TDD

This document contains all of the technical design decisions and architecture documentation. First iterations and thoughts are documented in my developer notebook. Some deliverables and project scope has changed due to time constraints. This includes sacrificing the spike template for gradual implementation.

Tech, Tools and Resources

Tech and Tools

The project was scripted in C++ 17 using Visual Studio Community 2022.

UML's and charts are made with *www.Lucidchart.com*

Source control is handled using Git.

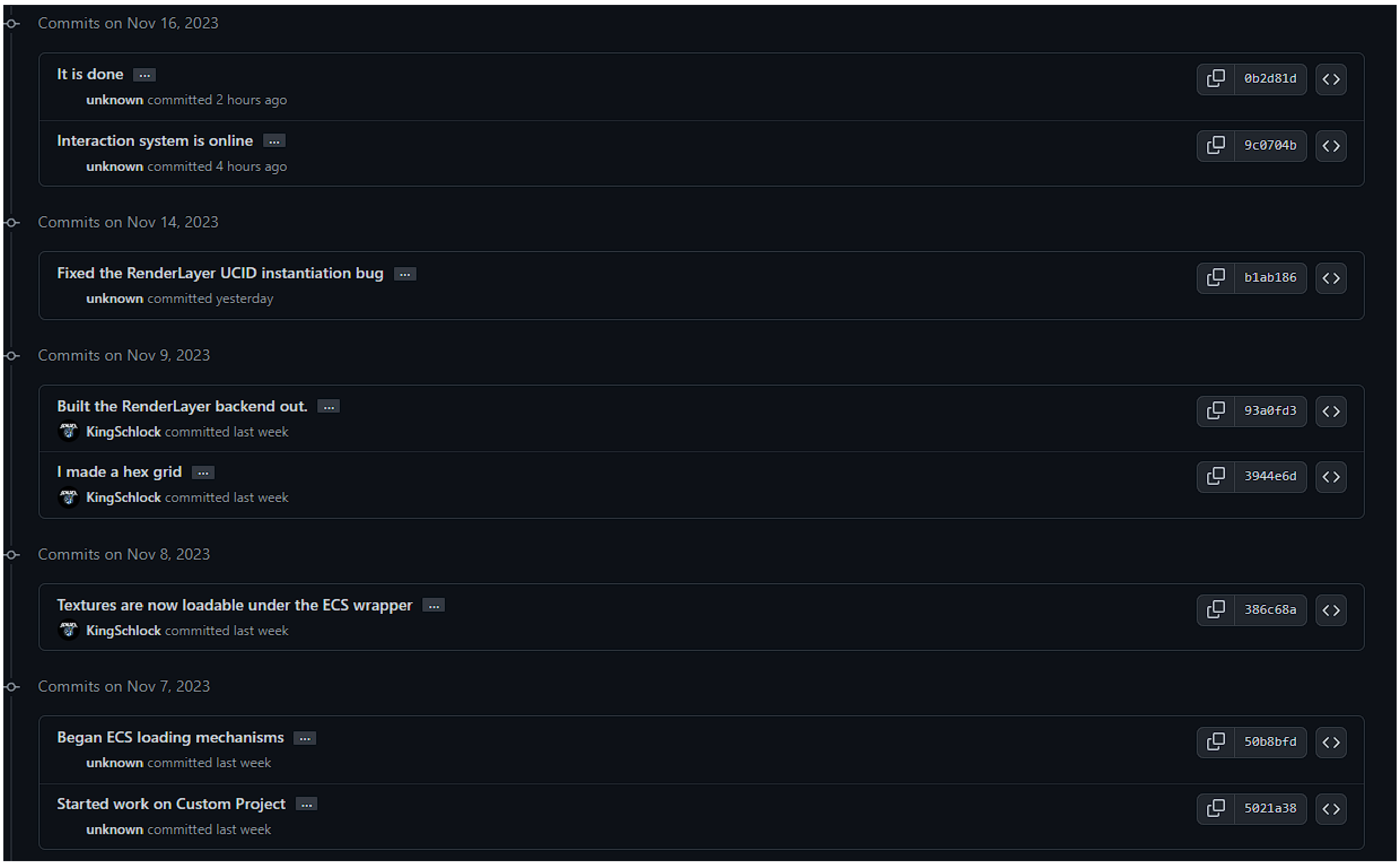
This project was made using:

- SDL2:
  https://github.com/libsdl-org/SDL
- SDL2_image
  https://github.com/libsdl-org/SDL_image
- SDL2_ttf libraries.
  https://github.com/libsdl-org/SDL_ttf

Resources

- My Previous Spikes:
  https://github.com/KingSchlock/COS30031-2023-103071494
- SDL Mouse Events Tutorial:
  https://www.youtube.com/results?search_query=sdl+mouse+events
- Lazy Foo: Texture Loading and Rendering:
  https://lazyfoo.net/tutorials/SDL/07_texture_loading_and_rendering/i
- SDL Image Tutorial:
  https://www.youtube.com/watch?v=ERGY54efC5k&t=552s
- Old school SDL RPG Game Making Series:
  https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx
- SDL Events Quick Intro:
  https://www.youtube.com/watch?v=6K5jBjFCzoY&t=217s

