Beary: Building a 3D Game Engine with OpenGL

Jill Platts, Advisor: Michael J. Reale

Graduate Independent Study, Department of Computer Science, SUNY Polytechnic Institute

Abstract- Using the OpenGL specification for rendering computer graphics, a 3D game engine, named Beary, was developed. With an iterative development approach, functionality was added through planning, implementation, testing, and evaluation. Using an adequate foundation of well-known libraries for basic game engine functionality, additional features implemented include a graphical user interface (GUI), a more flexible object loader, methods for saving and loading a scene, sound functionality, and the start of a physics system, with both bounding boxes and mouse picking.

Index Terms- computer graphics, game engine, OpenGL

I. INTRODUCTION

One of the most utilized graphic application programming interfaces (APIs), OpenGL [1] is a popular choice for rendering video games, game engines, and various other 2D and 3D graphic applications. With a myriad of function calls available, the specification offers considerable flexibility, actively evolving as new ideas surface. However, OpenGL [1] does suffer from some fragmentation as different vendors may ship implementations corresponding to different versions of the specification [14]. Nonetheless, OpenGL [1] is known for demonstrated ease of interaction directly with a computer's graphics processing unit (GPU), for hardware-accelerated rendering.

Working to explore and understand the advantages, disadvantages, and challenges of building a 3D game engine, OpenGL [1] was used as the rendering specification for the development of a 3D game engine, named Beary [2]. Though, there are several popular 3rd-party game engines available for game development, there are numerous benefits to building a custom game engine. Creative control and customizability are two of the main benefits. Extensibility is another notable benefit.

II. METHOD

Using an iterative approach, work on Beary [2] emphasized incremental builds. Once a feature or functionality was selected, planning and design was followed by code implementation. Code was then analyzed and observed, with necessary corrections made to achieve desired performance. Any established code impacted by a new feature or functionality was also altered or optimized, as necessary. Often utilizing available, established libraries, several working parts were added to the engine over the course of a few months.

A. Foundation

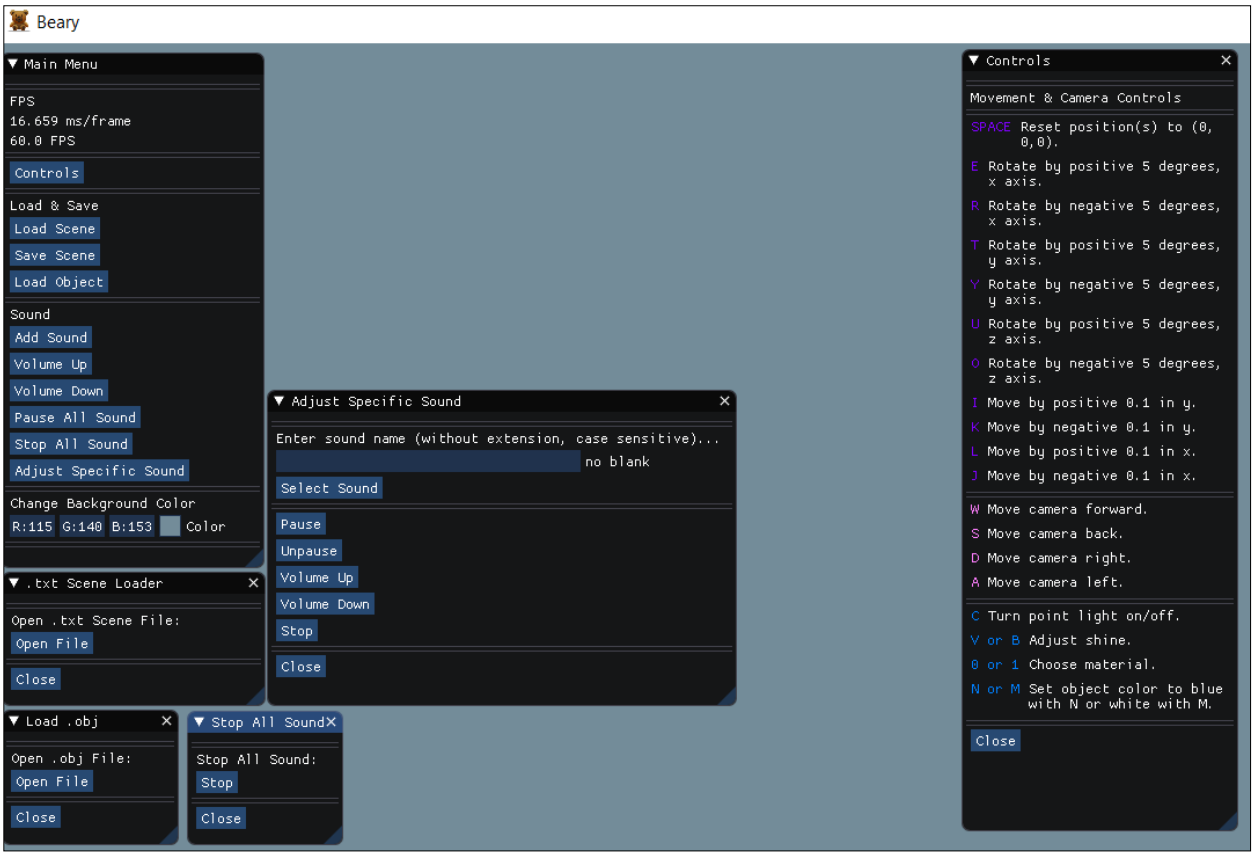
Using the OpenGL 4.6 [1] specification as a base, several libraries were immediately implemented, including GLFW 3.3.2 [3] for creating and managing windows, GLEW 2.2.0 RC3 [4] for querying and loading OpenGL extensions, GLM 0.9.9.7 [5] for lightweight mathematics, Assimp 5.0.1 [6] for parsing and loading sophisticated 3D models, and stb_image 2.23 [7], as well as stb_image_write 1.13 [7], for reading and writing images. Adding considerable functionality, as well as making further

