Game Level Design of Custom ECS Engine

A Capstone Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Andrew Niedringhaus

May 7, 2020

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: _____

Approved: _____ Date _____
Mark Floryan, Computer Science

Introduction

A game engine is a software development tool that assists developers who build video games. Game engines are used for developing video games on consoles (i.e. Xbox, PlayStation), personal computers, and mobile devices. The state of the art regarding game engines are Unreal Engine and Unity, which are open source. Most companies use their own proprietary game engine. Modern engines are currently reaching the limit on VR rendering and adding mobile gaming functionality.

For this capstone, I built upon an engine that was made from scratch in a previous capstone project. I created a game level using the engine, added new components and systems for the engine, and made some other minor fixes to the engine. I will start by going over how the engine works, the individual pieces that make up the architecture, and how to run the game level on your own device. Then I will demonstrate what I accomplished with the engine, detailing the level, explaining what components I added and what important steps I took during the capstone. Lastly, I will list out what other improvements can be made from the engine.

About L3Beh

The game engine made from last year is named L3Beh. It uses an entity-component-system (ECS) architecture. I will explain in the next section how the architecture works and how they are incorporated into L3Beh. The engine is written primarily in C++ and utilizes the OpenGL library. OpenGL is an API that helps L3Beh with rendering 2D and 3D vector graphics. We also heavily utilize two OpenGL extensions: GFLW and GLEW. GLFW is used to create and manage windows and OpenGL contexts. It also handles keyboard and mouse input. GLEW is used to improve the efficiency of loading OpenGL extensions.

Using a config txt file, the engine is capable of building and running a complete level that is fully customizable. The engine is capable of creating and rendering objects and constantly updating them. New components can be added to further improve the engines capabilities. The engine currently only support OBJ files for rendering, and does not yet support skeletal animation for entities.

A goal of the development of L3Beh is to use the engine as a way to teach students about game development under the hood. Students can recreate fragments of this engine, refer to features of the engine, or even extend the capabilities of the engine.

Architecture

Entity: the object in the game, consisting solely of a unique identifier. In L3Beh, entities are represented using a single integer to identify it. The integer values are used to connect components associated with a given entity. An example of this would be an entity representing a person can have a movement component, allowing the person to move to another location).

```
std::set<int> entities;
```

Component: an ability or description that is associated with an entity. Components are used in game engines to strictly define how an entity is allowed to interact with the world. In L3Beh, components are represented as structs that contain all the data that pertains to a certain description. An example from L3Beh would be the transform component. One function of this component is that it can be attached to an entity and describe the x, y and z position of it.

```
struct AIMovementComponent{
    bool active;
    bool grounded;
    bool movingRight;
    float startingX;
    float startingY;
    float startingZ;
    float health = 100;
    float walkSpeed = 2.0;
    float runSpeed = 4.0;
};
```

In this example we have a component to describe an AI that will move around the world, and can interact with the player. We can assign values like walkSpeed to control its speed while moving around the map, and startingX/Y/Z so the AI does not get lost while moving around the map.

System: running separately in their own thread, they define global actions that affect all entities possessing a certain component. In L3Beh, a list of required components is used to distinguish what entities a system can and cannot affect. L3Beh allows there to be multiple different lists that a system can allow.

```
class AIMovementSystem: public System {
    private:

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

In this example we want to create a system to make logic surrounding the AI movement component. To do this we create a subclass of the System class, include the struct from our custom components file, include GLM for the linear algebra in the update method, include our input file to get key presses, and our components and world file to get other components for an entity and to interact with the world the entity is in.

Update method example

The “update” method in AIMovementSystem is called every frame while the game is running. This is where we can insert logic to update a components contents, and will result in seeing actual changes in the world. Here is a simple example of what the AI movement logic could look like, and what the update method does.

```
void AIMovementSystem::update(float deltaTime){
    for (int e : entities){
        AIMovementComponent& ai = mWorld->getComponent<AIMovementComponent>(e);
        if (ai.active){
            TransformComponent& tc = mWorld->getComponent<TransformComponent>(e);
            PhysicsComponent& pc = mWorld->getComponent<PhysicsComponent>(e);

            if (tc.position.x >= ai.startingX + 5)
                    pc.velocity.x = -1.0f * ai.runSpeed;
            else if (tc.position.x <= ai.startingX - 5)
                    pc.velocity.x = -1.0f * ai.runSpeed;
        }
    }
}
```

In this simple example we are given an AI that is moving around the world and we want it to pace back and forth. We are given its starting X position and want it to move 5 units to the right and left of that spot. We can implement this logic by iterating through the list of entities until we get to one with an active AI Movement Component. We then access that entities Transform and Physics Component and check its location relative to the given starting X location. If our AI is moving in one direction and gets more than 5 units past the original X location, then we switch its direction by negating its velocity.

Running the Engine on Your Device

Here are the steps I used to set the engine up on Mac:

1. Install Homebrew (go to https://brew.sh/)
2. Go to terminal and enter the following:
    a. “brew install glfw3”
    b. “brew install glew”
3. If using the version v1.0 (last year’s one), then go to the “runner.h”, find the line:

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

and add the following line underneath it:

```
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

4. To compile everything run:

```
g++ -o ./compile/outRelease *.cpp core/*.cpp core/systems/*.cpp -std=c++17 -
lglew -framework OpenGL -lglfw
```

5. If the compilation was successful, then you can run your game!
    a. “./compile/outRelease” should start the game from the terminal.