Commits & ReadMe



Commits on Nov 16, 2023

It is done
unknown committed 2 hours ago — 0b2d81d

Interaction system is online
unknown committed 4 hours ago — 9c0704b

Commits on Nov 14, 2023

Fixed the RenderLayer UCID instantiation bug
unknown committed yesterday — b1ab186

Commits on Nov 9, 2023

Built the RenderLayer backend out.
KingSchlock committed last week — 93a0fd3

I made a hex grid
KingSchlock committed last week — 3944e6d

Commits on Nov 8, 2023

Textures are now loadable under the ECS wrapper
KingSchlock committed last week — 386c68a

Commits on Nov 7, 2023

Began ECS loading mechanisms
unknown committed last week — 50b8bfd

Started work on Custom Project
unknown committed last week — 5021a38

ECS Architecture & Design Decisions

💡 The ECS (Entity Component System) was designed around two primary constraints:

1. Components must be purely data.
2. Components, Entities and Systems must be completely decoupled from one-another.

With these restrictions in mind I designed the ECS around a hash-table ID system, minimizing repetitive traversal of large component sets. Additionally, as systems are expected to be completely decoupled, they too will only be able to interact with their relevant components via the same ID system.

Component Types and Identification

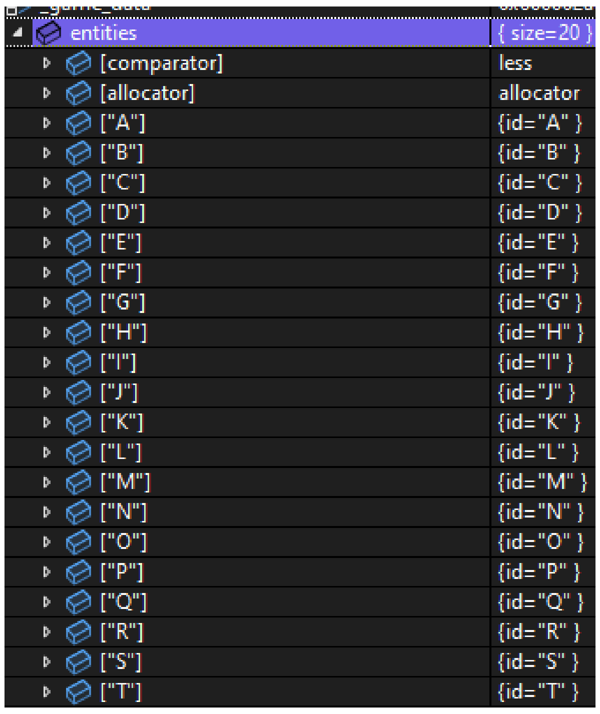
ComponentType	Data
`Component`	`component_id id;` `component_type type;`
`Transform`	`SDL_Rect bounds; SDL_Point origin;`
`Texture`	`SDL_Texture* texture; SDL_Rect bounds; bool render_this;`
`UILabel`	`SDL_Texture* label; SDL_Rect bounds; SDL_Color colour; label_content text; label_font font;`
`Interactable`	`bool is_interactable;`
`Movable`	`bool can_move; int radius;`
`Tag`	`EntityTag tag;`

For the identification system I chose to implement a string identifier. This was to completely dismiss issues which may arise from float imprecision and map addressing issues.

Each entity is given an `entity_id` which is just a string containing a set of up to 4 capital letters.
E.g. `0 = A, 26 = Z, 27 = AA`.

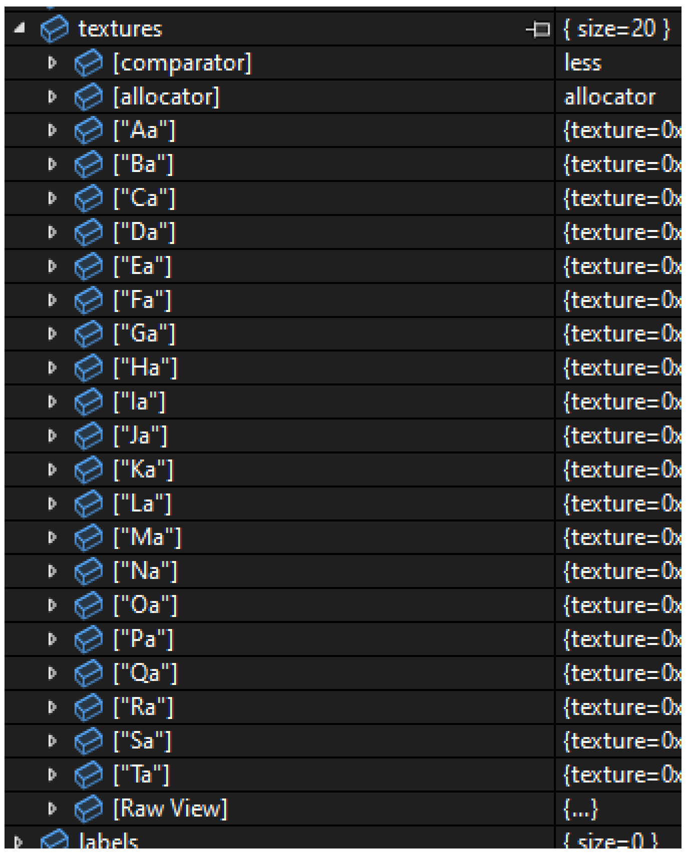
Each component is connected to it's entity via it's `component_id` which is a string containing a set of up to 4 lower case letters prefixed by it's owning `entity_id`.
E.g. First texture for entity 27 = "*AAa*"

