Introduction

This paper serves as a Quickstart guide and technical documentation for L3beh, an Entity-Component-System based game engine.

The first section of this paper contains a summary of what an Entity-Component-System entails. There is also a section dedicated to build instructions and requirements for the way that I have specifically been building/testing the engine and the things I found I needed to do to get things up and running at first.

The second section is a Quickstart Guide which provides a step-by-step set of instructions that help users of the engine get up and running to implement their own interactive 3D games/applications. This includes instructions on how to create Components, Systems, Worlds, and World Config Files, along with example code that can be run as a basic demo. There is also two notes on debugging technique and performance that users of the engine could find useful.

The third section of this document contains the technical documentation which describes much of how the engine works under the hood. This section begins with a discussion of every individual code file that makes up the infrastructure/scaffolding that allows the Entity-Component-System to run; with specific details provided about individual functions and variables and what they are for. This section also includes discussion of what is included in the core engine, such as the Systems and Components. The Systems include – Physics, Collision, Render, and Transform. An Appendix is included that contains information about the included Components and their corresponding world config file formatting.

The last section has discussion on the engine's potential usage as a teaching tool in a classroom setting as well as some ideas for future work that can be done to modify/improve the engine itself.

Overall Architecture

Entities - These are the individual "things" that are in the game itself and are a composition of discrete components. The entities within the code itself are represented simply as Integer IDs, and these integer IDs are used as handles to the ComponentManager to get components that pertain to the specified entity

Components - Components are meant to be very simple structs that encapsulate some state data (for example the Transform component stores the position, rotation, and scale of the entity). An entity that is specified to have a component will have an instance of said component be created and stored/kept track of by the engine.

Systems - Systems are where game logic is to be stored. Systems are to be provided a list of required components (represented as a bitfield) that allow the System to decide which entities it is allowed to act upon. A System can have multiple acceptable required component bitsets (For example, the CollisionSystem can accept entities with SphereCollider, Transform, and Physics OR BoxCollider, Transform, and Physics).

- Each System has an addEntity() function which you can override to modify how the list of entities it accepts is organized.
- There is also a removeEntity() function that you can override to cleanup entities that no longer fit within the system's component requirements.
- The init() function is called once at the beginning of the game and can be overridden to do initial setup of entities

Build Notes and Source

This project is hosted on GitHub at this link: https://github.com/The-O-King/L3beh. The version of the engine being described at the time of writing can be found as a GitHub release with the tag v1.0-Capstone.

This engine was designed, built and tested on Windows. In order to be able to build the engine/game on your own Windows machine these are some of the prerequisites that need to be taken care of before doing so:

- Built using MinGW-W64
    - All future instructions assume MinGW-W64 is installed at the root of the C:\ drive
- How I installed GLFW Binaries for Windows:
    - https://stackoverflow.com/questions/12886609/setting-up-glfw-with-mingw
    - Had to put GLFW/glfw3.h into C:\MinGW64\mingw64\x86_64-w64-mingw32\include

    - Had to put *.a static libraries into C:\MinGW64\mingw64\x86_64-w64-mingw32\lib
    - Put the glfw3.dll in System32 folder

- Since we are using the GLFW dll and linking statically, at the top of main.cpp you must define the macro
  #define GLFW_DLL
- Had to install GLEW to get support for openGL versions higher than 1.1 for Windows
    - http://glew.sourceforge.net/install.html
        - To do so you need to build it yourself for use with MinGW64. They only have prebuilt binaries available for use with Visual Studios and Visual C++. Here are some instructions I followed: https://www.opengl.org/discussion_boards/showthread.php/198730-How-do-I-build-GLEW-for-mingw-w64-using-cmake-on-Windows?p=1283379&viewfull=1#post1283379
    - But I have included the built binaries for MinGW64 in the GLTools/glew-2.1.0.lib folder
- Follow a similar pattern as with GLFW for install of GLEW:
    1. Place the include/GL folder in the C:\MinGW64\mingw64\x86_64-w64-mingw32\include
    2. Place the *.a static libraries into C:\MinGW64\mingw64\x86_64-w64-mingw32\lib
    3. Place the glew32.dll in System32 folder
- Since we are using the GLEW dll and linking statically, at the top of main.cpp you must specify the macro
    - #define GLEW_STATIC

With everything installed on Windows, you should be able to build the engine/game via this command:

```
g++ -std=c++17 -o compile/outRelease core/*.cpp core/systems/*.cpp *.cpp \
-lglew32 -lopengl32 -lglfw3  -lglfw3dll
```

Quickstart Guide

The usage of this game engine centers around the creation of System, Components, and a World Config File. The best way to view how using this game engine works is to follow along a quick example; creating basic player movement functionality. The code for this example can be found at the root of the project in the quickstart/ folder. To run the game built by this guide, delete all the individual files found at the root of the project and move all the files in quickstart/ out to the root of the project – the folder structure should look like this:

```
.vscode/
compile/
core/
GLTools/
graphics/
quickstart/
CustomComponents.hpp
main.cpp
MyWorld.cpp
MyWorld.h
PlayerMovementSystem.cpp
PlayerMovementSystem.h
worldConfig.txt
```

Component Definition

We will begin by defining a custom component. Components, as discussed in the overall architecture section, are simply encapsulations of data; no game logic is attached. They only store a portion of the current state that an entity is currently in.

To implement a PlayerMovementComponent, we should think about what information is needed to know the current state of the player's traversal abilities. We shall have variables for things like walk speed, run speed, jump force, jumping state, and running state.

