Chalkboard

cross-platform and API-agnostic graphics for Java

design, methodology, implementation, and documentation

version 1.0

developed by Zach Goethel

[github.com/zgoethel](https://github.com/zgoethel)

zach.goethel@gmail.com

for Gabby,

because she puts up with me

Contents

[Design Specification 1](#_Toc39761297)

[Description 1](#_Toc39761298)

[Criteria 1](#_Toc39761299)

[Design Model 1](#_Toc39761300)

[Prerequisites 1](#_Toc39761301)

[Module Requirements 2](#_Toc39761302)

[Use Cases 3](#_Toc39761303)

[Project Structure 4](#_Toc39761304)

[Object Design 4](#_Toc39761305)

[General Approach 4](#_Toc39761306)

[Conversational Self-typing 5](#_Toc39761307)

[Initializable, Generatable, Spawnable, and Destroyable 5](#_Toc39761308)

[Pointers 6](#_Toc39761309)

[Context and Window Type Parameters 6](#_Toc39761310)

[Object Signatures 7](#_Toc39761311)

# Design Specification

## Description

The goal of this library is to lower the barrier of entry for GPU-accelerated graphics. Graphical concepts, such as a “graphical context” or a “framebuffer object” are abstracted to a level where APIs are hidden from the developer. This allows a model of application design where the graphical engine is not beholden to any specific graphics API; the developer can easily switch between native bindings at their free will.

Releases of this library (at version 1.0) provide bindings for LWJGL 3 and its GLFW, OpenGL, and Vulkan extensions.

## Criteria

### Design Model

Native bindings for different graphics APIs are separated into modules. Common decorating bindings are in the core module (called ‘:chalkboard’ under Gradle).

An application instance is created by passing a graphics context object to the application backend at runtime initialization.

Each graphical context fits into a common signature, such that all contexts are interchangeable at any point in development.

Objects, such as framebuffers and textures, are produced by a decorated context object passed to the application at setup and every screen-update interval.

The graphical context and its related window provider (such as GLFW to OpenGL) are separate entities, but each graphical context defines limitations on which type of window API it supports.

### Prerequisites

|  |  |  |
| --- | --- | --- |
| LWJGL 3 (≥ 3.2.3) | GLFW bindings  OpenGL bindings  Vulkan bindings  STB font rendering | per-module  dependencies |
| Build platform | OpenJDK 8  Gradle Wrapper 6.3 |  |
| Supported systems | Oracle JRE 8/OpenJDK 8  OpenGL 2.0 or 3.0  Vulkan 1.0 or higher  Windows, macOS, Linux | per-module  APIs |
| Dependencies | JUnit 4.12  JOML 1.9.24 |  |

### Module Requirements

Any library context supplied to the application backend must meet general implementation requirements.

|  |  |  |
| --- | --- | --- |
| Context Modules | GraphicsContext implementation  Define limitations on window API  Multiple instances  Thread-safe (excluding internal engine)\* | generic construction |
| Window API Modules | WindowService implementation  Window implementation  Multiple windows  Thread-safe (excluding internal engine)\*\* |  |
| Pipelines | GLSL compilation  SPIR-V  Vertex, geometry, and fragment shaders  Developer-defined uniforms | abstract/decorated construction |
| Textures | Checked hardware support  Handle lacked dimension support  (non-2n dimensions)  Bindless objects |  |
| Framebuffers | Link render textures  Bindless objects |  |
| Meshes | Mesh and sub-mesh rendering  Memory allocation  Multi-texture  Uniforms  Rendering optimizations |  |
| Object disposal | Memory leak prevention  Relinquish all resources |  |
| UI Modules | Buttons, windows, grab handles, sliders, font-rendering, textboxes, settings/menu construction  Anti-aliasing | using decorated instances |
| Application Backend | Lifecycle management  Context management  Logging  Error-handling |  |

\* Calls for an OpenGL context can only be made from one thread at a time  
\*\* Certain systems limit from which thread inputs can be polled

## Use Cases

The library should aim to support these general use cases. Some items listed here are not yet supported or are speculation on what may be supported in the distant future.

