Ambrose Doppler and The Atlas Unlimited

Fully Functional Engine – Technical Documentation

GAM-250 Technical Guide - Spring 2023

Team Perfectly Normal

Programmers:
Mason Dulay - Programer
Ashkon Khalkhali - Technical Lead
Jonathan Meyer - Design Lead
Alijah Rosner - Producer

Artists:
Owen Svoboda - Art Producer. Art Lead, Rigger
Christi Thai -Character, UI
Jeff Chou - Background, Props

Contents:

Architecture Diagram……...……...4
Engine Overview……...5

- Entity Component System
    - Entity
    - Scene
    - Components
        - Behavior Components
        - Collider Components
        - Occluder Components
        - Vision Components
        - Transform Components
        - Physics Components
        - Animation/Sprite Components
        - Particle Components

Systems……..6

- ISystem
    - Audio
    - Graphics
    - Renderer
    - Input
    - Window
    - Manager
        - Scene

Graphics.....……..…….....7
Physics.....……..…….....8
Player Controls……...……...……..9
Behavior Implementation……...…….……...…9
Programming Overview……..……9,10

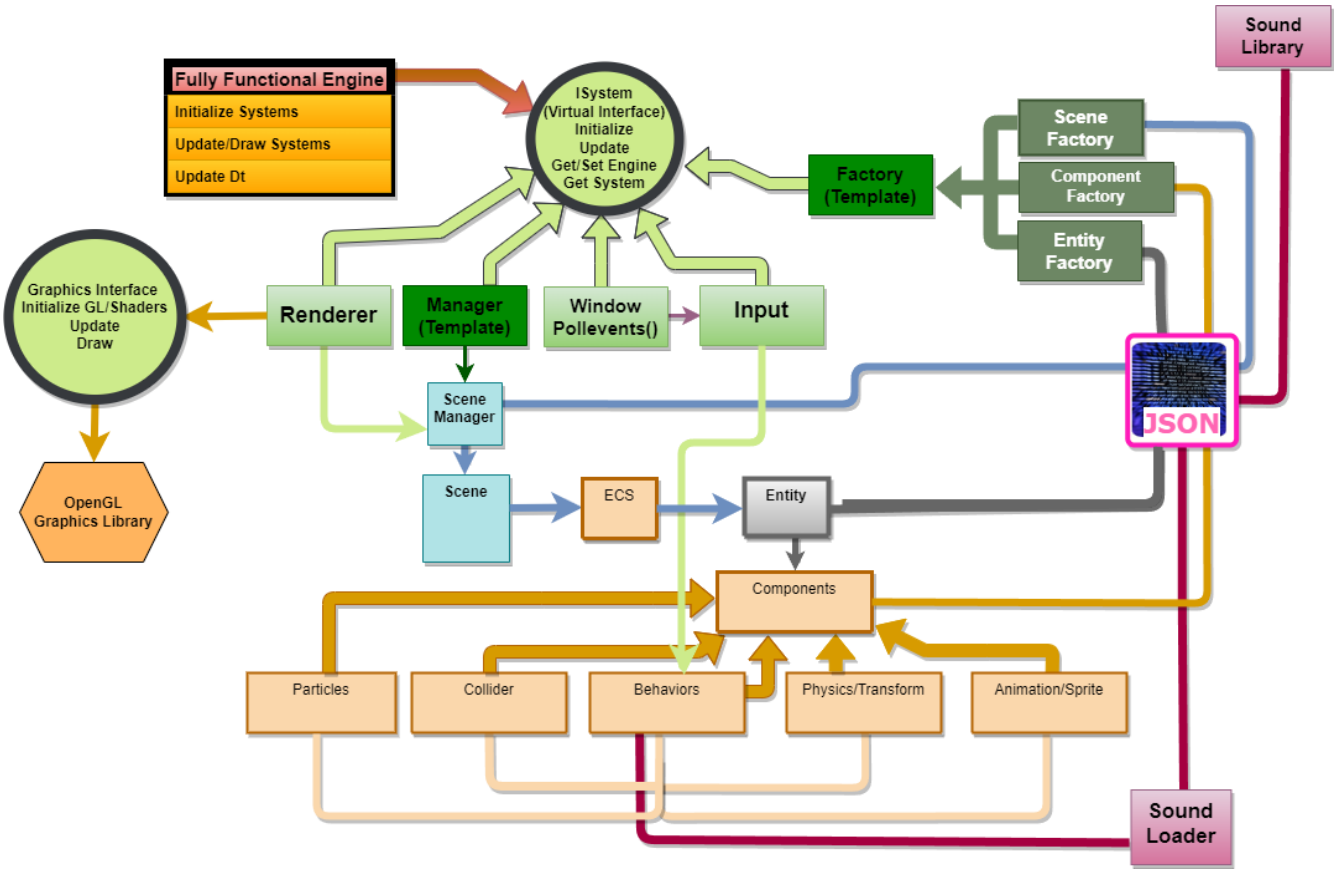
- Debugging
- Conventions

- Version Control
- Tools

Technical Risks...……11

Appendices……...……....12

Architecture Diagram:



Engine Overview:

The Fully Functional **Engine** is organized using an **Entity Component System** (ECS) based design pattern to facilitate 2D game making.

Engine: This object initializes/updates systems, and tracks frame intervals (dt).

Entity: Entities are integer values used to group components to a unique ID. Provides no functionality other than obtaining the **ECS.**

Entity Component System: A manager contained in each **Scene,** built to update all components within it.

Scene: Scenes are containers for all entities and **components** included in a level.

Components: Self contained objects that provide unique functionality.

- **Behavior Components:**
  - Allows unique functionality for entities.
- **Collider Components:**
  - Adds a shape to an entity that checks for collisions with other colliders, and communicates with other entities upon collision.
- **Occluder Components:**
  - Adds a rectangle boundary that can ‘occlude’ the player from vision components depending on height to enable stealth and.
- **Vision Components:**
  - Has a field of view and horizon threshold that is used to communicate with other components when the player is not only visible, but unoccluded (see occluder component)
- **Transform Components:**
  - Provides a position, scale, and rotation to an entity to manipulate.
- **Physics Components:**
  - Allows entities to have velocity, and tracks previous position.
- **Animation/Sprite Components:**