Let's look at how we can implement this custom component. The best practice I have found for adding additional custom components is to add a new file called CustomComponents.hpp and defining all of these components within it. Here is what CustomComponents.hpp would look like with the implementation of the PlayerMovementComponent:

```
#ifndef CUSTOMCOMPONENT_HPP
#define CUSTOMCOMPONENT_HPP

struct PlayerMovementComponent{
    float walkSpeed = 3.0f;
    float runSpeed = 5.0f;
    float jumpForce = 500.0f;
    bool isJumping = false;
    bool isRunning = false;
};

#endif
```

System Definition

The next step will be to actually define the game logic that will use the component we just defined. Systems, as discussed in the overall architecture section, are where the actual game logic lies. The creation of custom systems begins by creating a subclass of the System class. Let's create a PlayerMovementSystem.h and PlayerMovementSystem.cpp that will take advantage of our newly defined PlayerMovementComponent:

```
#ifndef PLAYERMOVEMENTSYSTEM_H
#define PLAYERMOVEMENTSYSTEM_H

#include "core/system.h"
#include "CustomComponents.hpp"
#include "core/glm/glm.hpp"
#include "core/components.h"
#include "core/world.h"
#include "core/input.h"

class PlayerMovementSystem : public System{
    private:

    public:
        PlayerMovementSystem(World* w);
        void init() override;
        void update(float deltaTime) override;
};

#endif
```

Let's walk through what is happening in PlayerMovementSystem.h.

- Includes:
    - **core/system.h** – This line is what allows us to create a subclass of the System class that is part of the core game engine
    - **CustomComponents.hpp** – This line allows us to have access to our newly defined PlayerMovementComponent
    - **core/glm/glm.hpp** – This line gives us access to GLM, our linear algebra library, for things like vectors
    - **core/components.h** – This line gives us access to components that are included with the core engine, such as the Transform component
    - **core/world.h** – This lets us use the world interface to give us access to GetComponent functionality we will use to access the component instances for a given entity
    - **core/input.h** – This is what gives us access to the input functionality that will let us poll for key presses on the keyboard

- **`PlayerMovementSystem(World* w);`** - This line is the forward definition of our constructor
- **`void init() override;`** - This line is the forward definition of the init function that we can override with custom game logic that will run once at the very start of the game
- **`void update(float deltaTime) override;`** - This line is the forward definition of the update function that we can override with game logic that we want to have run once every frame the game is running

Here is the definition of PlayerMovementSystem.cpp:

```cpp
#include "PlayerMovementSystem.h"

PlayerMovementSystem::PlayerMovementSystem(World* w){
    mWorld = w;
    componentSignature main;
    main[type_id<TransformComponent>()] = 1;
    main[type_id<PlayerMovementComponent>()] = 1;
    main[type_id<PhysicsComponent>()] = 1;
    neededComponentSignatures.push_back(main);
}

void PlayerMovementSystem::init(){

}

void PlayerMovementSystem::update(float deltaTime){
    for (int e : entities){
        PlayerMovementComponent& pc = mWorld->getComponent<PlayerMovementComponent>(e);
        TransformComponent& tc = mWorld->getComponent<TransformComponent>(e);
        PhysicsComponent& phys = mWorld->getComponent<PhysicsComponent>(e);
        glm::vec3 forward = glm::vec3(0, 0, -1);
        glm::vec3 left = glm::vec3(-1, 0, 0);
        glm::vec3 up = glm::vec3(0, 1, 1);
        float speed = 0;
        if (Input::getKey(GLFW_KEY_LEFT_SHIFT)){
            speed = pc.runSpeed;
            pc.isRunning = true;
        }
        else{
            speed = pc.walkSpeed;
            pc.isRunning = false;
        }
        if (Input::getKey(GLFW_KEY_W)){
            tc.position += forward * speed * deltaTime;
        }
        if (Input::getKey(GLFW_KEY_S)){
            tc.position -= forward * speed * deltaTime;
        }
        if (Input::getKey(GLFW_KEY_A)){
            tc.position += left * speed * deltaTime;
        }
        if (Input::getKey(GLFW_KEY_D)){
            tc.position -= left * speed * deltaTime;
        }
        if (tc.worldPosition.y < .1){
            if (Input::getKeyDown(GLFW_KEY_SPACE)){
                phys.sumForces += up * pc.jumpForce;
                pc.isJumping = true;
            }
            else{
                pc.isJumping = false;
            }
        }
    }
}
```

Let's also look at what is going on in this code snippet:

- **`PlayerMovementSystem::PlayerMovementSystem(World* w)`**
    - This constructor is what initializes the entire system. We pass in a pointer that corresponds to the World (or game instance) that this system belongs to, and make sure to set the systems mWorld variable to it. This will allow us to use the component access functionality that is tied to the world (such as GetComponent)
    - After this we see where we define the component signature that an entity must match in order to be considered a part of this system. We initialize an empty componentSignature bitfield (in this case called "main"), and then using the type_id<>() function we can get the indices that correspond to the types that an entity is required to have; and flip those bits in the componentSignature on. Lastly, we add the finished componentSignature to the list of needed componentSignatures so that the engine can check entities against this newly created componentSignature.
- **`void PlayerMovementSystem::init()`**
    - There is no functionality we need to implement at the very beginning of the game, so we left this function empty
- **`Void PlayerMovementSystem::update(float deltaTime)`**
    - Here is where the game logic lies. We first start off by iterating over all entities that have been approved to be a part of this system. The entities variable is an std::set that is included in the base System class and by default will contain all entities that match the needed component signatures
    - For each of these entities, we will immediately gain access to all the state information we need to implement our game logic. This takes the form of requesting all the necessary components that an entity has from the current World context.
        - **`mWorld->GetComponent<ComponentType>(entityID)`** is a template function that will return a reference to the ComponentType instance for an entity (specified by entityID)
    - Next, we setup some basic vectors for us to use in calculating our directionality
        - GLM includes types for many different vectors and matrix sizes. Here we are using glm::vec3 which is a 3-dimensional vector
    - The if-else ladder is where we do our keyboard button checks
        - The Input namespace provides multiple functions that allow you to check the state of a current key press
        - **`Input::getKey(keycode)`** will return true for every frame that a key is held down
        - **`Input::getKeyDown(keycode)`** will return true only on the first frame that a key is pressed down
    - Within different conditions of the if-else ladder, we can see how the components themselves are used/updated

- We can see that we check whether the user is holding down the shift key and set the isRunning field of the PlayerMovmentComponent accordingly
- We then check to see which of the WASD keys is pressed and update the TransformComponent's position field accordingly
- Lastly, we check to see what the current height of the player is and if the Space key is pressed to decide if we need to apply a jumping force to the player's PhysicsComponent (via the sumForces variable) as well as setting the isJumping field of the PlayerMovementComponent

And with that, we have implemented a custom System and Component! You can repeat these steps for as many Components and Systems as you need – and don't forget that there does not need to be a one to one mapping of Components to Systems; ideally in many cases you will want to try to reuse Components that already exist that store state you need across multiple Systems, for instance the PlayerMovementComponent has information about whether or not the player is jumping, which could be useful in a System that is trying to implement a flying mechanic that needs to know whether the player is in the air before they can take off for example.