This identification mechanism allows for different subsystems to understand which entity it's searching for based of the `entity_id` contained within a `component_id` and find each relevant component easily and quickly. Additionally, this system supports up to ~450,000 entity id's.



entities	{ size=20 }
[comparator]	less
[allocator]	allocator
["A"]	{id="A" }
["B"]	{id="B" }
["C"]	{id="C" }
["D"]	{id="D" }
["E"]	{id="E" }
["F"]	{id="F" }
["G"]	{id="G" }
["H"]	{id="H" }
["I"]	{id="I" }
["J"]	{id="J" }
["K"]	{id="K" }
["L"]	{id="L" }
["M"]	{id="M" }
["N"]	{id="N" }
["O"]	{id="O" }
["P"]	{id="P" }
["Q"]	{id="Q" }
["R"]	{id="R" }
["S"]	{id="S" }
["T"]	{id="T" }

Entity and Component ID's for 20 Entities.



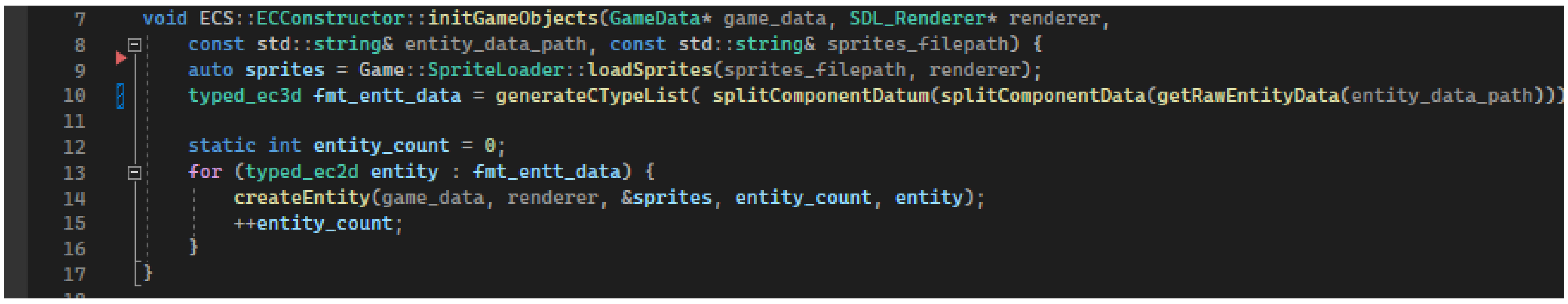
textures	{ size=20 }
[comparator]	less
[allocator]	allocator
["Aa"]	{texture=0x
["Ba"]	{texture=0x
["Ca"]	{texture=0x
["Da"]	{texture=0x
["Ea"]	{texture=0x
["Fa"]	{texture=0x
["Ga"]	{texture=0x
["Ha"]	{texture=0x
["Ia"]	{texture=0x
["Ja"]	{texture=0x
["Ka"]	{texture=0x
["La"]	{texture=0x
["Ma"]	{texture=0x
["Na"]	{texture=0x
["Oa"]	{texture=0x
["Pa"]	{texture=0x
["Qa"]	{texture=0x
["Ra"]	{texture=0x
["Sa"]	{texture=0x
["Ta"]	{texture=0x
[Raw View]	{...}
labels	{ size=0 }

TDD

Entity Construction

💡 Each component is constructed using the `ECConstructor` which is composed of relevant component “*Loader*” classes. The `ECConstructor` has the ability to read file data and format said data appropriately for the relevant “*Loader”*.

It can be inferred (if you stare for long enough) that the role of the `ECConstructor` is to run multiple passes through a set of data provided by a file. Each pass further refining and subsequently adding an extra dimension to the formatted set of entity component data.