  - Attaches sprite sheets or still images to entities respectively, and provides opacity functionality and drawn with pseudo-depth using Z-priority (Sprites with lower Z-Priority will draw first)
- **Particle Emitters:**
  - Added to entities and drawn separate from sprites, particle emitters will draw several instances of an image above other graphics.

All entities and components within a scene are **deserialized** from level files in the data directory.

Deserialization: Loads data from files, used for all level data and audio tracks. Levels are separated into a main file that points to the entities and environment folders where each entity can be separated into individual files.

Factory Template: Factories are built for each object type included in our scenes to use when loading data from disk.

All components are updated through **Component Managers** within a Scene's ECS. Scenes are loaded from file data through **deserialization**, and added to a **Manager System** upon creation.

ISystem: A virtual interface with generic functions for all systems.

Input: Initializes and updates vector list of keys to defined keystates.

Window: Initializes/updates main Window, polls key actions and mouse position.

Manager: Abstract template to contain, update, and deallocate objects.

Audio: FMOD system that will play sounds triggered by behavior components.

Renderer: ISystem that interfaces with Scene's ECS and Graphics to draw images.

Graphics Overview:

Graphics API: OpenGL 3.3.

Rendering Pipeline:

Initialize GLEW/OpenGL.

Create Quad Mesh and Default Shaders.

Assign shaders and draw meshes to screen.

Sprite Loading: Sprites are loaded through texture paths from Level Data files using STBI Library.

Shaders: a handful of shaders are used throughout our renderer, The default shader handles normal sprites and animations, and applies a film grain overlay by default. There are also

screen space shaders that draw over the screen to highlight collider shapes in worldspace, as well as highlighting lines of sight visible by the vision components in the scene, as well as a ‘choma’ hue scale for any time dilation nodes (distortions in dt around points in space).

Animations: Animations are loaded through sprite sheets using STBI Library, with UV offsets.

Particle: Particle images are loaded through texture paths, and treated as a unique component.

Physics Overview:

Kinematics:

Movement is accomplished using two vector values: **Position** and **Velocity**.

Position is calculated by:

P(t) = P(n-1) + (V(n-1) * dt)

Velocity is a constant vector set in entity data.

Collision:

Colliders are composed of **tags** and **signals:**

- **Tags:**
  - **Shape:** denotes collider form.
  - **Static:** determines resolution.
  - **Enabled:** determines active state.