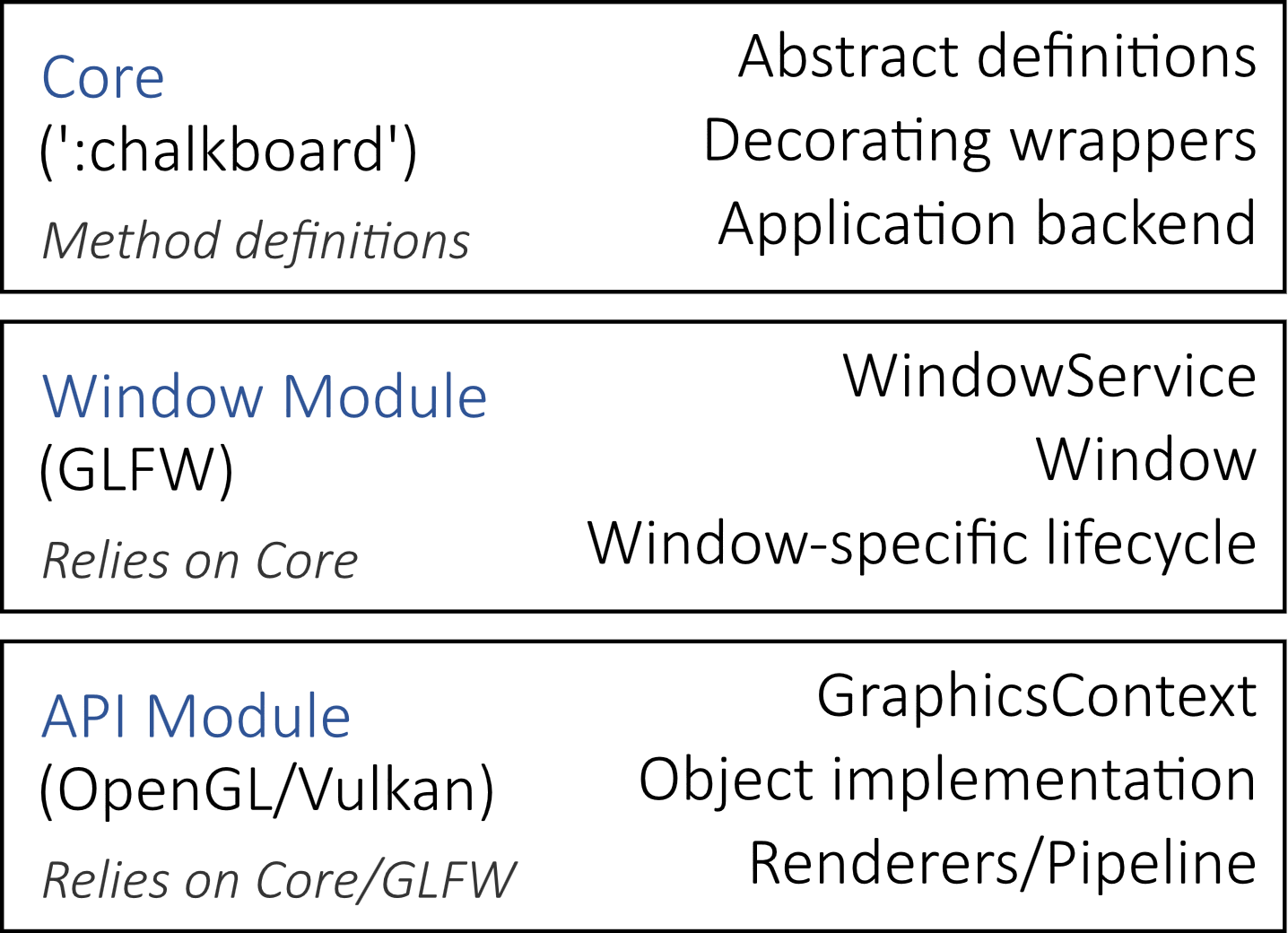
Based on the following use cases, the library should be versatile enough to fit all general-purpose applications in several types of distributions. As part of the modular design of this library, support for embedded systems may be added later as a new module.