World Subclass Definition

Now that we have these custom Components and Systems let's use it in a game! To do so, the first thing we will need to do is define a subclass of the World class. The World class we mentioned in passing previously when discussing Systems, but concretely the World class is what stores all the state of the entire game – which Components and Systems are registered to be used within the game itself, entity creation/destruction, the ComponentManager that deals with managing Component instances.

Two things must be done to implement a subclass of the World class – registration of all Components and Systems you want the game itself to have, and a definition of customWorldGen() that will specify how to parse in custom Components you have specified from a world config file. Here is an example of a World Subclass header file, MyWorld.h:

```
#ifndef MYWORLD_H
#define MYWORLD_H

#include "core/world.h"

class MyWorld : public World{
    public:
        MyWorld();
        bool customWorldGen(int entityID,
                            std::string command,
                            std::istringstream& data) override;
};

#endif
```

This will basically be the format for all of your custom World subclass header files. Here we see the forward declaration of the two functions we will need to implement – the constructor and the customWorldGen function.

Let's take a look at the MyWorld.cpp:

```cpp
#include "core/runner.h"
#include "core/components.h"
#include "core/systems/TransformSystem.h"
#include "core/systems/PhysicsSystem.h"
#include "core/systems/CollisionSystem.h"
#include "core/systems/RenderSystem.h"
#include "PlayerMovementSystem.h"
#include "MyWorld.h"

MyWorld::MyWorld(){
    mSystems.push_back(new TransformSystem(this));
    mSystems.push_back(new RenderSystem(this));
    // Add custom Systems here
    mSystems.push_back(new PlayerMovementSystem(this));
    // Stop adding custom Systems here
    mSystems.push_back(new CollisionSystem(this));
    mSystems.push_back(new PhysicsSystem(this));
    mComponents.registerComponent<TransformComponent>();
    mComponents.registerComponent<RenderComponent>();
    mComponents.registerComponent<PhysicsComponent>();
    mComponents.registerComponent<ColliderComponent>();
    mComponents.registerComponent<BoxColliderComponent>();
    mComponents.registerComponent<SphereColliderComponent>();
    mComponents.registerComponent<CameraComponent>();
    mComponents.registerComponent<PointLightComponent>();
    mComponents.registerComponent<DirectionalLightComponent>();
    // Register Custom components here
    mComponents.registerComponent<PlayerMovementComponent>();
}


bool MyWorld::customWorldGen(int entityID,
                             std::string command,
                             std::istringstream& data){
    if (command == "PlayerMovement"){
        PlayerMovementComponent pc;
        data >> pc.walkSpeed >> pc.runSpeed;
        addComponentToEntity<PlayerMovementComponent>(entityID, pc);
    }
    return true;
}
```

- **`MyWorld::MyWorld()`**
    - This is the constructor, and is where you MUST remember to register all the Systems and Components that your game needs. The order in which Systems are added DOES MATTER, since that is the order in which they will be iterated along during the game loop. In the code example above, I found this to be the proper order for Systems to be added to the game world. Adding Systems is done by pushing back an instance of them into mSystems, an std::vector within the base World class that is used for storage and iteration over the Systems themselves
    - Component registration is a little less strict. Order does not matter at all, so you just have to make sure to remember to mComponents.registerComponents<ComponentType>() as this is what assigns a component a type_id that can be used in the future to specify componentSignatures
- **`Bool MyWorld::customWorldGen(int entityID, std::string command, std::istringstream& data)`**
    - This function is where you get to define how to parse lines in the world config file that correspond to custom Components that you have defined. The config file itself is parsed one line at a time; with a line beginning with the name of the Component to be added, followed by all the data to be used to set the initial state of said Component for a given entity
    - What is passed into this function is
        - `entityID` of the entity that is currently being instantiated
        - `command`, which is the string corresponding to the name of the Component that is to be added to the entity;
        - `data`, an istringstream of all the following data that you need to decide how to parse to setup the component
    - What this function boils down to is a large if-else ladder that checks whether the command string matches a known Component type
    - If there is a match, you create an instance of that Component, and using the >> operator, read in each value from the data istringstream one at a time to set the component's fields properly
    - Once this is complete, you call the **`addComponentToEntity<ComponentType>(entityID,ComponentInstance)`** function in order to add the newly created component to the given entity
    - As you add more custom Components to the engine, be sure to return here and define the necessary parsing step for it to properly be read from a world config file!

World Config File Definition

Now that we have the custom World class defined, we can create a very basic world config file. These are simply text files that contain the initial state of the game world – an example of which is this; worldConfig.txt:

```
Entity Player
PlayerMovement 3 6
Physics 1 1 1 .2   0 0 0  0 0 0
SphereCollider 0   0 0 0    1
Transform 0    0 0 0   0 0 0  1 1 1
Camera 1 90
/Entity

Entity Light
Transform 0    -1 0 -5    0 0 0  1 1 1
PointLight 4 1 0 0
/Entity

Entity Floor
Physics 0 0 1 .2   0 0 0  0 0 0
BoxCollider 0    0 0 0   5 .5 5
Transform 0    0 -1.5 0   0 0 0  10 1 10
Render cube.obj cube.jpg 1 1 1 1 1 1 2
/Entity
```

Here we see that our game will start off with 3 entities. Individual entites are wrapped by the Entity and /Entity tags. Within the tags are the Component definitions themselves. The first entity we can see has PlayerMovement, Physics, SphereCollider, Transform, and Camera Components. We can also see that after the name of each component, we have the initial state of the variables within each of these components, which gets parsed by the worldGen and customWorldGen functions in the World class. Definitions for all the Components included in the core engine are defined in later sections.

