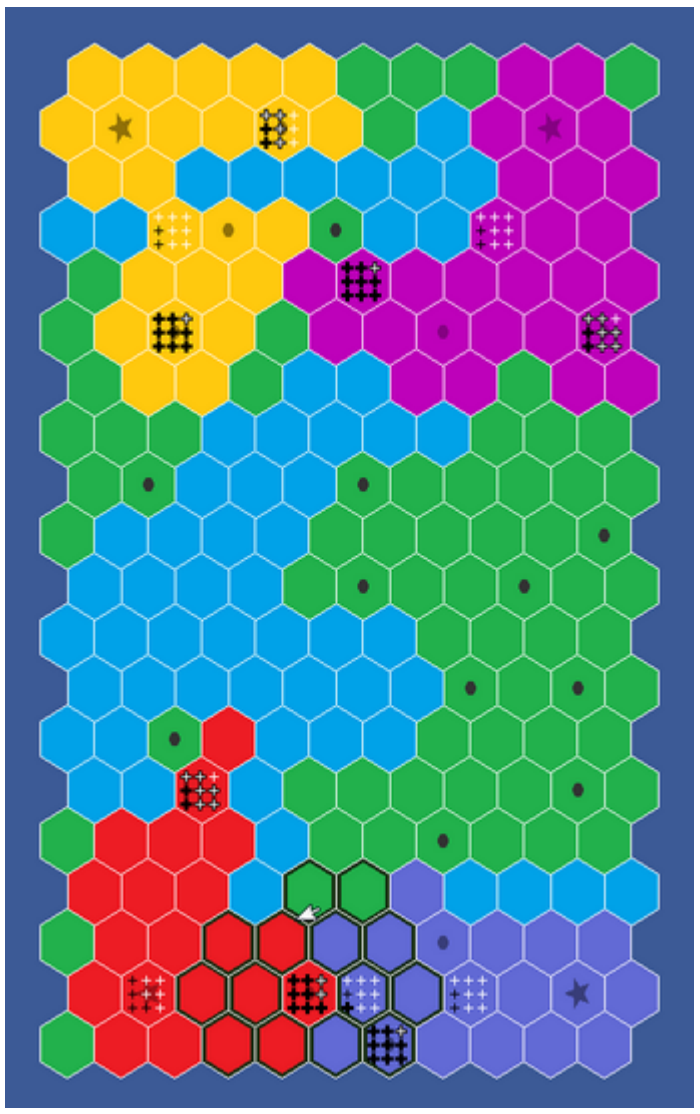
# 📜 Custom Project Plan

## Introduction

> 💡 This document serves as an overview and outline for my D level custom project. It contains relevant ‘*production promises*’ with design & technical outlines.
>
> This project stems from my thousands of hours spent in strategy games, so developing a Turn-Based Strategy game backend sounds like I project I’d be willing to stick with for a while and could potentially make it’s way onto a developer portfolio.

## My Project Description

> 💡 For this project I will develop the foundations for a TBS (Turn Based Strategy) game rendered on a hexagonal grid.



HexEmpire map. Simple hex grid example.


Total War Warhammer 3; building/ unit dependencies, source: https://www.moddb.com/mods/ultimate-skaven/images/building-chains

## Feature List

- Rendered Hex grid
- Upgradable building entities, stacking technology bonuses and entity tier / hierarchy systems
- Unique units and armies composed of moveable units some with building / technology dependencies

---

# What I will Document

> 💡 Progress for this project will be maintained through a set of Spike reports. Additionally, I will construct an initial UML design and changes made to this during development will be noted with final documentation presented.

## Supporting Documentation

- **Game Design Document**
  - Introduction / Description
  - Feature lists and project plan.
  - Release notes
- **Technical Design Document**
  - Tech, Tools & Resources
  - Spike Reports
  - Architecture and design decisions
  - Git commit history + readme

---

# What I will Make

## Entities, Components & Systems

> 💡 The game will be built on-top of an ECS backend. All systems (bar the Renderer) inherit from a System class. This provides them a unique `update()` which is called to manipulate the relevant components given a context.

Entities serve only as ID's in this implementation, acting as a link between components. For example, `Entity1.id = 'a'`, `component1Type1.id = 'aA'` `component1Type2 = 'aA'` any system which is called to work on Entity1 knows the Types and ID's of components it requires ahead of time and can search a cached hash table of local components for whatever it needs. Assuming all relevant components are located then that system is added to the `EventDispatcher._event_queue` and their `update()` are executed in order.

