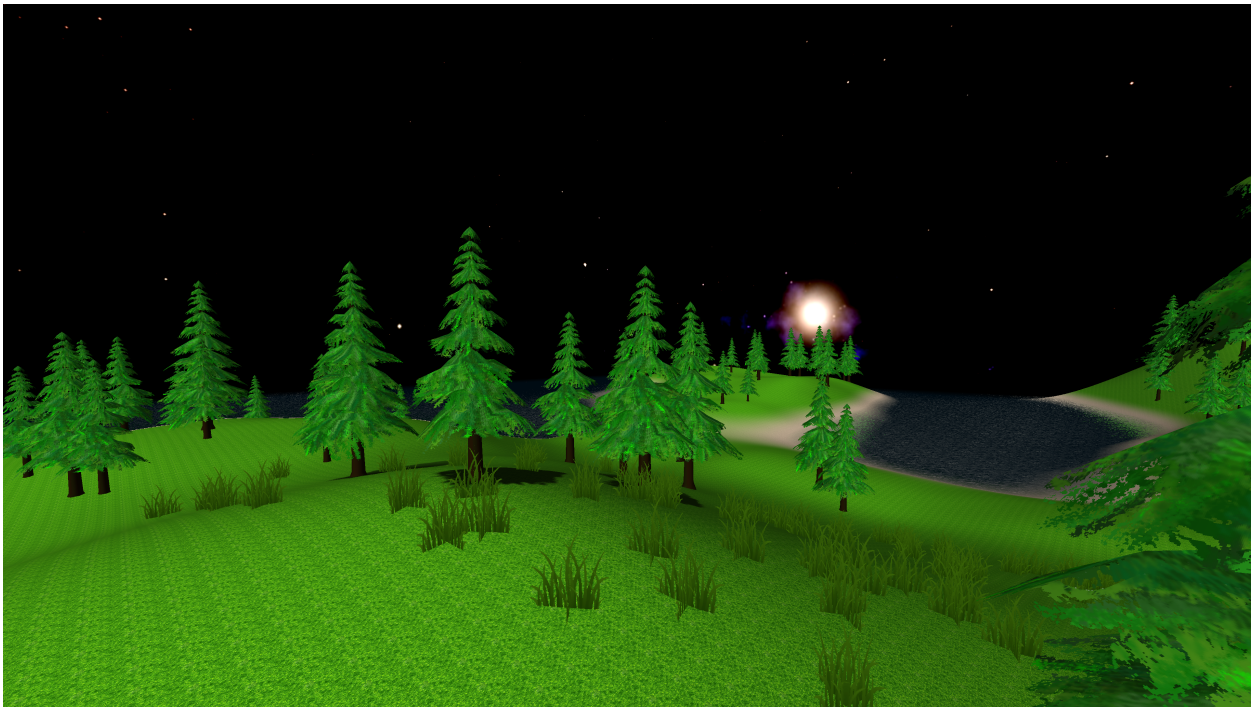
Luminos Release Notes
Version 0.0.1-Pre Alpha

Nicholaus Clark

April 22, 2017



Contents

1 Introduction to Luminos Engine

Luminos Engine is a lightweight Java gaming engine. It currently supports only Windows, but will be supporting other platforms that use the OpenGL API.

1.1 Getting Started

In order to get started, you first need to include the current Luminos jar into your project. All required binaries are pre-packaged into the jar, so you do not need to worry about setting native library locations.

Inside your project, you will need three classes to begin with: a main class, a class to hold your scene, and a class to hold your game object implementation. Each of these files will be gone over in detail later. Your main class is what runs the project, the scene is what holds the data, and the game object implementation is how the GPU will interpret the models.

1.1.1 Main Class

Your main class needs to extends the `Application` class. In your main method, load the settings file. With the jar should be the default provided configuration, `settings.lum`. You will then run the main file using the platform appropriate method to start the thread (`.start()` for Windows). You will also need a `start()` method inside your main class. This file will open a new window, load a scene, prepare the engine, render the scene, and finally dispose of the application. Here is an example Main class file.

```
// imports
// file name
public static void main(String[] args) {
    Thread Main = new Main();
    try {
        Application.loadSettings("luminos.lum");
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    benchmark.start();
}

public void start() {
    Window win = null;
    try {
        window = new Window("Hello Luminos!" 1280, 720, true, false,
            false, true);
        Loader loader = new Loader();
        Scene scene = TestScene.init(loader);
        Engine.createEngine(scene.getRenderer(), loader);
        Engine.start(window);

        this.swapScene(scene);
        window.primeFPSCounter();
        this.render(window);

        loader.cleanUp();
        window.close();
        Engine.close();
        this.close();
    }
```

```
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

1.1.2 Scene Class

We will be calling our scene `TestScene`. All scenes in the engine need to implement the `Scene` interface. A scene needs to store game objects, terrains, water tiles, point lights, directional lights, relevant master renderer, camera, and the focal game object. In order to initialize our scene, we will make a static `init(Loader)` method. This will load our terrains, game objects, water, lights, and sun to the scene.