main.cpp Definition

Finally, in order to actually get everything to run, we need to define a main.cpp file which will start the entire game itself:

```
#include "core/runner.h"
#include "MyWorld.h"


int main(){
    MyWorld newWorld;
    return run(newWorld, "worldConfig.txt");
}
```

All we need here is to create an instance of our custom World class and pass it into the run() function along with the config file we specified. The run function is a part of the core/runner.h file, and contains the main game loop that will actually iterate over all of our Systems, as well as window initialization via GLFW to actually allow us to present the graphics on the screen

With everything setup properly, you should be able to build this example game by going to the command line and running:

```
g++ -std=c++17 -o compile/outRelease core/*.cpp core/systems/*.cpp *.cpp \
-lglew32 -lopengl32 -lglfw3  -lglfw3dll
```

Debugging Note:

Due to the nature of the engine not having a proper GUI, the best way I have found to be able to debug the engine for logic errors (primarily during the definition of the update() function of a custom System) is to place an if statement within the update() function of a System that checks for some key having been pressed, and having a throwaway line of code within the if statement (such as the initialization of an integer) that you can set a breakpoint at:

```
void TestSystem::update(float deltaTime){
    if (Input::getKeyDown(GLFW_KEY_EQUAL)){
        int temp = 0;
    }
}
```

So with a breakpoint set at the line "`int temp = 0;`" when you press the equals key on the keyboard you will be able to pause execution of the game at any arbitrary frame and check variables/Component state as needed.

Performance Note:

The engine does not have any built in form of logging at this time so the way I have been logging state is simply using std::cout. This can be seen primarily in the frame times that are being printed out to the console for performance monitoring purposes. While this can be a convenient way to do additional debugging, printing to the console window using std::cout every frame in this manner incurs a sizeable performance hit. The removal of the std::cout in core/runner.h will have a noticeable impact on the game's framerate.

Moral of the story – using std::cout for printf debugging can be handy however remember to remove these calls when trying to do performance testing or finalizing code in order to see the true performance of the game/application itself without the console printing overhead.

Core Engine Details

runner.h Contains the main game loop and initial setup for openGL/GLFW for window creation Also computes the frame times

- You may need to come here to update the deltaTime that is passed into the update() function of Systems since it is currently fixed at 1/60s (for 60 frames per second). Faster/slower computers will have different framerates and as such this number will need to be adjusted accordingly (common framerates include 30, 60, 120, 240, 480). If not adjusted properly, the game will appear to either run too fast or two slow.

input.h/.cpp This is the code that provides Input services for the programmer. It uses the GLFW input API to get key/mouse data from the currently active window. For checking generating keyDown, keyUp, and keyHold events, a boolean array (keyState) is used to store the last state of the key (which is updated every frame using a callback provided by GLFW)

component_utils.h A file that contains utilities used to create metadata and functions for components. This includes:

- **componentSignature**: a typedef for the bitset used by systems to verify if an entity belongs to a system
- **invoker, make_invoker, and erase_at_index**: Functions used to create type-erased function signatures for deletion functions needed to remove an entity and it's corresponding components from the game world. More info can be found from this stackOverflow question and comments (The_O_King & Yakk - Adam Nevraumont, 2018)
- **generate_type_id()** and type_id(): Functions that are used to make it easy for the programmer to get the ID of a registered component in the game engine. This uses templated functions with static variables that generate a new ID every time a new component type is registered and will always return that value when provided the same component type.

componentManager.h/.cpp This code has all the logic that is used to manage component data. Before being used, component types need to be registered with the component manager (this is done using the world object though). Once registered, the component type gets an entry in a vector of vectors which stores the actual component instances (see the visualization below)

ComponentManager componentHolder				
TransformComponent	data...	data...	data...	data...
PhysicsComponent	data...	data...		
RenderComponent	data...			
BoxColliderComponent	data...	data...	data...	

The component instances themselves will as a result be stored linearly in memory, which should help a lot with caching! In order to access components for a specified entity, we keep another vector of dictionaries that map entityID -> index_in_componentHolder.

Special care is taken when deleting entities (and deleting their specified component instances as a result) to update subsequent entities whose component instances have indexes larger than the one deleted; since everything is contiguous in memory a deletion will shift all subsequent components down, thus invalidating the handle we currently stored for entities that correspond to those component instances.

A single world object has a single ComponentManager instance which is used to register/store the components of entities within that world. Functions that provide this functionality are:

- **registerComponent()**: Gives a new component type a specific ID as well as creating a vector in the componentHolder to store component instances
- **addComponent()**: A templated function that allows the programmer to add a specified component type to a given entity. This function can be called by providing the entity ID and an already initialized instance of the component type you want to add, or it can be called without a component instance, and will instead will add a component instance with default parameters.
- **setComponent()**: Allows you to replace an already existing component that an entity has with a new instance of the component that you pass in.
- **removeComponent()**: Removes a component from an entity entirely and removes it from corresponding systems as well.
- **destroyEntity()**: A function which removes an entity entirely from the world; deleting all of it's corresponding components and removing it from all systems. This will also remove the entity ID from the living_entities set in the world. The actual destruction of the entity occurs at the end of the frame this function is called in.
- **getComponent()**: A templated function that returns a reference to a specified component for an entity given it's entity ID. This allows the programmer to read or update that component's information

components.h This file includes the definitions for all the component types that the core engine include. These component types are defined as structs of just data. There are also some additional data structures and functions defined in this file that support the functionality of these components (such as the getRotation/getOrientation/getTransformation functions). Definitions for the fields in these functions can be found in Appendix A.

system.h This code provides the interface that a programmer needs to implement in order to create new systems for the engine. By default, a System has these member variables:

- **mWorld**: Pointer to the World object that this system belongs to
- **neededComponentSignatures**: A vector of all the componentSignatures that could allow an entity to fall within this system's control
- **entities**: A set that is provided by default to store all the entity IDs that fall within this system

These function prototypes are provided and need to be overridden:

- **init()**: Called just before the first frame of the game starts. Is a good place to do some initialization of values for entities
- **update()**: Called every frame of the game, this is where most game logic will sit. The deltaTime between the last frame and the current frame is also provided to allow for framerate independent logic to be implemented.

