Gladius Documentation

Release 0.2.0

Mozilla Foundation

August 03, 2012

CONTENTS

Gladius is a 3D game engine, written entirely in JavaScript, and designed to run in the browser. We leverage existing web technologies whenever possible and where gaps exist in support for games, we develop new solutions.

The engine consists of a core set of functionality that is common to all games and simulations like the game loop, messaging, tasks and timers. Common components like the spatial transform are also provided by the core. More specialized funcionality, like graphics or physics, is encapsulated into engine extensions that are designed to run on top of the core. A common set of extensions is maintained as part of this project, and support for third-party extensions is a strong design objective.

CHAPTER

ONE

ENGINE OVERVIEW

1.1 Engine Layout

Gladius is comprised of several modules, almost all of which are implemented as *extensions*:

- `gladius-core` provides the core functionality of the engine.
- `gladius-input` provides player input functions.
- **`gladius-cubicvr` provides 3d rendering capabilities using the `cubicvr`** library.
- `gladius-box2d` provides physics via the `box2d` library.

1.2 Components, Entities, and Spaces, Oh My!

Gladius uses an Entity-Component system to model the game world. In other words, behaviors, such as rendering graphics, computing physics, and playing sound, are defined by **components**. An **entity** is a collection of components, and represents something in the game world. A **space** is a collection of entities.

Components both perform some function and contain the state for that function. Entities are nothing more than dumb containers that manage their components, and don't carry any state. Components can be configured and swapped around between entities.

1.3 The Game Loop

The game loop in Gladius is split into three phases: input, update, and render.

The **input** phase is for reading input from various devices, including the keyboard, mouse, gamepads, etc. The `gladius-input` extension provides a set of input devices and convenient plumbing for retrieving the input state.

The **update** phase is for updating game state. This includes calculating movement, applying physics, performing collision checks, and other core game logic. During this phase he updater service provided by `gladius-core` will trigger an updateevent on every registered component.

The **render** phase is when graphics are actually rendered to the screen. Typically extensions like `gladius-cubicvr` will handle this phase.

1.4 Services

A **service** is a collection of tasks that will be run during the game loop. A **task** is a function that is associated with a game loop phase and a set of tags. These tags are used to resolve dependencies between tasks; several tasks can be under a `physics` tag, while another task can depend on the `physics` tag so that it does not execute until all the `physics` tasks are complete.

CHAPTER

TWO

EXTENSIONS OVERVIEW

Gladius features an extension system that allows libraries to be created that easily integrate with the engine and provide new functionality that the core library does not provide.

An extension is a collection of three types of items: components, resources, and services.

2.1 Components

Components are chunks of functionality and state that can be attached to entities. The `cubicvr` extension, for example, provides a `model` component that contains both the data for displaying a 3d model and logic to help render it. In order to render an entity to the screen, you attach a model component to the entity and configure it to the model you want displayed.

Components can also respond to events. If a component has an `onX` function (where `X` is replaced with the event type), the function will be called when an event with a matching type is dispatched to the component.

Components can depend on other components in order to function. When a component is added to an entity that does not have other components that it depends on, an error is thrown.

2.2 Resources

Resources are types of data that generally don't change and are shared among several entities. This includes things like 3d models, textures, sounds, etc. Extensions don't define individual resources, but instead define functions that can load resources. The `cubicvr` extension defines resources for things like light definitions and meshes.

A resource provided by an extension is a function that takes in a piece of data and produces some type of resource. This data is typically retrieved via an `XMLHttpRequest`, although this behavior can be customized. The returned data is passed to the resource constructor.

2.3 Services

Services are lists of tasks that are executed during the game loop. Each task in a service defines which phase of processing it takes part in: input, update, or render. Tasks can also define dependencies between eachother, so that tasks that require other tasks to be complete can be scheduled correctly. For example, the `cubicvr` extension defines a render task, tied to the render phase, which handles rendering entities in 3d space.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*