# Spike 28 - Initial ECS

**Transform**
+ bounds : SDL_Rect

**Origin**
+ origin : SDL_Point

**Texture**
+ render_this : bool
+ texture : SDL_Texture*
+ subtexture : SDL_Rect

**UILabel**
+ position : SDL_Rect
+ contents : string
+ font : string
+ colour : SDL_Colour

**Interactable**
+ currently_interactable : bool

**Movable**
+ move_radius : int

**Tag**
+ tag : EntityTag

**<<Enum>> EntityTag**
E_INVALID,
E_TILE,
E_BUILDING,
E_ARMY

**Entity**
+ _id : string

**Component**
+ _id : string
+ _type : ComponentType

**<<Enum>> ComponentType**
C_INVALID,
C_TRANSFORM,
C_ORIGIN,
C_TEXTURE,
C_UILABEL,
C_INTERACTABLE,
C_MOVABLE,

**<<Struct>> GameData**
+ entities : map<entity_id, Entity>
+ components : map <component_type, map <component_id, Component>

**InputHandler**
- _input_context : InputContext
+ getContext( ) : InputContext
+ setContext(InputContext) : void
+ constructEventContext(InputContext) : EventContext

**<<Struct>> InputContext**
+ click_pos : SDL_Point
+ left_or_right_click : bool
+ input_state : InputState

**<<Enum>> InputState**
IS_INVALID,
IS_DEFAULT,
IS_ARMY_SELECTED,

**Game**
- _game_data : GameData
- _renderer : Renderer*
- _input_handler : InputHandler
- _input_context : InputContext
- _event_dispatcher : EventDispatcher
+ Game( ) : Game
+ ~Game( ) : Game
+ update( ) : void
+ render( ) : void
- pollEvents( ) : SDL_Event*

**Renderer**
- _render_layers : stack<RenderLayer*>
- _max_layers : size_t
+ update( ) : void
+ render( ) : void

**RenderLayer**
- _is_active : bool
- _renderer : static SDL_Renderer*
- _layer_transform_ids : vector<component_id>
- _layer_texture_ids : vector<component_id>
- _layer_ui_label_ids : vector<component_id>
+ update( ) : void
+ render( ) : void

**EventDispatcher**
- _all_systems : map<EventType, System*>
- _event_queue : queue<System*>
+ constructEvents(EventContext) : void
+ dispatchEvents( ) : void
+ clearEventQueue( ) : void

**<<Struct>> EventContext**
+ selected_entity_id : string
+ event_type : EventType

**<<Enum>> EventType**
INVALID,
INTERACT,
MOVEMENT,
UPGRADE,
RECRUIT,

**System**
# _relevant_entities : vector<entity_id>
# _relevant_components : map <component_type, vector<component_id>
# _args : SystemContext
+ subsystem_tag : EventType
+ getTag( ) : SystemTag
+ setArgs(SystemContext) : void
+ pure virtual update( ) : void

**<<Struct>> SystemContext**
+ selected_entity_id : string

**InteractSystem**

**MovementSystem**

**UpgradeSystem**

**RecruitmentSystem**

I’m gonna make it… the specificities of systems are ignored as production must start sometime.

## Gameworld Interactivity Abstractions

> 💡 The only valid inputs within the game are left & right mouse click. Therefore, the context in which these inputs are used is the most critical data for triggering the correct system `update()` call.