game engine development a more pleasant experience, these libraries were added using the CMake [8] build manager, in combination with the engine's main IDE, Microsoft Visual Studio [9].

B. Graphical User Interface (GUI)

With several building blocks already established, including the ability to load complex 3D models, as well as the ability to create and manage a window with both mouse and keyboard input, the necessary first step was to implement a working graphical user interface (GUI). With the goal of abstracting away all current and future features and functionality to a more attractive and usable form, Dear Imgui [10] was implemented.

Designed to enable fast iterations, Dear Imgui [10] required a full reorganization of Beary's [2] main.cpp rendering loop to reflect cleaner, productivity-enhancing interactivity. Favoring a simple, yet highly customizable GUI style, various interactive elements were created with Dear Imgui [10]. Some elements created include a main menu with buttons linked to additional game engine functionality, an informational window listing the engine's movement, color, texture, and camera controls, an FPS tracker, a background color adjuster, and much more.



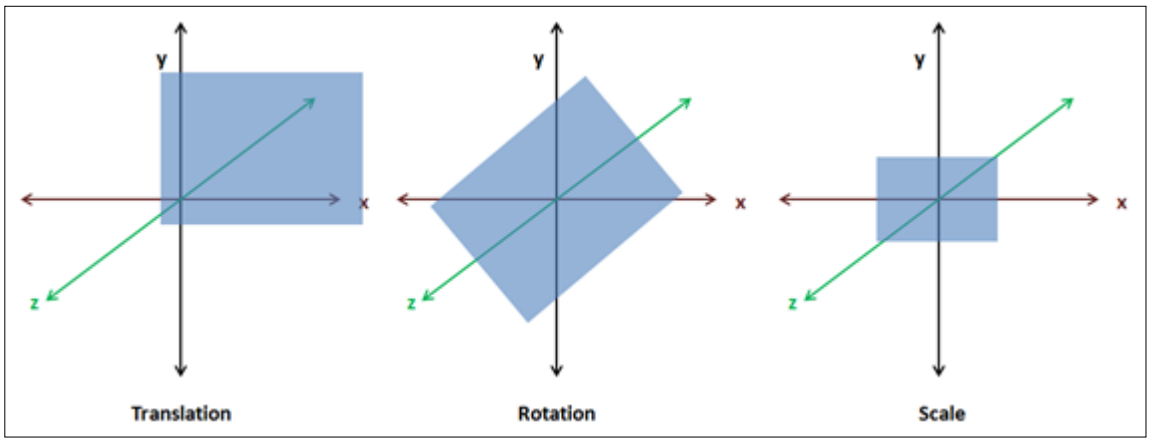
Example of various features accessed through implemented GUI. [2]

C. Loading Objects

Created as an extension of Dear Imgui [10], imgui-filebrowser [11] was added to Beary [2] for improved user interactivity when loading object (.obj) files. A GUI for browsing all files on one's computer, imgui-filebrowser [11] provides an abstraction for locating object files on a computer, as well as any other type of file needed. As an object file is loaded into the engine's current scene, vertices,

texture coordinates, and polygons that compose the 3D model are processed primarily by ModelLoader.cpp [2], assisted by Assimp 5.0.1 [6].

Focusing on extensibility of the load object files feature, models loaded into the engine are privy to multiple transformations. Dependent on relevant matrices, including model, view, and projection, keys E, R, T, Y, U, O, and SPACE, rotate a model around the X, Y, and Z axis', as well as reset model position, respectively. L, J, I, and K, translate a model across the X and Y axis', respectively. Additional key or GUI functionality can easily be added for scaling.



