffer(Buffer*);
item->draw(DrawMode, first_vertex, num_vertices);
item->drawIndexed(DrawMode, first_index, num_indices);
item->end();

  // ... later in the update loop ...
engine.graphics().enqueueDrawItem(item);

  // or alternative Immediate Mode / Command based API
engine.graphics().cmdBeginDrawItem(material_inheritance, 2);
engine.graphics().cmdPushTransform(Mat4x4*);
engine.graphics().cmdPushBuffer(Buffer*);
engine.graphics().cmdPushDraw(DrawMode, first_vertex, num_vertices);
engine.graphics().cmdPushDrawIndexed(DrawMode, first_index, num_indices);
engine.graphics().cmdEndDrawItem();

  // for when you want to draw the same thing but change something like the DrawMode
engine.graphics().cmdBeginClonedDrawItem(item);
engine.graphics().cmdPushDrawIndexed(DrawMode, first_index, num_indices);
engine.graphics().cmdEndDrawItem();
```

–> # Math - For (most of) the math module we are just going to use the glm library as that will lead

to the most efficient workflow. No need to write our own math library this time around. If we find any performance issues then maybe we can SIMD optimize the operations but I believe glm already makes use of SSE intrinsics.

Mat4x4 (typedef)

- glm::mat4

Vec4 (typedef)

- glm::vec4

Vec3 (typedef)

- glm::vec3

Vec2 (typedef)

- glm::vec2

Camera (class)

- This class is for managing the current state of the camera on a per Area basis.

User Interface

namesapce UI

- begin
- button
- checkBox
- sliderV
- sliderH
- end
- update(const Event&)

Physics

- This is a custom rigidbody 2D physics engine using the sperating axis theorem for collision detection.
- We are using an impulse based engine for collsion resolution.

RigidBodyComponent

- This is an object that can interact with the physics engine's forces.

ColliderComponent

- There are two types (Polygon and Circle)

Debug

- Drawing can be called from any update to draw some basic primatives.
- Logging has three different level which can be enabled or disabled with a "Logger::setDebugLevel"

```
// Drawing Utilities
void drawBox(const Rect2f& box, Color color);
void drawCircle(const Circle& circle, Color color);
void drawLine(const Vec2& p0, const Vec2& p1, Color color);
void drawVertices(const Vec2* vertices, std::size_t num_vertices, Color color);
void drawVertexArray(const VertexArray& vertex_array, Color color, unsigned int first_vertex, unsigne

// Debug Logging
#define dbg_push(...)     // Indent the messages for pretty formatting
#define dbg_verbose(...)  // General messages
#define dbg_warn(...)     // If there was an error but not fatal
#define dbg_error(...)    // If a fatal error has happened
#define dbg_pop(...)      // Deindent the messages for pretty formatting
```

Art Pipeline

Art Pipeline

Editor

```
// PROJECT GEMINI Editor Documentation
//
//     Mouse Drag  + Space      = Pan Camera
//     Right-Click + Drag       = Pan Camera
//     Mouse Drag  + Nothing    = Drag Entity
//     Mouse Drag  + Empty Area = Selection Rectangle
```

Cover Image

Engine Pseudo Code

```
  // NOTE: EXAMPLE_01: Adding an Entity to the current scene.
Area* const current_area = engine.currentArea();
```

```
  // If we even have a current scene, should be true but just know it can be nullptr.
if (current_area)
{
  Entity* const entity = current_area->addEntity("EntityName");

    // NOTE: EXAMPLE_02: Adding a component to an Entity.

  SpriteComponent& sprite = entity->add<SpriteComponent>(someTexture...);
  sprite.setColor(Color{100, 100, 100, 255});

    // NOTE: EXAMPLE_03: Grabbing an existing component from an Entity.
    // Is a pointer since there is a chance that it is null if the component isn't there.
  RigidBodyComponent* const rb = entity->get<RigidBodyComponent>();

    // This check is basically the "Entity::has<RigidBodyComponent>()" method.
  if (rb)
  {
    rb->setMass(20.0f);
  }

    // NOTE: EXAMPLE_04: Adding a behavior to an Entity.
    // These functions return a reference to the newly added behavior.
  entity->addBehavior<BehaviorType>(...args...);
  entity->addBehavior<LuaBehavior>("assets/scripts/behav_patrol_air.lua",
                                   { lua_params : "Data passing to lua" });

    // NOTE: EXAMPLE_05: Removing a component and behavior from an Entity.
  entity->remove<RigidBodyComponent>();
  entity->removeBehavior<BehaviorType>();

  // NOTE: EXAMPLE_06: Removing an Entity safely
  entity->deleteSekf();
}
```

Notes and Additional Resources

- Font Handling Tool