Rising Tide : Engine Specification 2018-2019 (v1.1.4)

Created By The Rising Tide Team Main Editor - Shareef R.

September 16, 2018

Abstract

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

Version Changes

v1.1.4 (Graphics Update)

- DEPRECATED: GameStateManager
  - The Engine now handles the responsibilities of the 'GameStateManager'
- Area class updates
  - DEPRECATED: AreaManager
    - The Engine now handles the responsibilities of the 'AreaManager'
  - Entities are now connected with a linked list rather than a vector to reduce the cost of allocation of many small entities.
  - Needs an allocator passed in so that it know where to retrieve memory for entities from.
- Added RenderMaterial* to Resources union of types.
- (Re)Added the Low Level Scripting Implementation back to the document.
  - Althoug the name is now Custom Scripting Language Spec. (Very Low Priority Project).
- Changed the title of this document a bit. ("Spec." -> "Specification") and ("2018" -> "2018-2019")
- Updated DrawItem API Example to reflect my more modern ideas on the Renderer's API design.
  - Started merging the "useless" notes at the bottom with the main document.
- Removed duplicate Camera definition in the Graphics module.
- Fixed minor errata in the Transform class;

v1.0.4 (Minor Fixes and Changes)

- Added a Version Changes section so that it is clear what has changed since the last version.
- Added a Engine Pseudo Code section because I know people don't like reading documentation.
- Added a cover image that may be removed in future versions or updated to the final logo image.
- Changed the way the header looks.

  - Different font and with more informaction such as when it was last updated.
  - Now has a subheader with the name of the author (The Team as a Whole and Shareef Raheem).
  - Added a basic abstract to this document just to fill out for information for pdf readers.
- The Engine class now has a more in-depth description to reflect the current implementation.
- Added a small description to Animation.
- Added a more full description and some member functions to the FileSystem class.
- Added more information to the Artificial Inteligence module.
  - ActionList and IActionNode now have valid descriptions.
- Reduced the size of the margins around the document to be more compact.
- Fixed String's description as it was defined as itself by mistake.
- Fixed some typos and grammar mistakes but I'm sure there is many more.

v1.0.3 (Initial Release)

- This is the initial public release of the unfinished API.

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 eachother 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.

Library	Explanation
DEAR ImGui	Provides an easy way to create GUIs for in engine editing of our game.
SOIL	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 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->addSpriteComponent(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->rigidbody();