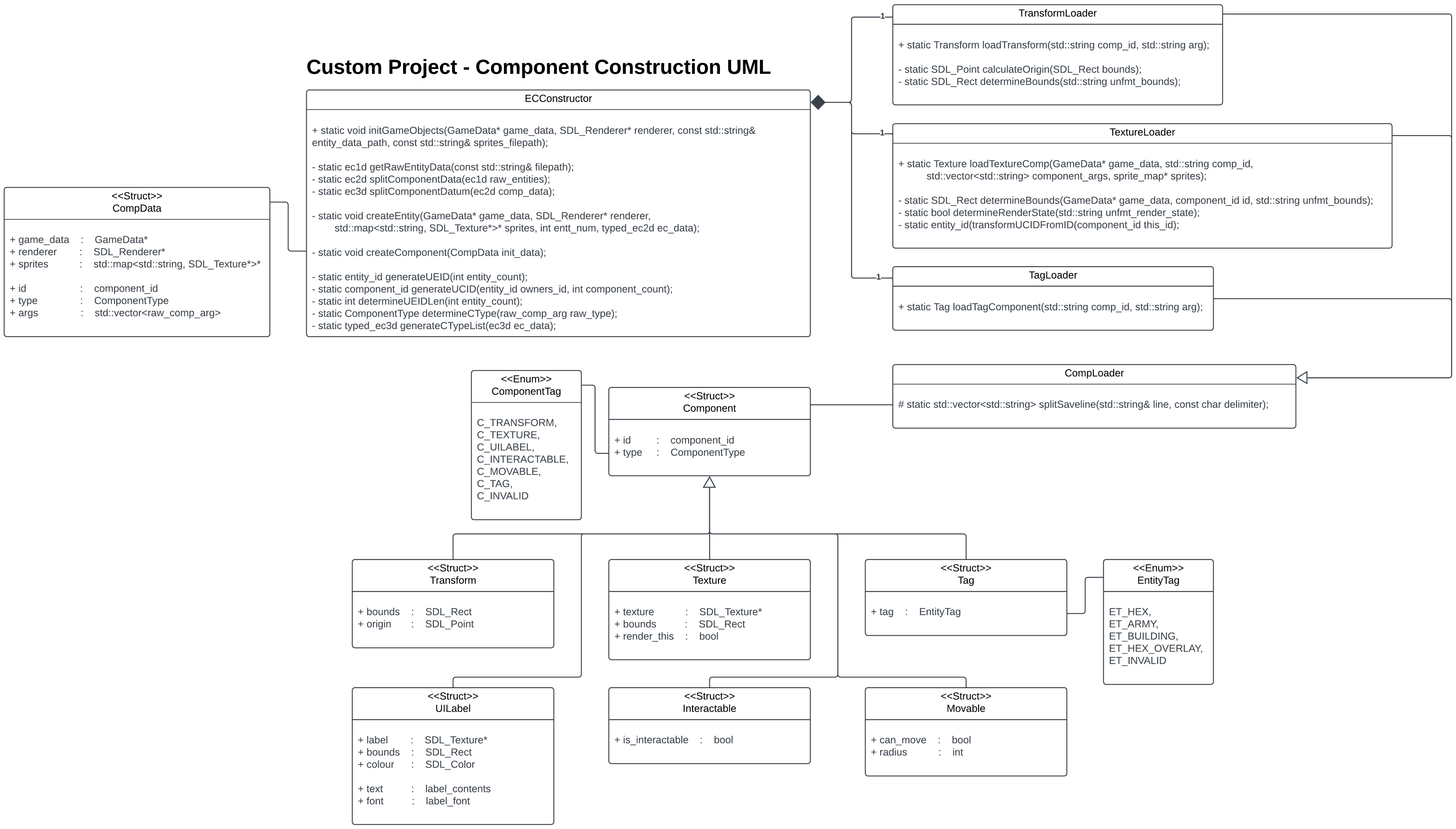


```
void ECS::ECConstructor::initGameObjects(GameData* game_data, SDL_Renderer* renderer,
    const std::string& entity_data_path, const std::string& sprites_filepath) {
    auto sprites = Game::SpriteLoader::loadSprites(sprites_filepath, renderer);
    typed_ec3d fmt_entt_data = generateCTypeList( splitComponentDatum(splitComponentData(getRawEntityData(entity_data_path)))

    static int entity_count = 0;
    for (typed_ec2d entity : fmt_entt_data) {
        createEntity(game_data, renderer, &sprites, entity_count, entity);
        ++entity_count;
    }
}
```

How the Entity Component Constructor instantiates each component within the game world.