```
        public static TestScene init(Loader loader) {
            // Load models and textures to TexturedModel
            List<GameObject> objects = new Vector<GameObject>();
            for (int i = 0; i < 100; i++) {
                objects.add(new Entity(TexturedModel, Position, Rotation,
                    Scale));
            }
            List<Terrain> terrains = new Vector<Terrain>();
            for (int x = 0; x < 5; x++) {
                for (int z = 0; z < 5; z++) {
                    terrains.add(new Terrain(x, y, Integer, loader,
                        TerrainTexturePack))
                }
            }
            List<PointLight> lights = new Vector<PointLight>();
            DirectionalLight sun = new DirectionalLight(new
                Vector3(1, 1, 1), new Vector3(-0.3f, 1, -0.6f), 1.0f);
            List<WaterTile> tiles = new Vector<WaterTile>();
            for (int x = 0; x < 10; x++) {
                for (int y = 0; y < 10; y++) {
                    tiles.add(new WaterTile(x, y, 0, new Vector2(50, 50)));
                }
            }

            TestScene scene = new TestScene(objects, terrains, tiles,
                lights, sun);
            Camera cam = new Camera(objects.get(0));
            scene.masterRenderer = new MasterRenderer(loader, cam);
            scene.camera = cam;
            scene.entity = objects.get(0);

            return scene;
        }
```

All of the other methods required for the implementation of `Scene` are outlined by the Javadocs.

1.1.3 Game Object Implementation

The implementation of the `GameObject` abstract class will be called `Entity`. It will be as follows:

```
        // set up class, imports
        public Entity(TexturedModel model, Vector3 position,
            Vector3 rotation, Vector3 scale) {
            super(model, positio, rotation, scale);
        }
```

Once complete, the code will display a window with a scene. You are now on your way to mastering the Luminos Engine.

2 Added Features

This section focuses on the features added to the Luminos Engine in the past iteration. It will also give a brief summary of how to implement them into your application or game.

2.1 Engine Components

The Luminos Engine is focused around the idea that a game engine should have a substantial number of features built in, but a way to add more. Currently implemented in Luminos is a networking framework, physics engine framework, and rendering engine.

2.1.1 Networking

Luminos' networking framework is written on top of User Datagram Protocol (or UDP). The current iteration of networking only supports basic PING/PONG network packets, but serialization is in place for the transfer of more complex objects across networks.

While the current networking framework may be basic, it still has the ability to be attached to an application. In order to add it to the engine, there must be an `Application` instantiated. The `Client` is then attached to the `Application`.

```
        public class App extends Application {

            Client client = null;

            public void init() throws Exception {
                client = new Client(this, "localhost");
                this.attachThread(client);
            }

            public void destroy() throws Exception {
                this.close();
            }

        }
```

First, the client must be instantiated. The client's constructor can throw an exception if the socket cannot be bound to. Once the socket has be instantiated, the client can then be added to the application. By adding the client to the `Application`, it is then controlled synchronously with the application, ensuring destruction when the application is closed.

Adding a server thread is very similar. This can either be done in the same application or in a stand-alone application. Like the `Client` addition, it can be attached to an `Application`. To implement the `Server`, I will demonstrate it be used in conjunction with the prexisting client code above.

```
    public class App extends Application {

        Client client = null;
        Server server = null;

        public void init() throws Exception {
            client = new Client(this, "localhost");
            this.attachThread(client);

            server = new Server(this);
            this.attachThread(server);
        }
```

```
        public void destroy() throws Exception {
            this.close();
        }

    }
```

Like the client, a server must be instantiated before attachment. By attaching the newly created server to the application, it provides the same benefits as with the client.

2.1.2 Physics

Currently, there is only a framework for physics in the Luminos Engine. The engine component will support rigid body physics once matured. The engine will allow for both collision detection and response by the objects effected.

In order to implement physics in the engine, a `PhysicsEngine` object must be created. Once created, it can be attached to the global `Engine`.

```
    public class App extends Application {

        Client client = null;
        Server server = null;
        PhysicsEngine pEngine = null;
        Scene scene = null;

        public void init() throws Exception {
            // Create new scene

            client = new Client(this, "localhost");
            this.attachThread(client);

            server = new Server(this);
            this.attachThread(server);

            pEngine = new PhysicsEngine(scene);
            Engine.addPhysicsEngine(pEngine);
        }

        public void destroy() throws Exception {
            this.close();
        }

    }
```

2.1.3 Rendering

Perhaps the easiest component to add to your application is the rendering engine. The rendering engine is able to be created by the `Engine` class, as was the physics engine.

To add a rendering component to the engine, the `Engine` class' `createEngine()` function needs a `MasterRenderer` and a `Loader`. The function throws an exception if the scene manager cannot be created behind the scenes.