    // This check is basically the "Entity::hasRigidbody()" 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->removeRigidbody();
  entity->removeBehaviorAt(0);
}
```

Engine Sub-Systems

- Asset IO
  - Area
  - File
  - FileSystem
  - JsonParser
  - JsonValue
  - Resource
  - TileMap
- Animation
  - Animation
  - AnimationFrame
  - AnimationSet
- Artificial Inteligence
  - ActionList
  - IActionNode
- Audio
  - AudioStream
  - AudioSoundFx
- Core
  - Engine
  - IGameState
- Data Structures
  - Vector
  - HashTable
  - ForwardLinkedList
  - String
  - Variant
  - Any
  - SparseArray
- ECS
  - Behavior
  - Component
  - Entity
  - ECSSystem
    - AudioSystem
    - BehaviorSystem
    - CollisionSystem
    - IECSSystem
    - ParticleSystem
    - PhysicsSystem
    - SpriteRenderer
    - TilemapRenderer
    - ResourceSystem
- Graphics
  - BlendMode
  - Color
  - CustomMaterial
  - DrawMode
  - MaterialVariable
  - RenderMaterial

  - Sprite
  - Texture2D
  - Transform
  - Vertex
- Math
  - Mat4x4
  - Mat4x4Stack
  - Vec4
  - Vec3
  - Vec2
  - Camera
  - Rect2T
  - Rect2f
- Memory Management
  - CAllocator
  - FreeListAllocator
  - IMemoryManager
  - LinearAllocator
  - PoolAllocator
  - ProxyAllocator
  - StackAllocator
- Meta Type Information
- Threading
  - Job
  - JobQueue
- User Interface // Out of Alphabetical order:
- Physics
  - Collision
    - SphereCollider
    - BoxCollider
    - PlaneCollider
  - RigidBody
  - Contact
  - ForceGenerator
  - Joint
- Debug
- Scripting
  - [LuaBehavior]
- Events

Asset IO

TileMap (class)

- This class represents a grid of tiles that makes up an Area graphically.
- Member Variables
  - `std::uint32_t  m_Width;`

- `std::uint32_t m_Height;`
- `std::array<TileLayer, 3> m_Layers;`
- `ResourceHandle m_Texture;`
- `BitSet8 m_IsLayerEnabled;`

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 Variables
  - `IMemoryManager& m_Allocator;`
  - `Camera m_Camera;`
  - `ComponentList m_ActiveComponents;`
  - `ComponentList m_InActiveComponents;`
  - `BehaviorList m_ActiveBehaviors;`
  - `BehaviorList m_InActiveBehaviors;`
  - `Entity* m_ActiveEntities;`
  - `Entity* m_InActiveEntities;`
  - `TileMap m_TileMap;`
- 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?

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.

FileSystem (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.
- 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 resource 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

Resource (class)

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

Animation

- This module is a very simple Sprite / Frame based animation framework. It supports variable times frames.

AnimationFrame (class)

- Member Variables
  - `std::uint32_t m_FrameIndex;`
  - `float         m_FrameTime;`

Animation (typedef)

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

AnimationSet (class)

- Member Variables
  - `HashMap<String, Animation> m_Animations;`

  - `Vector<Rec2f>  m_FrameRects;`
- 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.
- Member Variables (Components should have public members)
  - `AnimationSet* animation_set;`
  - `Animation*  current_animation;`
  - `std::uint32_t current_index;`
  - `float  time_left;`
- 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.

Artificial Inteligence

- 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.

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.

AudioSoundFx (class)

- This class is for smaller sound effect sounds that can be easily kept in ram and is very short lived.

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.

Graphics

- 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 Variables
  - `BlendMode m_BlendMode;`
  - `DrawMode m_DrawMode;`
  - `HashMap<const char*, MaterialVariable> m_Variables;`
  - `ShaderProgram* m_Shader;`
- 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)

- Member Variables
  - `Vec4 m_Position;`
  - `Color m_Color;`
  - `Vec2 m_UV;`
- Member Functions
  - `Vec4& position();`
  - `Color& color();`
  - `Vec2& uv();`

Transform (class)

- Member Variables
  - `Vec3 m_Position;`
  - `Vec3 m_Origin;`
  - `Vec3 m_Scale;`
  - `Vec3 m_Size;`
  - `Mat4x4 m_Matrix;`
  - `float m_Rotation;`
  - `mutable bool m_IsDirty;`
- 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;
```

Transform (Implementation Details)

```
const Mat4x4& Transform::matrix() const
{
  if (isDirty())
  {
    m_Matrix  = glm::translate((m_Parent) ? m_Parent->matrix() : Mat4x4(1.0f), -m_Origin);
    m_Matrix  = glm::scale(m_Matrix, m_Size * m_Scale);
    m_Matrix  = glm::rotate(m_Matrix, m_Rotation, Vec3(0.0f, 0.0f, 1.0f));
    m_Matrix  = glm::translate(m_Matrix, m_Position + m_Origin);
    m_IsDirty = false;
  }

  return m_Matrix;
}
```

SpriteComponent (class)

- Member Variables
  - `RenderMaterial* m_Material;`
  - `Rec2f           m_TextureRect; // The UVs of the Sprite for Animation`
  - `Transform       m_Transform;`
  - `Flags           = { HAS_ALPHA, FLIP_X, FLIP_Y }`
- Member Functions
  - TODO(Shareef): Write out getter and setters specification

Mid Level Graphics

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.renderer().enqueueDrawItem(item);

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

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

Threading

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

UIContext

- button
- checkBox

Notes and Additional Resources

- Font Handling Tool

Custom Scripting Language Spec. (Very Low Priority Project)