Custom Project - Component Construction UML

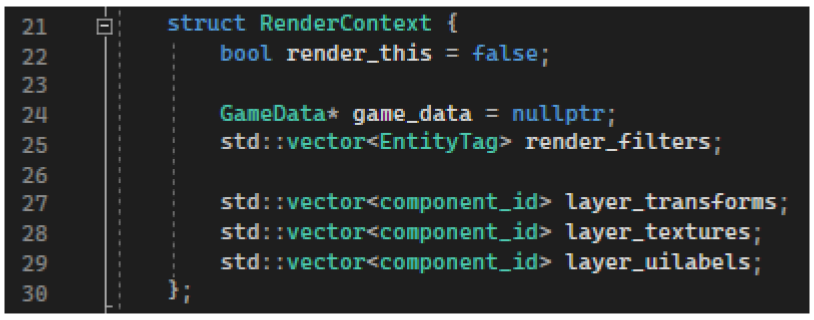


<<Struct>>
CompData
+ game_data : GameData*
+ renderer : SDL_Renderer*
+ sprites : std::map<std::string, SDL_Texture*>*
+ id : component_id
+ type : ComponentType
+ args : std::vector<raw_comp_arg>
ECConstructor
+ static void initGameObjects(GameData* game_data, SDL_Renderer* renderer, const std::string& entity_data_path, const std::string& sprites_filepath);
- static ec1d getRawEntityData(const std::string& filepath);
- static ec2d splitComponentData(ec1d raw_entities);
- static ec3d splitComponentDatum(ec2d comp_data);
- static void createEntity(GameData* game_data, SDL_Renderer* renderer, std::map<std::string, SDL_Texture*>* sprites, int entt_num, typed_ec2d ec_data);
- static void createComponent(CompData init_data);
- static entity_id generateUEID(int entity_count);
- static component_id generateUCID(entity_id owners_id, int component_count);
- static int determineUEIDLen(int entity_count);
- static ComponentType determineCType(raw_comp_arg raw_type);
- static typed_ec3d generateCTypeList(ec3d ec_data);
1
TransformLoader
+ static Transform loadTransform(std::string comp_id, std::string arg);
- static SDL_Point calculateOrigin(SDL_Rect bounds);
- static SDL_Rect determineBounds(std::string unfmt_bounds);
1
TextureLoader
+ static Texture loadTextureComp(GameData* game_data, std::string comp_id, std::vector<std::string> component_args, sprite_map* sprites);
- static SDL_Rect determineBounds(GameData* game_data, component_id id, std::string unfmt_bounds);
- static bool determineRenderState(std::string unfmt_render_state);
- static entity_id(transformUCIDFromID(component_id this_id);
1
TagLoader
+ static Tag loadTagComponent(std::string comp_id, std::string arg);
CompLoader
# static std::vector<std::string> splitSaveline(std::string& line, const char delimiter);
<<Enum>>
ComponentTag
C_TRANSFORM,
C_TEXTURE,
C_UILABEL,
C_INTERACTABLE,
C_MOVABLE,
C_TAG,
C_INVALID
<<Struct>>
Component
+ id : component_id
+ type : ComponentType
<<Struct>>
Transform
+ bounds : SDL_Rect
+ origin : SDL_Point
<<Struct>>
Texture
+ texture : SDL_Texture*
+ bounds : SDL_Rect
+ render_this : bool
<<Struct>>
Tag
+ tag : EntityTag
<<Enum>>
EntityTag
ET_HEX,
ET_ARMY,
ET_BUILDING,
ET_HEX_OVERLAY,
ET_INVALID
<<Struct>>
UILabel
+ label : SDL_Texture*
+ bounds : SDL_Rect
+ colour : SDL_Color
+ text : label_contents
+ font : label_font
<<Struct>>
Interactable
+ is_interactable : bool
<<Struct>>
Movable
+ can_move : bool
+ radius : int

TDD

EventHandler and Component Subsystems

💡 The systems built here are designed to know nothing of the components they work on. Each system operates under assumptions made through analyzing a context which is constructed and passed the system via the `EventHandler`. Additionally, each system doesn't store pointers to components but sets of `component_id's` which it can use to quickly access whatever component it needs.

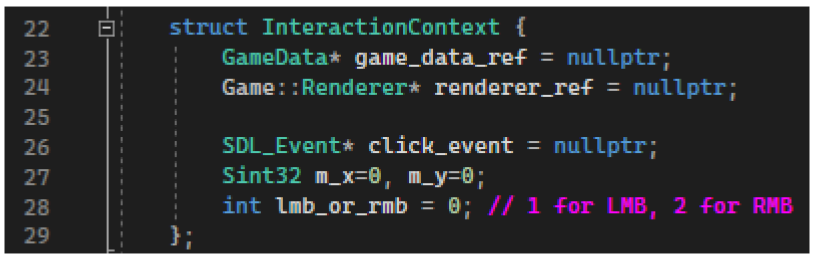


```
struct RenderContext {
    bool render_this = false;

    GameData* game_data = nullptr;
    std::vector<EntityTag> render_filters;

    std::vector<component_id> layer_transforms;
    std::vector<component_id> layer_textures;
    std::vector<component_id> layer_uilabels;
};
```