```
    public class App extends Application {

        Client client = null;
        Server server = null;
```

```
        PhysicsEngine pEngine = null;
        Scene scene = null;
        MasterRenderer masterRenderer = null;
        Loader loader = null;

        public void init() throws Exception {
            // Create new scene
            // Create masterRenderer
            // Create loader

            client = new Client(this, "localhost");
            this.attachThread(client);

            server = new Server(this);
            this.attachThread(server);

            pEngine = new PhysicsEngine(scene);
            Engine.addPhysicsEngine(pEngine);
            Engine.createEngine(masterRenderer, loader);
        }

        public void destroy() throws Exception {
            this.close();
        }

    }
```

2.1.4 Implementing Your Own Component

The goal of the engine is to always have everything you need at your fingertips. If there comes a scenario where it does not include what you need, you can implement your own component.

All engine subcomponents are descendants of the `EngineComponent` abstract class. Each component must have at least two methods, `update(Scene)` and `dispose()`. This ensures that each time the window is updated, the custom component's state can be updated as well. All resources of the component that need to be released upon application close must be relinquished in the `dispose()` method.

HelloWorldComponent.java

```
        public class HelloWorldComponent extends EngineComponent {

            @Override
            public void update(Scene scene) {
                System.out.println("Hello World!");
            }

            @Override
            public void dispose() {
                System.out.println("Closing!");
            }

        }
```

2.2 File Utilities

The Luminos Engine contains many file system utilities. Currently supported are both plaintext and binary files. For models, both Collada (.DAE) and Waveform Object (.OBJ) files currently are supported. Luminos also has the ability to load resources as streams internally from the JAR. Currently, shaders are packed into the JAR and are loaded using the internal loader.

There are many plaintext reading and writing options included in Luminos. The engine provides a way to load and write files that are formatted so the engine can quickly read them, as well as are readable to the naked eye. These are via the `FileReader` and `FileWriter` classes. They each read/write `PlainTextObjects`. Another option for data exporting is the `CSVWriter`. Currently in progress is a `CSVReader`.

Luminos also supports serialization. There are 5 data structures currently supported for serialization and deserialization:

1. Fields
2. Arrays
3. Strings
4. Objects
5. Databases

The current iteration of the engine allows for reading and writing of serialized databases to and from files. Future iterations of the engine will also support streaming to the networking framework.

Finally, Luminos supports the reading of XML files. It uses Java's built-in pattern recognition system. The parser loads each node into memory, storing its name, attributes, data, and children. Each of these fields can be accessed and mutated by the developer.

2.3 Input

The Luminos Engine makes use of many different input mechanisms, including controllers, keyboards, and mice.

2.3.1 Controller

The only controller currently supported is Microsoft's XBox One controller. All buttons, joysticks, and bumpers/triggers are currently enabled for usage.

Methods List:

```
float getHorizontalAxis()
float getVerticalAxis()
float getHorizontalAxisLook()
float getVerticalAxisLook()
boolean isButtonDown(int button)
float leftTriggerPower()
float rightTriggerPower()
```

2.3.2 Keyboard

Keyboards are fully supported by Luminos, allowing for checking if the key is down, pressed, or released.

Methods List:

```
boolean isDown(int key)
boolean isPressed(int key)
boolean isReleased(int key)
```

2.3.3 Mouse

There are two main functionalities to the mouse, position and button clicks.

```
double getX()
double getY()
boolean isDown(int key)
boolean isPressed(int key)
boolean isReleased(int key)
```

2.4 Rendering

2.4.1 Renderers

There are two divisions of renderers in the Luminos Engine, UI shaders and scene shaders. Each renderer is tasked with a smaller portion of the task. For example, there is a dedicated rendering pipeline for terrain. All of the rendering functions can be accessed by the `MasterRenderer` class, which instantiates once of each type of renderer. Each renderer can be instantiated individually as well, as so only what is needed is created in memory.

2.4.2 Shaders

Shaders are what allow for programmable graphics pipelines. All shaders use the abstract `ShaderProgram` class, which loads, manages, and destructs the shader data. All variable locations are cached by the class, which can be used to access or modify the variable data on the GPU. As with everything else in the engine, if a built-in shader does not do what is needed, the developer can construct their own.

2.5 Window and Devices

The Luminos Engine uses GLFW for its window and context creation. By default, it opens on the primary monitor. The window supports features such as full screen and vertical synchronization. The window manages all callbacks, including the resizing of framebuffers, inputs, and closing.

In order to determine defaults, the information for the primary monitor is polled from Java's built in Abstract Window Toolkit library. This information can be retrieved by the user by instantiating a new `Device` or via the `Window` class.

3 Modified Features

No modified features.

4 Deprecated Features

No deprecated features.

5 Removed Features

No removed features.