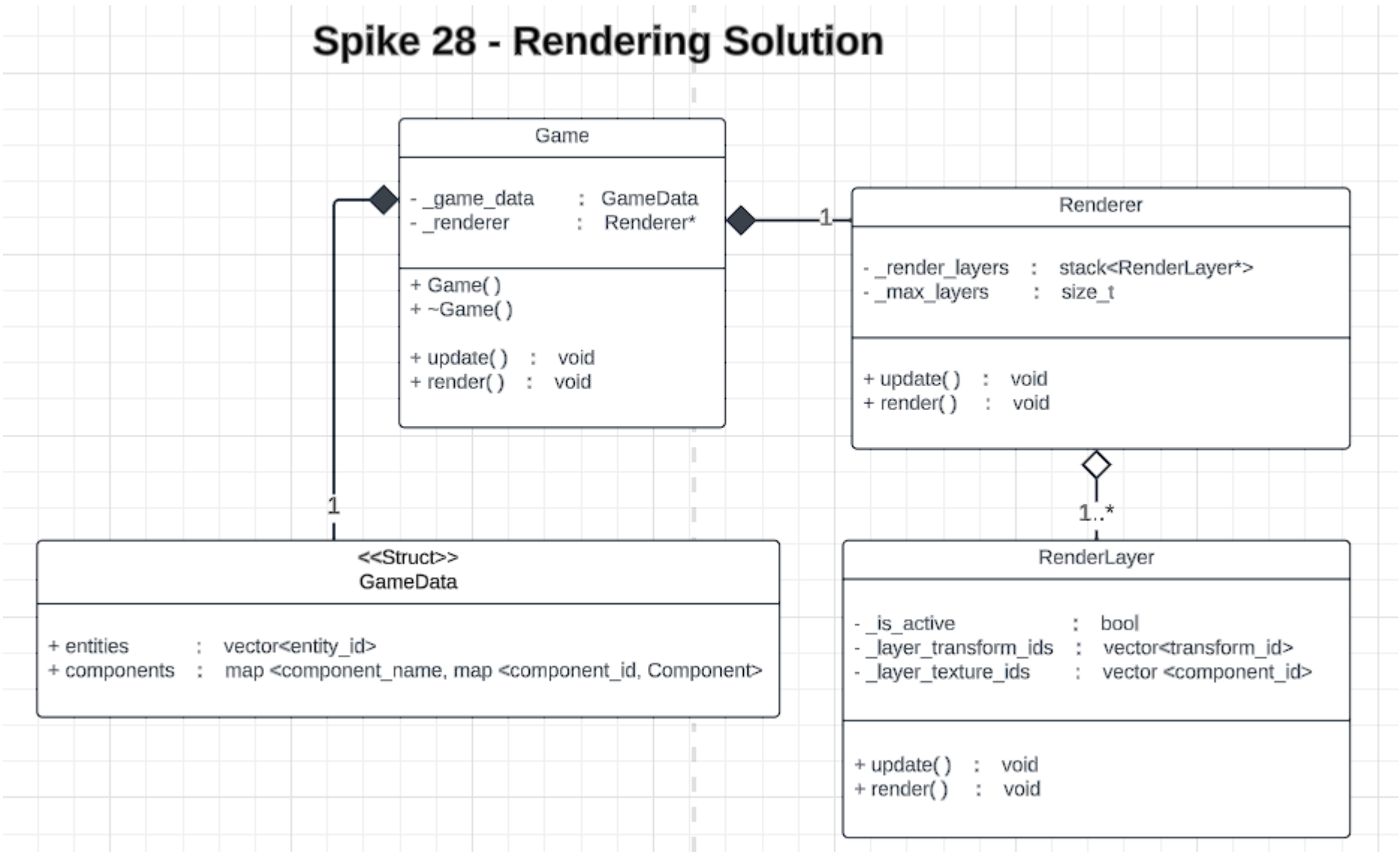
I believe the context of the users input can be derived from only the input position, previous input state and whether the input was a Left or Right Mouse Down Event. Once an InputContext has been constructed, the EvenDispatcher is able to construct an EventContext and queue the relevant systems for component manipulation. The UML above is constructed with this assumption.

## UI & Rendering Shenanigans

> 💡 As UI and concise data display is a critical component of quick to learn and replayable Turn-based Strategy Games, it’s architecture has been heavily considered. The Renderer is an ECS system but does not inherit from the System class due to it’s specific nature.

My initial solution involves togglable rendering layers overlaid onto the Gameworld each containing UI elements and whose rendering state (on or off) is modified via Event Dispatch and Command Queue mechanisms. Each layer will be rendered from the bottom up and the state of each renderer contained within the Layer is mutable by the Layer, this will allow for the bypassing of unnecessary renders.

Buttons and UIPanels are Entities containing Texture, UILabel and (in the case of buttons) Interactable components.




Planned implementation of a 2D multilayer rendering pipeline.

# What I will Present At The Interview

> 💡 This section is written in order of importance using stages. Stage 1 is minimum viable product with further stage demo’s being presented if time allows.

## Stage 1

- Developer Notebook
- Demo map containing tiles, upgradeable buildings and recruitable armies
- Entity Component System architecture
- Layered Rendering mechanisms
- Game Design Documentation
- Technical Design Documentation

## Stage 2

- Movement system for armies
- Tech tree providing global bonuses
- Entity dependency architecture

---

# Conclusion

> 💡 For my D level custom project I will be implementing the backend for a 2D Turn-Based Strategy game. The backend will contain a scalable Entity Component System which I will demonstrate along-side a Tech Demo at the interview.

---