Along with these functions, additional functions that can be overridden if needed include:

- **addEntity()**: Function which is called whenever a new entity fits the needed componentSignature of the system. By default, the function adds the entity to a set of all entities that fit within the system. Overriding this function can allow you to better organize components that fit within the system once they are added.
- **removeEntity()**: Function that is called when an entity needs to be removed from a system because it no longer has the required componentSignature. This will most likely need to be overridden if you overrode the default addEntity() function in order to remove the entityID from a custom organization of entities that you implemented.

Lastly, one function provided by default that does not need to be changed:

- **getNeededComponents()**: Returns a vector of componentSignatures that this system can accept.

When implementing a new System, the workflow looks something like this:

1. Create a new class (for example "DeathFlowerSystem") that inherits from the System class
2. Implement a constructor which takes in a World pointer as an argument; set the mWorld member variable of the system equal to that world and then define all the required

componentSignatures that you need entities to have, adding them to the neededComponentSignatures vector

3. Override the addEntity(), removeEntity() functions if you have a specific organization for systems in mind
4. Implement the init() and update() functions with the necessary game logic you want entities in this system to exercise

world.h/.cpp This file contains the definition of the World base class. This class is designed to manage the entire state of the game and provide an interface for creating/destroying entire entities as well as creating/destroying certain components. By default, a World has these member variables:

- **currID** An integer representing the next available entity ID
- **mWindow** a pointer to the GLFWwindow that corresponds to the game itself
- **mSystems** A vector which contains all of the Systems that are a part of the game itself. This vector gets iterated over in the game loop to call the update() function of each System once per frame
- **mComponents** An instance of the ComponentManager that corresponds to this specific game
- **liveEntities** An unordered_map which maps entityIDs (integers) to their corresponding componentSignature
- **entitiesToDestroy** A set that contains entityIDs (integers) of entities that will be destroyed at the end of the current frame

The World base class also implements these functions that act as an interface for entity and component creation/destruction:

- **baseWorldGen()** Called on the initialization of the World and reads in the given config file to instantiate the initial state of the game world. It automatically create a root entity that represents the game world itself before adding the entities specified in the config file
- **createEntity()** A function that allows the user to pass in an entity configuration as a string to have it instantiated with all the specified components. If this function is called with no arguments then it will create an entity with no components at all. The function will always return the entityID of the newly created entity
- **destroyEntity()** This function is to be used in game logic to tell the World to remove the specified entity at the end of the current frame. This function adds the passed in entityID

to the entitiesToDestroy set that will be iterated over at the end of the frame for destruction. This function also checks that the passed in entityID is a valid live entity

- **destroyEntities()** This is the function that is called at the end of every frame, in which we iterate over the entitiesToDestroy set and delete an entity from our game. This is done by taking the specified entity's componentSignature and iterating through it to remove all component instances that corresponded to this entity. The function will also destroy all entities that are children to this entity in the same fashion that the original parent entity was destroyed
- **getSystems()** Returns a reference to a the mSystems vector
- **addComponentToEntity<T>()** These functions allow you to add a component of a specified type T to a given entityID. You can optionally pass in a component instance that you have prepopulated the variables for. If you do not pass in a component instance, a component instance with it's default values will be added to the entity
- **removeComponentFromEntity<T>()** This function will remove a component of a specified type T from a given entityID. The removal occurs instantaneously and will remove the entity from all Systems that it no longer is qualified to be in
- **getComponent<T>()** This function returns a reference to a component of type T corresponding to the given entityID
- **setComponent<T>()** This function allows you to completely replace the state of an existing component of type T for a given entityID with the state of the component instance passed into the function call

Core Engine Systems

Physics

- **Required Components:**
  - Transform, Physics
- **Summary:** The physics system integrates the rigidbody kinematic equations of motion for both rotation and translation.
- **Details:** This system uses the velocity verlet integration technique for solving for the translational motion:

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \vec{v}(t)\,\Delta t + \frac{1}{2}\,\vec{a}(t)\Delta t^2$$
$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{a}(t) + \vec{a}(t+\Delta t)}{2}\Delta t$$

and quaternion Euler integration to solve for the rotational motion:

$$\mathbf{q}(t+\Delta t) = \mathbf{q}(t) + \frac{1}{2}\,\boldsymbol{\omega}(t)\mathbf{q}(t)\Delta t$$
$$\boldsymbol{\omega}(t+\Delta t) = \boldsymbol{\omega}(t) + \frac{\boldsymbol{\alpha}(t) + \boldsymbol{\alpha}(t+\Delta t)}{2}\Delta t$$

Where $\mathbf{q}(t)$ is the quaternion representing the current rotation and $\boldsymbol{\omega}(t)$ is the angular velocity vector that is converted into a quaternion for this specific computation.

Collision

- **Required Components:**
  - Transform, Physics, BoxCollider
  - Transform, Physics, SphereCollider
- **Summary:** The collision system implements collision detection and resolution techniques for box and sphere colliders
- **Details:** When entities are added to this system, their inverse inertia tensor is calculated since this is dependent on the sphere radius, box dimensions, and mass of the collider.

  This system's update function can be viewed in two phases – the collision detection phase and the collision resolution phase.