- **Signals:**
  - EnterCollision2D: Called on collider entry.
  - ExitCollision2D: Called on collider exit.
  - EntityFound: Called when the vision component spots the playerController
  - EntityLost: Called when the vision component no longer spots the PlayerController.

Detection:
Determined using the separating axis theorem (SAT) makes all convex shapes supported.

Resolution:
The trigger is simply set to their last position in cases where the collider is “Static.”

In the future, mean value theorem (MVT) or something comparable will be implemented for a cleaner resolution calculation.

Player Controller Overview:

Our engine uses a **polling-based** input system. Player controller logic will check keystates set by GLFW event polling. Custom input mapping is currently not supported.

Input manager: Manages keyboard/button actions and mouse cursor position.

Behavior Implementation:

Behaviors are derived from an interface that automatically subscribe to their entities’ signals.

- **Collider Signals:**
  - Behaviors listen for:
    - EnterCollision2D: Sent by collider on entry.
    - ExitCollision2D: Sent by collider on exit
- **Vision signals:**
  - Behaviors listen for:
    - EntityFound: Send by the vision component when the player becomes visible.
    - EntityLost: Sent by the vision component when the player is no longer visible.

Behaviors can opt into using any number of Finite State Machines, allowing for easy adding, removing, and debugging of states, The behaviors themselves contain the logic for updating any included states.

Debugging:

Debug Drawing/God Mode: Simple Debug Drawing is togglable. When toggled on, collision boxes are shown and player character is indestructible.

Instant Win Condition: All conditions required to move to the next level can be met using "**-**" key.

Scene Reload: Pressing Key "K" triggers the scene reloader and is implemented to reload entities added into JSON on runtime.

Frame Rate: Frame rate is displayed and updated on the game's window, in the title bar.

Assertions: Global assert macros exist to check validity of data deserialized at runtime.

Console Logging: Trace logging is used throughout the engine to serialize desired debug values. Colliders currently log signals sent on entry.

Coding Methods:

All files are stored in the source folder.

File Naming Conventions:

UpperCamelCase.cpp

UpperCamelCase.h

Coding Name Conventions:

- Functions:
  - UpperCamelCase
- Defines:
  - SCREAMING_SNAKE_CASE
- Variables:
  - UpperCamelCase_
  - lowerCamelCase

Styling:

File Headers

Curly braces on their own line

Guidelines:

Use the stub file for every new file created.

Meaningful comments on version commits.

Try to put unique classes in separate files.

Comments above classes encouraged.

Version Control:

SVN used by Programmers and Artists through folders “FullyFunctional” and “Assets” respectively.

Tools:

- Project is contained in a **Visual Studio** solution, compiled in C++20.
- **FMOD** is used to initialize and play sounds from our sound library.
- **GLFW** is used for Window creation and event polling.
- **GLEW** is used to call OpenGL functions in our Graphics Interface.
- **GLM** library is used for collision and physics math.
- **RapidJson** is used for deserialization.
- **OpenGL** is used for our Rendering system.

Editor Implementation:

- All entities are written in JSON level data files, deserialized through RapidJSON. At the moment, no editor implementation is provided.

Technical Risks:

Behavior Overflow:

Due to the tightly-coupled nature of our behavior implementation with their assigned entities, it is very likely that the influx of behavior components can lead to misuse and code bloat. Our suggested solution is to research modular implementations for behaviors, such as generic behavior types for interactables, pickup items, and NPCs.

Duplicate Components:

As of now, entities are only capable of having one unique component per type. This is an issue for Entities with colliders and vision cones for instance. Our proposed solution would be to allow factories to tag components of the same type to a single entity, if requested.

However, this would prove to be problematic when determining components for debug purposes, as well as obtaining components using EntityID's, which is currently built to handle single instances only.

Appendices:

Appendix A: Art Requirements

Naming Conventions: ASSET_UpperCamelCase_Version

Image Formats: PNG, where X and Y size are divisible by 2.

Art Pipeline: Add the asset to the appropriate file directory. Open the Scene data file that corresponds to the desired asset location, and attach all corresponding components along with the sprite.

Art Assets: Programmer and artist art.

Appendix B: Audio Requirements

Naming Conventions: lowerCamelCase.mp3

File Formats: mp3

Sound Pipeline: Programmers add the audio file to the audio folder and load it into the engine through the Audio Factory to store in the library for use. Sounds are obtained through the soundly library.