Technical design document

Target platforms
Windows

Development platforms
Windows

Languages
C++ 17
HLSL

Development tools
Visual Studio 2019
Git
GitHub

Version control strategy
The project will be version controlled with Git. Remote repositories will be stored on GitHub with the most up to date stable version of the project in the main branch. Feature branches will be created for separate development of features before they are merged back into the main branch upon feature completion.

Third party libraries
Dear ImGui
Yaml-cpp
DirectXToolkit for Direct3D 11
DirectX
Direct3D 11

Engine library implementation
The engine library will build from and extend the functionality provided in Scarle engine, housing common functionality among its target applications. A static library will be used for the engine allowing client application projects to link against it and compile its functionality into their binary executables.

The engine will build into a static library, outputting a .lib binary file.

# Code style guide
\`member variable = m_variable_name\`
\`local variable = variable_name\`
\`paramater variables = param_name\`
\`functions = functionName()\`
\`class/struct = ClassName/StructName\`

`enum/enum class = EnumName`
`const variables = CONST_VARIABLE`

Entity component system

The goal of this system is to implement an entity and component model that can support many entities in the world with acceptable performance. In an object oriented implementation, the CPU caches will become filled with unnecessary data causing fetches to slower memory and potential for cache misses. The system that will be implemented uses a data oriented design where component data is stored contiguously and keeps related data together when loaded into CPU cache, reducing the amount of slower memory fetches.

Class design and layout

The Entity Component System will comprise three main systems. These will be the Entity Manager, the Component Manager and the System Manager.
The Entity Manager will be responsible for tracking and providing the unique ids that will be being used to define the entities.
The Component Manager will be responsible for constructing the arrays where the component data will be stored as well as providing an efficient lookup system for getting that data with only the entity id.
The System Manager will be responsible for registering systems that are in use as well as notifying all systems that an entity has changed composition or been deleted.

For this implementation there will also be a wrapper class called ECS that will act as the interface for the entire entity component system. The wrapper class will be a unique_ptr member variable of the Game class, and will be created and initialised in Game’s `init()` function. The ECS will then be used as an interface to interact with the entity component system.

```
m_ecs = std::make_unique<ECS::ECS>();
m_ecs->init();
```

Components will need to be registered with the ECS before they can be used by the ECS systems. With this implementation of an entity component system it will use generic templates to convert normal structs into components.

```
m_ecs->registerComponent<Gravity>();
m_ecs->registerComponent<RigidBody>();
m_ecs->registerComponent<Transform>();
```

Systems will then need to be registered with the ECS just like components. Systems will use a bitflag to create a signature that states what combination of components it cares about.

```
m_physics_system = m_ecs->registerSystem<PhysicsSystem>();
m_physics_system->init(*m_ecs);
```

Each system inherits from the System base class that is a part of the ECS namespace. Though the system does use inheritance it is not virtual. For abstraction and general code style each system should include an `init()` function that takes a reference to the ECS. In the `init()` function we will then define the components that that system cares about by constructing a signature.

```
void PhysicsSystem::init(ECS::ECS& ecs)
{
    ECS::Signature psSig;
    psSig.set(ecs.getComponentBitflag<Gravity>());
    psSig.set(ecs.getComponentBitflag<RigidBody>());
    psSig.set(ecs.getComponentBitflag<Transform>());
    ecs.setSystemSignature<PhysicsSystem>(psSig);
}
```

Example usage

The engine will provide an example physics system that integrates velocity for entities that have a rigid body, gravity and transform component and then can update the position in the transform component. Entities will be created in the game's `start()` method and will have components added to them:

```
auto& ecs = *ECS();
m_entity1 = ecs.createEntity();
ecs.addComponent<Gravity>(m_entity1, Gravity(-2.f));
ecs.addComponent<RigidBody>(m_entity1, RigidBody());
ecs.addComponent<Transform>(m_entity1, Transform());
```

During the game's game object `update_objects()` method, the physics system will be updated, updating the components of entities containing the required components:

```
physics()->update(*ECS(), gd->m_dt);
```

State navigation

States will be navigated using setState, pauseState and unpauseState functions. This allows for each part of the game to be stored in its own state and not need to interact with other states at-all other than to load another state and end its own. States will be owned and controlled by the main game where the current states update function will be used to update any game objects or components that are modified. The setState function will work by first ending the current active state and then initializing the provided new state. Pausing a state will work by storing the pointer to the current state before initializing a new state, when the new state is ended unpauseState can be called which then sets the paused state's pointer back to be the current state. If there is already a paused state pointer stored then the game will not allow pause state to be called again as this would cause the pointer to the previously paused state to be lost.

Components

Transform component

Contains the 2D position, rotation (in radians) and the 2D scale data. This component will be added to every entity.

SpriteComponent

Contains the 2D relative position, relative rotation (in radians), 2D relative scale, the origin to apply transformations around, the colour, source texture id, Z order, source texture rectangle and a flag to control if the component is rendered. The relative transform properties are relative to the entity's transform component that this component will be added to.

SpriteAnimComponent

Contains the 2D relative position, relative rotation (in radians), 2D relative scale, the origin to apply transformations around, the colour, source texture id, Z order, the source rectangle of each frame in the animation, the playback rate (in seconds) of the animation, a flag to control looping of the animation and three attributes not intended to be edited by client users of the system: the current frame index the animation is on, a completion flag and the time that the animation last moved to the next frame. The relative transform properties are relative to the entity's transform component that this component will be added to.