"Basic operations to act upon any model – translation, rotation, scale" [12].

D. Load and Save Scene

Further utilizing imgui-filebrowser [11] for interactivity, the ability to both load and save a collection of models, as well as their positions, was added to Beary [2]. Loading a scene text (.txt) file layers seamlessly with the basic object (.obj) loading functionality. Current positions (x, y, z) of each model in a scene are also saved and loaded to and from the scene text file. All objects, as well as their positions, are further stored, using C++'s vector class, in a growable array of objects, as they are loaded in, for ease of scene and model editing. Focusing on extensibility of the scene feature, a scene text file can be further enhanced to store various additional information, such as color, texture, lighting, etc., both per object and per overall scene.

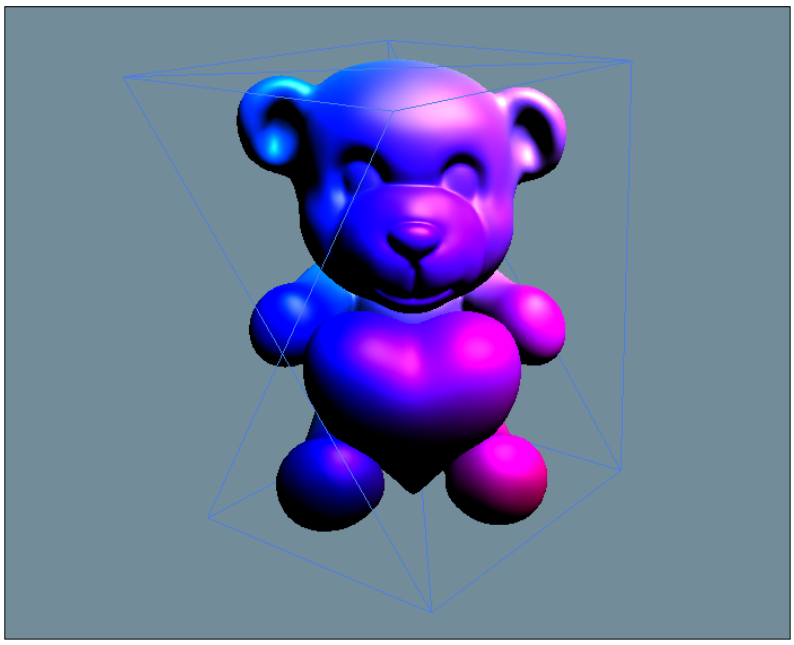
E. Sound

Adding life to a game or 3D experience, audio capabilities allow a game engine to provide functionality for adding sound effects, adding background music, layering sound, attaching sounds to model movement, and much more. Equipped with irrKlang [13], Beary [2] again uses imgui-filebrowser [11] for user interactivity when adding both WAV and MP3 formatted sounds. Sound functionality includes the ability to load multiple sounds, turn up or turn down the volume of all sounds, as well as individual sounds, and stop or pause all sounds or a specific sound. Focusing on extensibility of the audio feature, further editing functionality can be added, such as changing the speed of a sound, attaching model movement to a sound, and much more.

F. Physics: Bounding Boxes

A working physics system is an extensive and important implementation when developing a game engine. Choosing to use an axis-aligned bounding boxes (AABB) approach, each model loaded into Beary [2] is wrapped in a cube. Located in Beary's [2] MeshShaderGL.cpp, a cube is created, then

applied to a model. The size and position of the cube, for each model, is calculated within ModelData.cpp, via the min/max of the vertexes on all three axes (x, y, z). Moving beyond a static axis-aligned bounding box implementation, cube size and position adjusts as object moves along an axis or is rotated along an axis.



Example of an axis-aligned bounding box (AABB) loaded with a model. [2]

G. Physics: Mouse Picking

Developing another necessary aspect of a collision and, ultimately, a properly functioning physics system, mouse picking is implemented in Beary's [2] MousePicker.cpp class. Working from screen coordinates through clip space and eye space, a world space ray is delivered. Utilizing the previously implemented bounding boxes, the ray gathered from this aspect of the still-developing physics system can be used for collision detection, as well mouse-based object selection and translation.

```
glm::vec3 MousePicker::calculateMouseRay() {
    //screen coordinates
    double x1 = x;
    double y1 = y;
    glm::vec2 normalCoords = getNormalizedDevCoor(x1, y1);
    //clip space
    clipCoords = glm::vec4(normalCoords.x, normalCoords.y, -1.0f, 1.0f);
    //eye space
    glm::vec4 eyeSpaceCoords = toEye(clipCoords);
    //world space
    glm::vec3 worldSpaceRay = toWorldCoor(eyeSpaceCoords);
    worldSpaceRay.z = -worldSpaceRay.z;
    return worldSpaceRay;
}
```