|  |  |  |  |
| --- | --- | --- | --- |
| Distribution | Description | Audience | Considerations |
| Specialized Software | Industrial/corporate application | Small | Hardware support across managed enterprise fleets should be fairly consistent; aim to support modern OpenGL and Vulkan.  Use bleeding-edge features if the fleet can reliably support them. |
| Commodity Application/Gaming | Widely available application or a video game | Wide | Hardware type, age, and support varies widely across this audience; aim to support a mid-range feature set.  Do not enable bleeding-edge features by default. |
| Embedded/IoT | Single product with graphical frontend | Wide | Embedded systems are not yet explicitly supported\* |
| Android | Mobile frontend or game | Wide |  |
| WebGL Surface | Advanced website graphic or game | Wide | WebGL and GWT are not yet supported\*\* |

*\* Embedded ARM systems such as the Raspberry Pi and Android devices may be supported in a later 1.x release  
\*\* WebGL and GWT support should not be expected until a later major version release*

# Project Structure

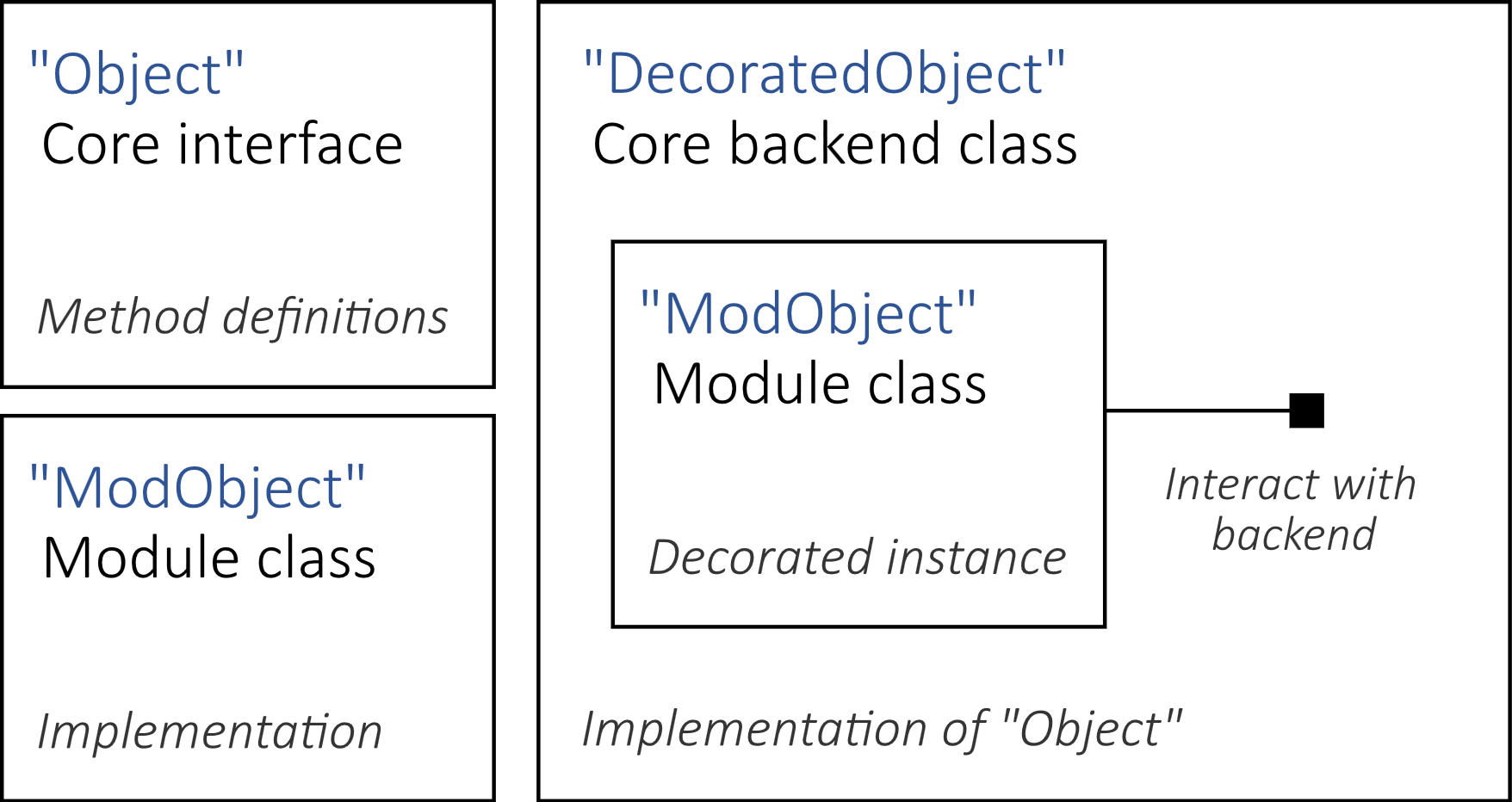
As described in the design criteria, the graphics library is split into several modules. This modular approach allows for light-weight distribution; developers can pick and choose their modules without the whole bill of materials bloating their application.