  The collision detection phase does a pairwise test between all entities within the scene. Specifically, these tests are checking whether the two entities are overlapping or not. First, we define a collision detection jump table that stores pointers to various implementations of collision detection algorithms (called collisionTestTable in CollisionSystem.cpp), a visual representation of the table and currently supported collision shape is below:

Box vs Box	Box vs Sphere
Sphere vs Box	Sphere vs Sphere

Selection of which function to call is done by indexing into collisionTestTable using the type variable found in the ColliderComponent corresponding to each entity. Each of these functions accepts the entityIDs of the two entities being tested, their corresponding TransformComponent, a collisionInfo struct, and the pointer to the world this system corresponds to. The collisionInfo struct that is passed in will specify whether a collision actually occurred, and if so, will also contain all the necessary information for doing collision resolution between these two entities. Specifically, the collisionInfo struct contains:

- **`int entity1, entity2`** – The IDs of the entities involved in the collision
- **`float distBetweenObjects`** – The largest penetration distance between the entities
- **`glm::vec3 normal`** – The vector representing the normal of the separating plane that the collision occurred on
- **`glm::vec3 Tangent[2]`** – an array of 2 vectors that represent the two directions that represent the tangent plane
- **`contactPoint points[8]`** – an array of size 8 that represents all the contact points that are involved in the collision
- **`int numContacts`** – the total number of contact points involved in the collision

The way we know that a collision has actually occurred is checking the numContacts variable is greater than 0; and if so we know that a collision has occurred.

The specifics of the implementation of each of these collision shape pair tests are as follows:

- Box vs Box: Separating Axis Theorem, which checks to see whether there exists a separating plane between the two BoxColliders, and generates the contact points of the collision if there exists no separating plane via Sutherland-Hodgeman Clipping the incident face to the planes that surround the reference face. Additional details of this algorithm can be found in Dirk Gregorious's GDC presentation (Gregorious, 2015)
- Sphere vs Box and Box vs Sphere: This test becomes an Axis Aligned Bounding Box vs Sphere (or the converse) by transforming the sphere into the coordinate space of the Oriented Bounding Box. Then we find the closest point on the AABB to the sphere's center and check whether that distance is less than the sphere's radius and if so, a collision has occurred and a single contact point is computed based on the line from the center of the sphere to the closest point on the AABB. More details can be found in the 3D Collisions gitbook (G, 2016)

- Sphere vs Sphere: This test simply checks the distance between the centers of the two spheres and sees whether the distance between the centers is less than the sum of the radii of the spheres themselves, and if so generates the single contact point on the line defined by the sphere centers.

The next step is to update the persistentCollisionData map used for the sequential impulse solver with the new collisionInfo struct, or delete an existing one if there is no longer a collision between these two objects (this situation occurs if in the previous frame the objects were colliding but in the current frame they are no longer colliding). Alongside the persistentCollisionData update, there is also an update to the collision history lists contained within the CollisionComponent. These are lists that represent the current collision state of an entity, and let the user know whether an entity has just collided (collisionEnter), has been colliding (collisionStay), or has just stopped colliding (collisionExit). These 3 collision histories are represented as sets of integers that store the set of entityIDs that this current entity is interacting with.

If a collision has occurred between two entities, we begin the process of collision resolution via the application of impulses that are designed to force the two rigidbodies apart. This process is based on the Sequential and Accumulated Impulses, an iterative solution to the linear system of equations that represents a non-penetration constraint between two rigidbodies. This solver is based on the work done by Erin Catto's work on the Box2D physics engine (Catto, 2008) and Randy Gaul's work on Qu3e physics engine (Gaul, 2017).

The primary goal of the Sequential Impulse solver is to compute the magnitude of the impulse in order to separate the two colliding entities. An impulse that is too strong will cause jittering and instability, while a weak impulse will see objects sink into each other. The equation that represents the magnitude of the normal impulse $\boldsymbol{P}$ is:

$$\boldsymbol{P} = P_n \boldsymbol{n}$$

$$P_n = max\left(\frac{-\Delta\bar{\boldsymbol{v}}\cdot\boldsymbol{n}}{k_n}, 0\right)$$

$$\Delta\bar{\boldsymbol{v}} = \bar{\boldsymbol{v}}_2 + \bar{\boldsymbol{\omega}}_2 \times \boldsymbol{r}_2 - \bar{\boldsymbol{v}}_1 - \bar{\boldsymbol{\omega}}_1 \times \boldsymbol{r}_1$$

$$k_n = \frac{1}{m_1} + \frac{1}{m_2} + [I_1^{-1}(\boldsymbol{r}_1 \times \boldsymbol{n}) \times \boldsymbol{r}_1 + I_2^{-1}(\boldsymbol{r}_2 \times \boldsymbol{n}) \times \boldsymbol{r}_2] \cdot \boldsymbol{n}$$

Where

- $\boldsymbol{n}$ is the normal vector of the collision
- m_i is the mass of entity i
- $\bar{\boldsymbol{v}}_i$ is the linear velocity of entity i,
- $\bar{\boldsymbol{\omega}}_i$ is the angular velocity of entity i,
- $\boldsymbol{r}_i$ is the vector from the center of entity i to the collision point
- $I_i^{-1}$is the inverse inertia tensor of entity i in the world reference frame

Similar equations govern the strength of the tangent impulses that are used to simulate friction as well, the only difference is that in 3 dimensions there are two tangent directions that friction is applied to and each of these directions gets its own impulse strength $\boldsymbol{P_{t_i}}$:

$$\boldsymbol{P_{t_i}} = P_{t_i}\boldsymbol{t_i}$$

$$P_{t_i} = \text{clamp}(\frac{-\Delta\bar{\boldsymbol{v}} \cdot \boldsymbol{t}}{k_{t_i}}, -\mu P_n, \mu P_n)$$

$$k_{t_i} = \frac{1}{m_1} + \frac{1}{m_2} + [I_1^{-1}(\boldsymbol{r}_1 \times \boldsymbol{t_i}) \times \boldsymbol{r}_1 + I_2^{-1}(\boldsymbol{r}_2 \times \boldsymbol{t_i}) \times \boldsymbol{r}_2] \cdot \boldsymbol{t_i}$$

Where:

- $\boldsymbol{t_i}$ is one of the two tangent vectors
- μ is a coefficient of friction calculated from mixing the coefficients of friction from the two colliding entities

These impulse calculations are done multiple times for a fixed number of iterations, with each iteration converging to the true solution to the non-penetration constraint this collision resolution technique is trying to resolve.