Function, from MousePicker.cpp, that calculates mouse ray in world space. [2]

III. RESULTS

A rendering approach, as well as a camera, shaders, and additional basic game engine functionality were implemented successfully. A lightweight GUI, ability to load complex models, loading and saving a scene, sound capabilities, a mouse ray position, bounding boxes, and the start of collision detection all perform as expected, with each feature's functionality easily extensible. Every building block advanced the engine further, offering a more customized environment, with few limitations on additional advancements.

While implementing features, continuously optimizing neighboring code, and exploring avenues for adding additional functionality, it became evident that the greatest caveat of building a custom game engine is time. Working on the engine for a few short months resulted in an excellent starting point, with the beginnings of multiple pertinent features. Due to the complexity of GPU-based development, each feature and functionality requires a considerable amount of time for planning, design, implementation, and optimization. However, once a feature is implemented, immense extensibility allows for a significant amount of customization and flexibility.

IV. CONCLUSION

Due to the popularity of 3rd-party game engines, online resources for developing a custom 3D game engine were sparse. Learning how to and, ultimately, successfully implementing additional game engine functionality was time-consuming. Code debugging and optimization were required on a continuous basis to ensure harmonious implementation of even the smallest functionality. Nonetheless, every feature added further customized the engine, as well as laid additional groundwork for more advanced features.

Through the challenges of implementing various features in Beary [2], including a GUI, ability to load complex models, loading and saving a scene, sound capabilities, a mouse ray position, and bounding boxes, the main benefits of developing a custom 3D game engine were discernable. Creating a game engine that is both easy-to-use and filled with targeted functionality is challenging, but purposeful work. Unconstrained by the limitations of a commercial game engine, Beary [2] is both a platform for skill development and, ultimately, a customizable set of tools for developing a truly unique game or 3D experience.

V. FUTURE WORK

Considering the work already done on Beary [2], utilizing the successful bounding box and mouse ray implementations to achieve mouse-based translation and collision detection is the logical next step for the engine. A work in progress, the engine's completed physics system will clear the way for additional features, such as Physically-Based Animations (PBAs). A user's ability to perform model scaling, terrain generation, and bump maps are all additional functionalities that would also benefit the game engine.

An extensive area of GPU-based development, a game engine can benefit from the implementation of a wide array of features and functionality. With no real chronological order of implementation, a game engine created off an established rendering specification (in this case, OpenGL [1]) offers the advantage of nearly full creative freedom with regards to further feature implementations. A custom game engine's functionality, customizability, and extensibility is only limited by one's

available time, GPU capabilities, and, most importantly, the ability to harness one's ideas into working implementations.

REFERENCES

[1] OpenGL - The Industry's Foundation for High Performance Graphics. Retrieved May 9, 2020, from https://www.khronos.org/opengl/

[2] 3dgameengine_jillplatts. Retrieved May 9, 2020, from https://github.com/jmppmj/3dgameengine_jillplatts

[3] Graphics Library Framework. Retrieved May 9, 2020, from https://www.glfw.org/

[4] The OpenGL Extension Wrangler Library. Retrieved May 9, 2020, from http://glew.sourceforge.net/

[5] OpenGL Mathematics. Retrieved May 9, 2020, from https://glm.g-truc.net/0.9.9/index.html

[6] Open Asset Import Library. Retrieved May 9, 2020, from https://www.assimp.org/

[7] stb single-file public domain libraries for C/C++. Retrieved May 9, 2020, from https://github.com/nothings/stb

[8] CMake. Retrieved May 9, 2020, from https://cmake.org/

[9] Visual Studio IDE, Code Editor, Azure DevOps, & App Center. Retrieved May 9, 2020, from https://visualstudio.microsoft.com/

[10] Ocornut. ocornut/imgui. Retrieved May 9, 2020, from https://github.com/ocornut/imgui

[11] AirGuanZ. AirGuanZ/imgui-filebrowser. Retrieved May 9, 2020, from https://github.com/AirGuanZ/imgui-filebrowser

[12] lwjglgamedev. lwjglgamedev/lwjglbook-bookcontents. Retrieved May 9, 2020, from https://github.com/lwjglgamedev/lwjglbook-bookcontents

[13] irrKlang high level audio engine. Retrieved May 9, 2020, from https://www.ambiera.com/irrklang/

[14] McReynolds, Tom, and David Blythe. *Advanced Graphics Programming Using OpenGL.* Elsevier Morgan Kaufmann Publishers, 2005.