# Object Design

## General Approach

The signature of a specific set of bindings is defined as an abstract class or interface in the core module. Each extension module must complete the definition of these signatures. Then it is up to the application backend to wrap each object in a decorating container. This ensures consistency across all API implementations.



## Conversational Self-typing

For developer convenience, objects defined in this library should generally return themselves instead of ‘void’. The ‘Conversational’ interface takes one type parameter ‘THIS’, which is the type name by which the object refers to itself. The interface also adds a ‘self()’ method which automatically casts ‘this’ to type ‘THIS’.



The developer can now more conveniently call multiple methods on this object.



## Initializable, Generatable, Spawnable, and Destroyable

These four functional interfaces are simple yet powerful concepts. Each task name generally encapsulates the purpose of the task, but some further explanation is helpful.

|  |  |  |
| --- | --- | --- |
| Initializable | For objects whose class should only be set up once per application runtime. Each class type is cached so its initialization method is only ever called once; invoke this caching behavior by calling ‘initializeOnce()’ instead of ‘initialize()’. |  |
| Generatable | For objects which are set up once for every instance. This includes any object which must generate a pointer. |  |
| Spawnable | For objects which should branch off every time they initialize, such as contexts which create a new context object when generating instead of returning themselves. |  |
| Destroyable | An AutoCloseable interface; release resources here. |  |

## Pointers

Any API object which is handled as a numerical pointer can be wrapped in a class implementing this interface. It takes one type parameter which is the type of the pointer: usually Integer for OpenGL, Long for GLFW.

The pointer itself should be generated in the ‘generate()’ method from ‘Generatable’.

## Context and Window Type Parameters

‘GraphicsContext’, ‘WindowService’, and ‘Window’ have some funky generics going on. They follow the general format ‘CONTEXT, WINDOWSERV, WINDOW’:



These are the type parameters which allow module implementations to specify and restrain which window implementation and APIs they should use:



## Attachment Model

Some objects require direct communication with their parent contexts or services. For example, contexts which use GLFW extend ‘Attachable<GLFWWindow<. . .>, . . .>’ since they tend to require a window instance before a context can be generated. The window is attached with ‘attach(window)’ and the attachment on an object can be retrieved with `this.attachment()’.

Another example is the OpenGL static mesh implementation. Since rendering techniques have changed with GL versions, multiple implementations of the ‘GLStaticMesh’ exist (derived from the interface ‘GLMeshExtension’) which extend ‘Attachable<GLStaticMesh, . . .>’; this allows the multiple implementations to retrieve data stored in their attached parent mesh object.

## Lifecycle

Application lifecycles are defined primarily by window modules. The general stages include the following.

Any object generated by the lifecycle itself will be released by the lifecycle thread. This includes the context, window, and internal system objects such as swap-chains and surfaces. Any application-allocated objects or services, such as the window service, textures, or framebuffer objects, should be destroyed by the application.

All ‘Destroyable’ objects are also AutoCloseable, so try-with-resources construction can help ensure that objects are destroyed.

|  |  |  |
| --- | --- | --- |
| *Initialize* | *pre-init* | *Called before any context is created; good for loading vital assets or starting logs* |
|  | *init* | *Used for defining allocations, pipelines, and objects after a context has been created* |
|  | *post-init* | *Final initialization step—mostly non-distinguishable from regular init in this version; good for setting up right before the lifecycle loop begins (physics, etc.)* |
| *Update*  *(looped)* | *pre-update* | *Called before the context has prepared the frame (before buffers cleared)* |
|  | *update* | *Render and update* |
|  | *post-update* | *Buffer swap and input polling* |
| *Destroy* | *pre-destroy* | *Called before any graphical cleanup* |
|  | *destroy* | *Release application-allocated resources* |
|  | *post-destroy* | *Context destruction* |