API

Low Level Implementation

- Bytecode Format
  - 32bits of awesome

```
0     5         14         23         32
[ooooo|aaaaaaaaa|bbbbbbbbb|ccccccccc]
[ooooo|aaaaaaaaa|bxbxbxbxbxbxbxbxbxb]
[ooooo|aaaaaaaaa|sBxbxbxbxbxbxbxbxbx]
opcode  =       0 -  32
rA      =       0 - 512
rB      =       0 - 512
rBx     =       0 - 262143
rsBx    = -131072 - 131071
rC      =       0 - 512
```

- Instruction Set (Max 32)
  - Load Instructions
    - LOAD_GLOBAL
      - rA = _G[rB]
    - LOAD_CONSTANT (Numbers or Strings)
      - rA = K[rB]
    - LOAD_BOOLEAN
      - rA = (Boolean)B
      - if (C) PC++
    - LOAD_NULL
      - rA... (rA + B) = null
    - LOAD_OBJECT_KEY
      - rA = rB[rC]
    - LOAD_ARRAY_INDEX
      - rA = rB[rC]
  - Store Instructions
    - STORE_GLOBAL
      - rB[rC] = rA
    - STORE_ARRAY_INDEX
      - rB[rC] = rA
    - STORE_OBJECT_KEY
      - rB[rC] = rA
    - MOVE
      - rA = rB
  - Boolean Logic Instructions
    - CMP_LE
    - CMP_LT
    - CMP_EQ
    - CMP_OR

        * CMP_AND
        * LOGICAL_NOT
        * CMP_GT = (LOGICAL_NOT + CMP_LE)
        * CMP_GE = (LOGICAL_NOT + CMP_LT)
    - Arithmatic Instructions
        * ADD
        * SUB
        * DIV
        * MULT
        * MODULO
        * POW
        * UNARY_MINUS
    - System Instructions
        * MAKE_OBJECT
        * RETURN
        * PUSH_PARAM
        * TAIL_CALL
        * CALL
        * JUMP
            · if ((rA & CPU_STATE) == CPU_STATE) PC += rsBx
        * GET_RETURN

RANDOM NOTES NOT THAT USEFUL FOR NOW

- Each draw item is a very compact structure, containing state IDs.
- XOR'ingtwo draw items creates a bitmask that highlights any changes.
- Masking out sections of that bitmask and comparing them to zero lets you quickly check if a state has changed since the previous draw item. ## IDEA : Use Lua as a DSL (Domain Specific Language) for defining 'Techniques' rather than hardcoding them
- Use sorting Keys
- But if Alpha blended: *~(u32)*distance as the sorting key
- Opaque can use a hybrid to do front to back. (mix original sort key and distance)
- Shaders
    - Shaders can indicate what slots it uses using a bit mask
        * EX ((1 << 0) | (1 << 2)) for using samplers 0 and 2

Vulkan

- Vulkan's Memory Model
    - Has two aspects "visibility" and "availibility"
        * HOST_COHERENT / vkFlushMappedMemoryRanges is all about visibility

        * Availibility is from waiting on resources using primitives such as: Events, Fences, and Semaphores.
            · The device would use "vkPipelineBarrier"s rather than fences (which is generally for the host)
- USE HOST_COHERENT or "vkFlushMappedMemoryRanges"
- vkQueueSubmit automatically creates a memory barrier between (properly flushed) host writes issued before vkQueueSubmit and any usage of those writes from the command buffers in the submit command (and of course, commands in later submit operations).
- All you have to do for "vkMapMemory" since it doesn't tell you if the device is using the data is to use a fence (the fence from the submission).
    – Since when you submit is the only time the device is going to use that mapped memory.

Buffer Usage Patterns

Life Time	Example	Desc
"Forever"	Level Geometry	> Should be loaded from behind a load screen since this is so rare. > Make these immutable since we wont be updating these. > Use a staging buffer so these can this can use GPU device local storage.
Long Lived	Character Models	
Transient	UI, Text	
Temporary	Particles (Streamed)	
Constants	Shader Uniforms	

- Long Lived
    – The average case so use the defaults for this.
    – Should probably use a staging buffer still.
- Temporary
    – Uses a dynamic buffer that is host visible.
    – Pretty much lives in RAM. (map and unmap)
    – OpenGL - DYNAMIC flag.
    – Constants