An important aspect of this Sequential Impulses technique is the idea of accumulated impulses. This is the way in which we can iteratively solve collisions with multiple contact points. For every contact point, a sum is stored of the accumulated impulse's magnitude we have applied to the entities in this frame, and the clamping procedure seen in the computation of P_{t_i} and P_n are how we ensure that we do not apply a bad impulse that sends other contact points back into collision with the other entity. More details of this can be found in the GDC presentation by Erin Cato (2008)

Render

- **Required Components:**
  - Transform, Render
  - Transform, Camera
  - Transform, PointLight
  - DirectionalLight
- **Summary:** The Render System does all the work required to actually draw the final frames of the game on the screen
- **Details:** This system uses OpenGL 3.3 to interact with the GPU on a system in order to draw the frames. There are multiple different required entity types needed to do such a thing:
  - Entities that have a 3D representation in the game world
  - Entities that act as the views into the world
  - Entities that provide light to the world

Inside the init() function of this system, there is some initial setup of the OpenGL state machine that is performed. This includes creating a Vertex Attribute Array (VAO) which is needed to store the memory layout of vertex attributes that will be passed into the GPU, setting the viewport of the screen that will be rendered to (in this case a 1280x720 resolution window), enable the GL_DEPTH_TEST bit to enable the depth buffer for culling out entities that are occluded by other entities, the clearColor which is the default color of a pixel should no entity end up covering that area of the screen, and lastly is the initialization of the vertex and attribute shader that will be used for computations on the GPU itself.

The addEntity and removeEntity functions have also been overridden to better organize the various entities that can fall into this system. There are different std::set member variables that are used to store these entities; specifically

- **`renderableEntities`** – contains entityIDs corresponding to entities with a 3D representation in the world
- **`cameraEntities`** – contains entityIDs corresponding to entities that have possible views in the world
- **`pointLightEntities`** – contains entityIDs corresponding to entities that act as Point Lights
- **`dirLightEntity`** – contains the entityID that corresponds to the Directional Light that this game world has (only 1 supported so this variable is just an integer)

Additional work needs to be done for entities that have a 3D representation in the game world – specifically, we need to load in their corresponding model and texture onto the GPU by binding the vertex and texture data to buffers on the GPU. We only need to do this once per model and texture, so entities that share the same model and/or texture can also share the same buffer that corresponds to said model/texture. All of this functionality is found within the loadModel() function, which takes advantage of the loadOBJ() function found in RenderSystemUtils.h for loading OBJ files, a file originally found at the opengl-tutorials site (2018). The stbi_load() function found within the stb_image.h library (Nothings, 2019) is used to load the textures, including JPG, PNG, BMP, and TGA.

Currently the only supported model type are OBJ files that have vertex coordinates, texture coordinates, and normal coordinates. All graphics assets (models and textures) need to be placed relative to the graphics/ folder at the root of the project since that is where the code searches for said assets.

The vertex shader, cube.vert, is a simple program that takes the given vertex position in model space and transforms it into normalized device coordinate space. This is done by multiplying the model space coordinates by the Model-View-Projection matrix for an entity. Alongside the normalized device coordinate, we pass additional vertex information into the fragment shader, specifically:

- **`vec3 fragPos`**: represents the position of the vertex in world space
- **`vec2 fragTexCoord`**: specifies where in the texture to sample from for this vertex
- **`vec3 fragNormal`**: a normalized vec3 that represents the normal of the surface at this vertex

All this information is passed into the fragment shader, cube.frag, which is where the color of the given pixel is computed. Here, we use a forward rendering approach to do lighting computations by taking an array of PointLights and a single DirectionalLight and iterate over them to calculate each light's contribution to the pixel's color. We also sample the passed in texture in order to get a baseline color that the entity will take. Specifics on how the lighting computations are calculated can be found at OpenGL tutorial site (2018).

In the update() function of the Render System, a drawn frame is broken up into multiple steps. The first involves selection of which entity's perspective we will be drawing the scene from. This involves iterating over all entities within the cameraEntities std::set and selecting the last active camera entity. The selected entity's TransformComponent and CameraComponent information is then used to calculate the View and Projection portions needed to compute the Model-View-Projection matrix.

The next step is setting up all of the point lights that will be used for lighting calculations in this frame. This involves iterating over the pointLightEntities std::set and setting up the uniform array on the GPU with all the point light entities' characteristics. Currently, the maximum number of points lights in a scene that the engine can support is 100.

The Directional Light is slightly different in the sense that no iteration is needed to setup the uniforms on the GPU that represent the Directional Light since only one is ever supported by the engine – so all that is done is pass in the characteristics from the DirectionalLight entity that correspond to the uniforms on the GPU.

A similar process to that of the point lights occurs with the renderable entities in the scene. We iterate through the rendereableEntities std::set, and for each one, we compute the Model-View-Projection matrix, inverse transpose of the model matrix for the normal, as well as enable and bind the vertex buffers that correspond to the vertex coordinates, texture coordinates, and normal vector of the entity being rendered. Once all this data is passed in, we can the glDrawArrays() function that draws the individual entity on the screen.

Transform

- **Required Components:** Transform
- **Summary:** The Transform System computes the world position of entities that are children of a parent entity.
- **Details:** The current way that parenting works in the engine is that every frame the engine goes from the root of the transform hierarchy, which begins with a dummy "world" entity that is created on application startup, and does a depth-first search to set the worldPosition, worldRotation, and worldScale TransformComponent variables for child entities. This is done since a child entity's regular Transform information (position, rotation, scale) are all relative to the parent's transform information. This allows for the basic pairing of entities based on their movements as well as facilitation of future features such as skeletal animations.