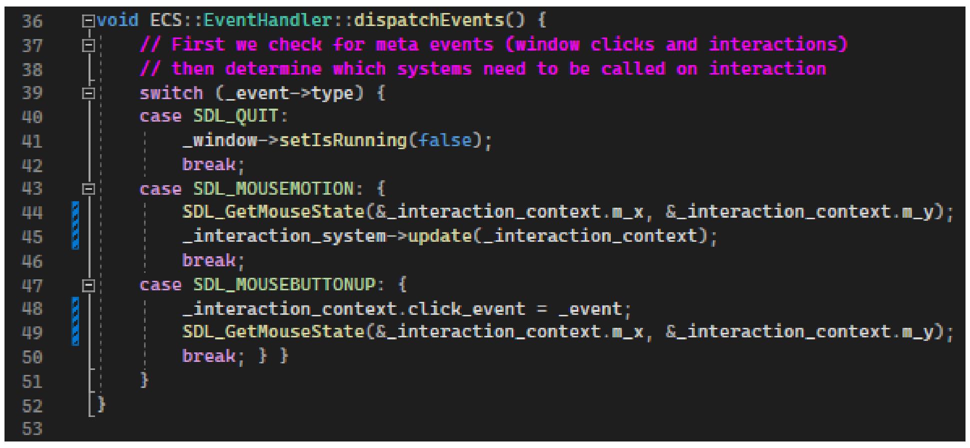
RenderContext



```
struct InteractionContext {
    GameData* game_data_ref = nullptr;
    Game::Renderer* renderer_ref = nullptr;

    SDL_Event* click_event = nullptr;
    Sint32 m_x=0, m_y=0;
    int lmb_or_rmb = 0; // 1 for LMB, 2 for RMB
};
```

InteractionContext



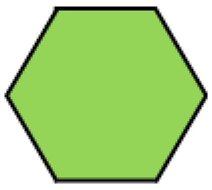
```
void ECS::EventHandler::dispatchEvents() {
    // First we check for meta events (window clicks and interactions)
    // then determine which systems need to be called on interaction
    switch (_event->type) {
    case SDL_QUIT:
        _window->setIsRunning(false);
        break;
    case SDL_MOUSEMOTION: {
        SDL_GetMouseState(&_interaction_context.m_x, &_interaction_context.m_y);
        _interaction_system->update(_interaction_context);
        break;
    case SDL_MOUSEBUTTONUP: {
        _interaction_context.click_event = _event;
        SDL_GetMouseState(&_interaction_context.m_x, &_interaction_context.m_y);
        break; } }
    }
}
```

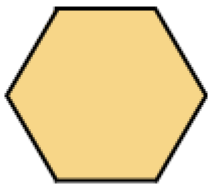
As we're only caring about mouse inputs (Turn-based strategy game assumption) the context of these inputs become more important. Have we already got an entity selected? Are we in a menu? Hence the use for Context structs.

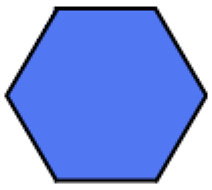
Layered Rendering

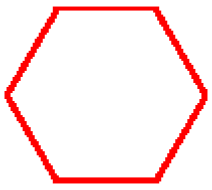
💡 One of the big hurdles for this project was building a rendered layering system whose elements are able to be initialized from a tag and updated manually each tick if required. Each layer contains sets of texture, transform and label component id's whose render statuses it's able to manipulate and whose textures it's able to display at the correct position.