TextRenderComponent

Contains the 2D relative position, relative rotation (in radians), 2D relative scale, the text string to render, the id of the font to render the text with, the origin to perform transformations around, the colour of the text and the Z order. The relative transform properties are relative to the entity's transform component that this component will be added to.

HUDImageComponent

Contains the same data as the SpriteComponent but is used by the HUDImageRendererSystem to render images to the game's HUD.

HUDTextComponent

Contains the same data as the TextComponent but is used by the HUDTextRendererSystem to render text to the game's HUD.

Data

The game will use a data driven approach where each of the systems and components can create their own data structures which can then be passed to other systems or components, other systems can also get references to the data and read or write to it when needed. An example of this is InputData which is used to store keybindings, these keybindings can be set to any directX keycode and then passed to another system to be interpreted. The main use in the game is to store the key bindings for each player which allows them to be edited from various other systems via editing the data inside the struct.

Component Data

The components' data is created by the component manager as a single block of memory making it quicker for the game / engine to use these, by making it more likely that the CPU will find the next value, which is going to be accessed by a system, inside the CPU cache. To achieve this components should avoid having vectors in them, because the allocation and reallocation of vectors in memory would disrupt this design of all component data being held in a single block. In most cases the use of vectors in components can be avoided by defining a maximum value that the components may need to contain.
This also brought up the question whether the use of strings is going to disrupt the block of memory, and whether replacing them with char arrays would further optimize the engine.

Systems

Audio Manager

The Audio manager is a system which will be used to control all things audio in the project. It will handle the creation and deletion of audio soundbytes and will be owned by the game itself. The audio manager will save memory by only loading the raw wave data of an audio file once and then using DirectX sound effect instances of the loaded files, this will allow for multiple instances of a sound to be played without needing to load the raw data into memory more than once. To help prevent sound files from being loaded more than once the manager will store the loaded file into a vector and then in a separate vector, it will store a pair of the index of the of the file in the vector and the system file path of the file loaded, when loading a new audio file the manager will first check whether that file path has been loaded before and if so, return the index of that loaded file rather than loading it again.

Rendering

Sprite renderer system

The sprite renderer system operates on entities that are composed of a transform component and a sprite render component. The goal of the system is to draw sprites in the game world. The system will provide two public methods to client code: init and render. Init initialises the system, setting its system signature with the ecs.

The render method iterates over the system's tracked entities and attempts to render them. The render method will early exit if the entity's sprite component's render flag is set to false or if the texture id is invalid. Once these checks pass the sprite will be drawn into the sprite batch passed into the render call.

Sprite anim renderer system

The sprite anim renderer system will operate on entities that are composed of a transform component and a sprite anim component. The goal of the system is to draw animations in the game world. The system will provide three public methods for client code to use, init, tick and render. The system stores the elapsed time that has passed since the engine's game loop started.

The init method registers the system's signature with the ecs.

The tick method adds the delta time since the last tick to the elapsed time. Each entity's sprite anim component's properties are used to update the status of each animation by checking if the animation needs to be progressed to the next frame and incrementing the component's frame index if necessary.

The render method will have early out checks to see if the sprite anim component has frames and a valid texture id. Once these checks are passed a sprite with the animation's current frame's source rectangle will be drawn to the sprite batch passed to the render call.

Text renderer system
The text renderer system will operate on entities that are composed of a transform component and a text render component. The goal of the system is to draw text in the game world. The system will provide two public methods, init and render.

The init method will register the system's signature with the ecs.

The render method will iterate over the system's tracked entities and first perform early out checks to ensure that the entity's text render component has a valid font id. Once these checks are passed, a string will be drawn with the font into the sprite batch passed to the render call.

HUD image renderer system
The HUD image renderer system will operate on entities that are composed of a transform component and a HUDImageRenderComponent. The system will perform the same operations as the sprite renderer system however, it will draw to the game's HUD batch instead of the game world.

HUD text renderer system
The HUD text renderer system will operate on entities that will be composed of a transform component and a HUDTextRenderComponent. The system will perform the same operations as the text renderer system however, it will draw to the game's HUD batch instead of the game world.

Tools
Level editor

# Objective

The level editor is a tool to allow designers to build and edit levels for the game. It should be able to output new content that can be run by the game.
The level editor should provide tools to manage entities in the level and the components that those entities have as well as create and edit tile maps and load and save levels.

The level editor should also manage the asset pipeline by importing commonly used data file formats from existing software packages, manipulating the data and exporting optimised data file formats supported by the game.

Levels will be serialized in a human readable YAML file format with the .LEVEL file extension.

# .LEVEL file
The level file format will store information about the level in a human readable format. A level consists of entities, their components and those component's properties. These elements will be stored in the following format inside the level:

```
Level : name
Entities:
        Entity: id
                Component:
                        Property: value
                        Property: value
                        …
                ...
        ...
```

References

Morlan, Austin. (2019) A Simple Entity Component System (ECS) [C++]. Code Contact LinkedIn [blog]. 25 June. Available from: https://austinmorlan.com/posts/entity_component_system/ [Accessed 02 March 2021].

GitHub (2021) imgui. Available from: https://github.com/ocornut/imgui [Accessed 17 April 2021].

GitHub (2021) *yaml-cpp*. Available from: https://github.com/jbeder/yaml-cpp [Accessed 17 April 2021].