Potential Classroom Usage

One of the goals of this project is to use this engine as a means to teach students about game development. There are many ways that an engine like this can be used in an educational setting, for instance:

- Throughout the semester, students can go about reimplementing the core engine features on System at a time so that they have a better understanding of how the engine functions under the hood
- The professor teaching the class can use the core engine features as a reference during lectures in order to educate students on details of engine development itself
- Students can be asked to extend the engine with their own Systems and Components in order to create their own games based around the Entity-Component-System paradigm

Future Work

There are many opportunities available for this work to be improved upon. Individual systems could be expanded upon; for example:

- Collision
    - Restitution coefficient is accepted but not currently used in the impulse computation (so varying the bounciness of entities doesn't work at the moment). All this needs is using the coefficient in the calculation of the bias term in presolve step of the collision system – see Qu3e for details (Gaul, 2017)
    - Support more shapes beyond the simple Box and Sphere, such as Capsules Convex Hulls, and Terrains
    - Support compound colliders (colliders that are part of children of an entity are treated as a single collider, requires recomputing the center of mass)
    - Add raycast querying abilities
    - Improve performance by implementing a Spatial Bounding Volume solution to decrease the number of entity pairs checked per frame

    - Implement a collision detection matrix that allows masking of collision detection for different types of entities based on their tag
- Rendering
    - Support for different model files other than OBJs
    - Skeletal Animation support
    - Implement a deferred rendering system that allows an arbitrary large number of lights to be placed in the scene without performance degradation
    - Proper Skybox/Skydome support
    - Improve rendering of textures by supporting MIP maps
    - Support additional light types such as spotlights
    - Text rendering
    - GUI rendering and point/click button support
- Overall Engine
    - Improve support for parenting entities together that applies the rotation and scale properly (just applying the parent transformation matrix and then extrapolating these values using glm would work)
    - Support for naming/tagging different entities and allowing users to get entities by name/tag
    - A logging feature that allows for writing to a log file – this could replace the need to use of std::cout for debugging and as such also remove its performance cost
    - A message passing feature that allows Systems to send messages to each other similar to a publisher-subscriber model

References

Cato, E. (2008). Modelling and Solving Constraints. Retrieved March, 2019, from https://box2d.org/downloads/

De Vries, J. (2019, January). Multiple lights. Retrieved November, 2018, from https://learnopengl.com/Lighting/Multiple-lights

Gaul, R. (2017, October 27). RandyGaul/qu3e. Retrieved January, 2019, from https://github.com/RandyGaul/qu3e

Gregorious, D. (2015, May). Robust Contact Creation for Physics Simulation. Retrieved January 30, 2019, from http://media.steampowered.com/apps/valve/2015/DirkGregorius_Contacts.pdf

Nothings. (2019, March 05). Nothings/stb. Retrieved November, 2018, from https://github.com/nothings/stb

Opengl-Tutorials. (2018, February 23). Opengl-tutorials/ogl. Retrieved November, 2018, from https://github.com/opengl-tutorials/ogl/tree/master/common

G. (2016). Sphere-AABB. Retrieved January 15, 2019, from https://gdbooks.gitbooks.io/3dcollisions/content/Chapter2/static_sphere_aabb.html

The_O_King, & Yakk - Adam Nevraumont. (2018, September 19). Call a vector's function given the vector's pointer as a void *. Retrieved from https://stackoverflow.com/questions/52397451/call-a-vectors-function-given-the-vectors-pointer-as-a-void

Appendix A: Core Components and Associated Config File Format

- Transform:
  - int – Parent ID (0 = Parent to the world coordinate system)
  - float – Position.x (local)
  - float – Position.y (local)
  - float – Position.z (local)
  - float – Rotation.x (local, Euler Angles, will be converted to Quaternion)
  - float – Rotation.y (local, Euler Angles, will be converted to Quaternion)
  - float – Rotation.z (local, Euler Angles, will be converted to Quaternion)
  - float – Scale.x (local)
  - float – Scale.y (local)
  - float – Scale.z (local)
- Render:
  - string – Model file name (MODEL_NAME.obj)
  - string – Texture file name (TEXTURE_NAME.jpg)
  - float – Diffuse.r (range 0-1)
  - float – Diffuse.g (range 0-1)
  - float – Diffuse.b (range 0-1)
  - float – Specular.r (range 0-1)
  - float – Specular.g (range 0-1)
  - float – Specular.b (range 0-1)
  - float – Shininess (range 0-MAX_FLT)
- Physics
  - float – mass (0 mass means the entity is immovable, such as a wall)
  - float – Gravity Scale (range 0-MAX_FLT, recommended between 0-1)
  - float – Restitution Coefficient (range 0-1, how bouncy the entity is)
  - float – Friction Coefficient (range 0-1, higher friction means less sliding)
  - bool – LockRotation.x (0 = free to rotate, 1 = rotation on axis locked)
  - bool – LockRotation.y (0 = free to rotate, 1 = rotation on axis locked)
  - bool – LockRotation.z (0 = free to rotate, 1 = rotation on axis locked)
  - bool – LockPosition.x (0 = free to translate, 1 = translation on axis locked)
  - bool – LockPosition.y (0 = free to translate, 1 = translation on axis locked)
  - bool – LockPosition.z (0 = free to translate, 1 = translation on axis locked)
- BoxCollider:
  - bool – isTrigger (0 = False, 1 = True, collision only detected, not resolved)
  - float – offset.x (offset the center of the collider from the entity's position)
  - float – offset.y (offset the center of the collider from the entity's position)
  - float – offset.z (offset the center of the collider from the entity's position)
  - float – halfsize.x (half the extent of the box collider on this axis)
  - float – halfsize.y (half the extent of the box collider on this axis)
  - float – halfsize.z (half the extent of the box collider on this axis)

- SphereCollider:
    - bool – isTrigger (0 = False, 1 = True, collision only detected, not resolved)
    - float – offset.x (offset the center of the collider from the entity's position)
    - float – offset.y (offset the center of the collider from the entity's position)
    - float – offset.z (offset the center of the collider from the entity's position)
    - float – radius (radius of the sphere)
- Camera
    - bool – isActive (0 = False, 1 = True, whether this will be chosen to render from)
    - float – Field of View (the Y-direction FOV of the camera)
- PointLight
    - float – intensity (How bright the source light is)
    - float – color.r (range 0-1)
    - float – color.g (range 0-1)
    - float – color.b (range 0-1)
- DirectionalLight
    - float – color.r (range 0-1)
    - float – color.r (range 0-1)
    - float – color.r (range 0-1)
    - float – direction.x (Euler Angles, degrees – converted to radians)
    - float – direction.y (Euler Angles, degrees – converted to radians)
    - float – direction.z (Euler Angles, degrees – converted to radians)