Assets

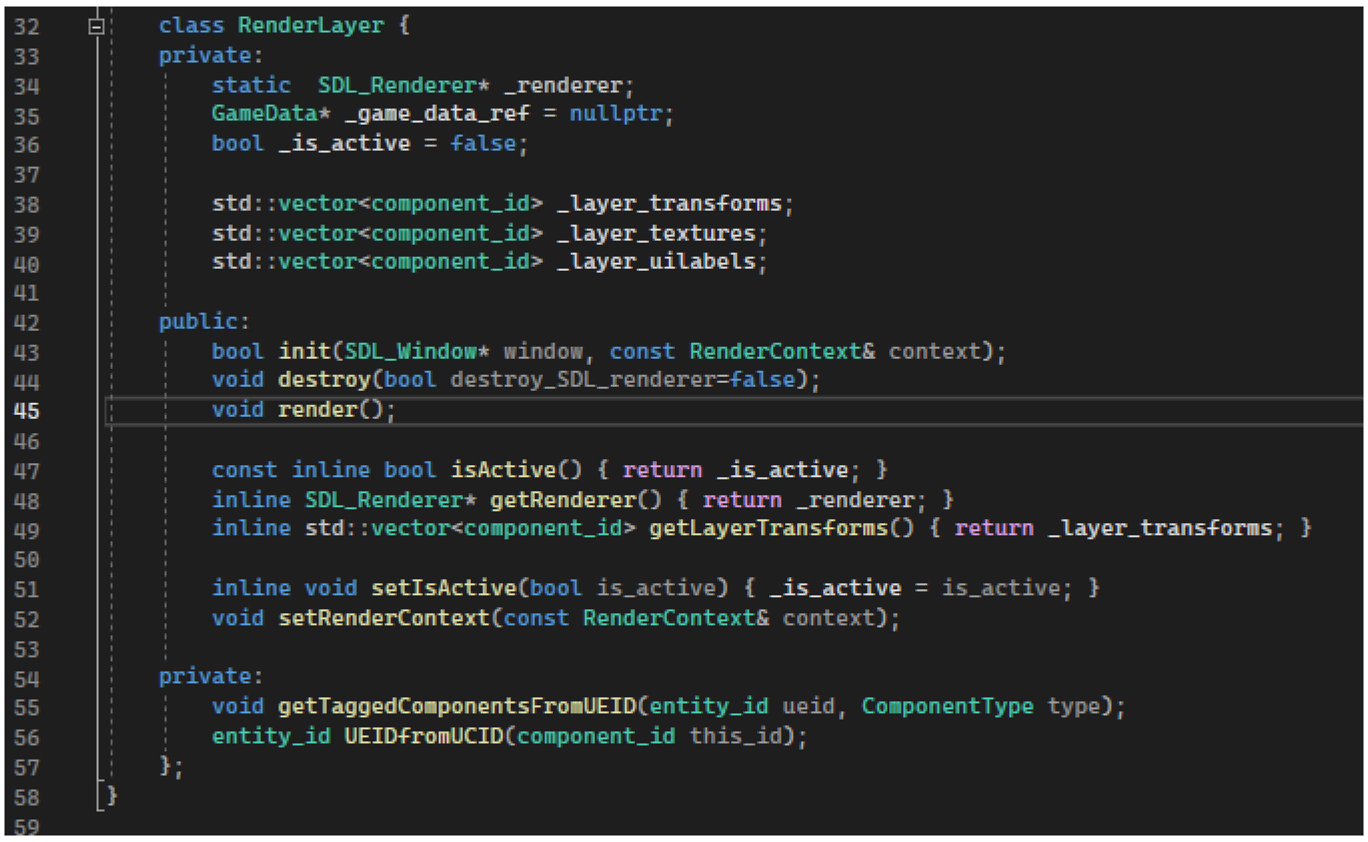








Render Layers



```
    class RenderLayer {
    private:
        static  SDL_Renderer* _renderer;
        GameData* _game_data_ref = nullptr;
        bool _is_active = false;

        std::vector<component_id> _layer_transforms;
        std::vector<component_id> _layer_textures;
        std::vector<component_id> _layer_uilabels;

    public:
        bool init(SDL_Window* window, const RenderContext& context);
        void destroy(bool destroy_SDL_renderer=false);
        void render();

        const inline bool isActive() { return _is_active; }
        inline SDL_Renderer* getRenderer() { return _renderer; }
        inline std::vector<component_id> getLayerTransforms() { return _layer_transforms; }

        inline void setIsActive(bool is_active) { _is_active = is_active; }
        void setRenderContext(const RenderContext& context);

    private:
        void getTaggedComponentsFromUEID(entity_id ueid, ComponentType type);
        entity_id UEIDfromUCID(component_id this_id);
    };
}
```