Using last year's build notes, this is how to set up the engine on Windows:

- 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 o 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-Howdo-I-build-GLEW-for-mingw-w64-using-cmake-onWindows?p=1283379&viewfull=1#post1283379
    - But I have included the built binaries for MinGW64 in the GLTools/glew2.1.0.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”

Game Level

Creating a game level using L3Beh requires importing OBJ files to render and configuring the game in a text file. We use the configuration file to create entities and add components to each entity. An example of this is the config file below:

```
Entity Player
PlayerMovement 1 0
Physics 1 0 1 .2   0 0 0  0 0 0
SphereCollider 0   0 0 0    1
Transform 0    0 0 -2   0 0 0  1 1 1
Camera 1 150
/Entity
```

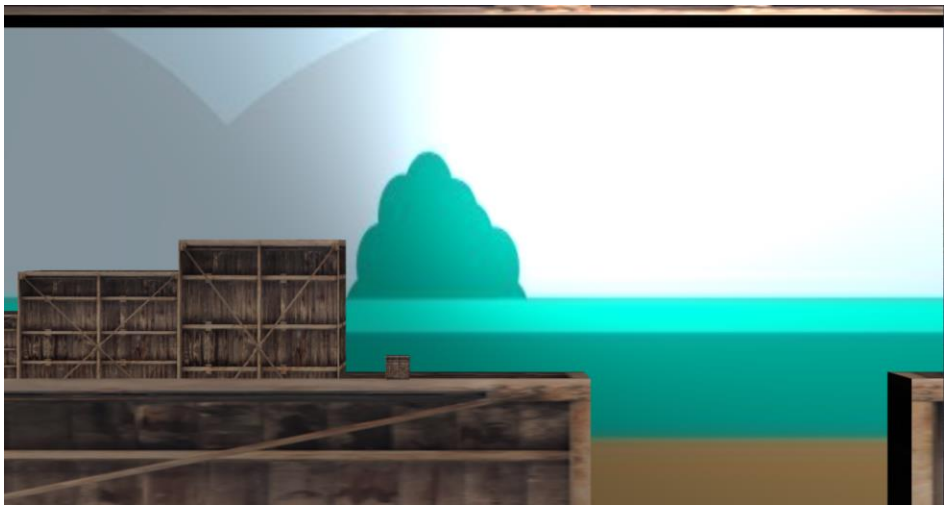
In this example we make an entity named “Player.” We then give the entity the following components:

- Player Movement: to control how Player can move around the world
- Physics: to give Player mass and friction coefficients
- Sphere Collider: to allow us to add logic when Player collides with another entity
- Transform: to locate its position and rotation in the world
- Camera: to determine the field of view of the camera to the entity

When we add new components to our engine, we can take care of their input in the customWorldGen method of the MyWorld class. For example, the way we handle input for Player Movement looks like:

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

From here we can add as many new components as we like. Some examples of components I added were an AI Movement component and projectile component. In the next page are a couple of pictures from a game level that I created.



Next Steps

There are many ways this engine can improve. Two things I wanted to add, but did not finish were split screen capability and adding a library of components to the engine. Here is a list of next steps for the engine:

In Progress:

- Split Screen Capability
- Making a library of premade components for future developers

Rendering:

- Support more model files (currently the engine only accepts OBJs)
- Allow skeleton support for the body of entities
- Text/HUD rendering capability

Collisions:

- Support for more shapes, currently the engine only supports box and sphere shapes
- Fix friction/restitution coefficient. Currently causing weird bug where if an object moves into an unmovable object, then the moving object gets flung back and never stops.

Miscellaneous:

- Logging feature, so you can check a logging file to get more details about an error
- System-to-system communication. When the update method is running for a given system it cannot communicate with other systems.
- Add feature where you can retrieve entities by name, or add a way to assign “tags” to entities so you can filter/sort entities.

References

El Sheikh, O. (n.d.). The-O-King/L3beh. https://github.com/The-O-King/L3beh

OpenGL. (n.d.). http://www.opengl-tutorial.org/beginners-tutorials/tutorial-1-opening-a-window/

Opengl-Tutorials. (n.d.). opengl-tutorials/ogl. https://github.com/opengl-tutorials/ogl/tree/master/common

Khronos. (n.d.). Getting Started. https://www.khronos.org/opengl/wiki/Getting_Started