RenderLayer declarations

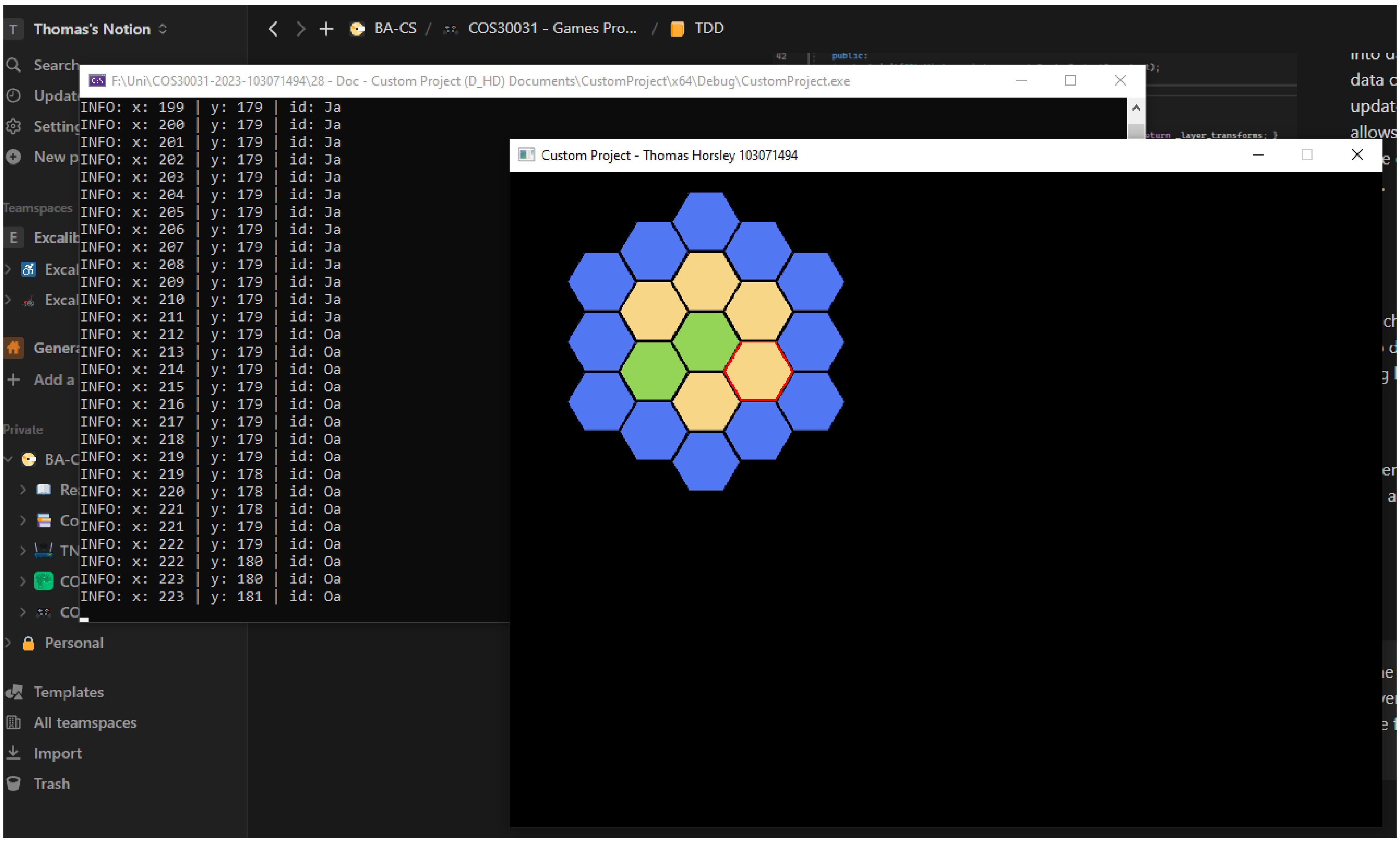
The render layer can be toggled and is able to take a context. This context can be decoded into data and from this components updated. This system is what allows the red hexagon to be overlaid on the a unique position on the grid.

This architecture allows for UI groups to be instanced and painted on-top of each other. Interactions between the user and entities on the current layer can trigger UI to display / disappear or components to be updated by systems without RenderLayers directly owning large sets of likely shared (and therefore copied) component data.

Each RenderLayer is held in a vector owned by the renderer. RenderLayers are rendered from the bottom up using a painters algorithm (Layers are only rendered if they’re set to active).

Conclusion

💡 My custom project started as the backend for a Turn-based strategy game and finished as an Entity Component System with multiple render layers and a custom EventDispatching architecture. Though the system is clearly incomplete, I feel as though the foundations are solid and I look forward to expanding on it in the future.


Thomas's Notion
BA-CS / COS30031 - Games Pro... / TDD
Search
Updat
Setting
New p
Teamspaces
Excalib
Excal
Excal
Genera
Add a
Private
BA-C
Re
Co
TN
CO
CO
Personal
Templates
All teamspaces
Import
Trash
F:\Uni\COS30031-2023-103071494\28 - Doc - Custom Project (D_HD) Documents\CustomProject\x64\Debug\CustomProject.exe
Custom Project - Thomas Horsley 103071494

```
INFO: x: 199 | y: 179 | id: Ja
INFO: x: 200 | y: 179 | id: Ja
INFO: x: 201 | y: 179 | id: Ja
INFO: x: 202 | y: 179 | id: Ja
INFO: x: 203 | y: 179 | id: Ja
INFO: x: 204 | y: 179 | id: Ja
INFO: x: 205 | y: 179 | id: Ja
INFO: x: 206 | y: 179 | id: Ja
INFO: x: 207 | y: 179 | id: Ja
INFO: x: 208 | y: 179 | id: Ja
INFO: x: 209 | y: 179 | id: Ja
INFO: x: 210 | y: 179 | id: Ja
INFO: x: 211 | y: 179 | id: Ja
INFO: x: 212 | y: 179 | id: Oa
INFO: x: 213 | y: 179 | id: Oa
INFO: x: 214 | y: 179 | id: Oa
INFO: x: 215 | y: 179 | id: Oa
INFO: x: 216 | y: 179 | id: Oa
INFO: x: 217 | y: 179 | id: Oa
INFO: x: 218 | y: 179 | id: Oa
INFO: x: 219 | y: 179 | id: Oa
INFO: x: 219 | y: 178 | id: Oa
INFO: x: 220 | y: 178 | id: Oa
INFO: x: 221 | y: 178 | id: Oa
INFO: x: 221 | y: 179 | id: Oa
INFO: x: 222 | y: 179 | id: Oa
INFO: x: 222 | y: 180 | id: Oa
INFO: x: 223 | y: 180 | id: Oa
INFO: x: 223 | y: 181 | id: Oa
```