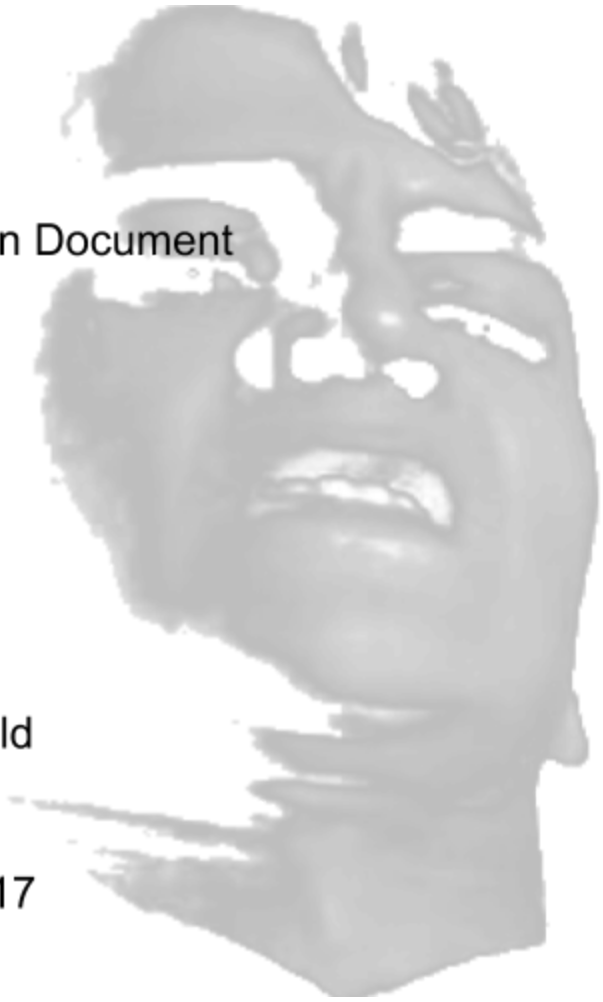
Group 5
Tedm
Software Design Document



Theodore Ahlfeld
David Watkins
Matthew Haigh
Date: 04/28/2017

TEDM Design Document

Theodore Ahlfeld, Matthew Haigh, David Watkins
{twa2108, mlh2196, djw2146} @columbia.edu

Introduction **2**
1.1. Purpose 2
1.2. Scope 3
1.3. Overview 3

Background **3**

System Overview **3**

SYSTEM ARCHITECTURE **5**
4.1 Architectural Design 5
4.2 Decomposition Description 7
4.3 Design Rationale 7

DATA DESIGN **8**
5.1 Data Description 8
5.2 Data Dictionary 8

COMPONENT DESIGN **8**
6.1 State Life Cycle 8
6.2 Game Life Cycle 9
6.3 Event Handling 10
6.4 State Life Cycle 11

HUMAN INTERFACE DESIGN **12**
7.1 Overview of User Interface 12

1.Introduction

1.1. Purpose

There are many video game engines but there is no introductory choice for developers interested in starting out in the industry. Additionally, the video game industry is very competitive and a portfolio of projects is usually required to gain employment. This project serves as an attempt to bridge the gap for entry-level developers looking to start their own video game projects. By providing developers with a basic framework to create environment, player, objects, and events, we will empower others to begin their game-development career.

1.2. Scope

Tedm is designed to give the developer a simple interface for basic functionality. By allowing the user to inherit basic classes and extend them, they are limited only by their imagination. We do not presume to restrict the user to a particular graphics library, but we provide SDL implementation to make game development easier and provide test cases and examples.

This document is intended for individuals that have the desire to get involved in the development of computer game programming. This document need not be read sequentially; users are encouraged to jump to any section they find relevant.

1.3. Overview

The Tedm graphics library begins with the Game class. The developer will extend the Game class and create their own. The game contains a state which determines what objects are in the game and what they do. The user extends states and registers listeners for keypresses and other events. Functions are defined to execute when an event occurs through EventListeners and EventHandlers. Player can be extended to define the players in the game and Object can be extended for any game element that is not an object. Game contains a run() function which executed the listeners and actions.

2. Background

Video games are a large industry and there are many resources available. Existing libraries range from small javascript applets to massive game engines for top corporations.

3. System Overview

The functionality of the Tedm is the utilization of a standard game loop used in practice throughout industry. The context though is the utilization of user designed states that allows for simple game creation without having pain over the heavy game logic.

4. SYSTEM ARCHITECTURE

4.1 Architectural Design

The Game class is the primary game object which represents a game. The primary components within the Game class are the eventHandler, State, and Context. The graphics object is also contained within Game. The Game State and Context represent the condition of the game; what should be rendered, what should happen when keys are pressed, what actions are happening, etc..

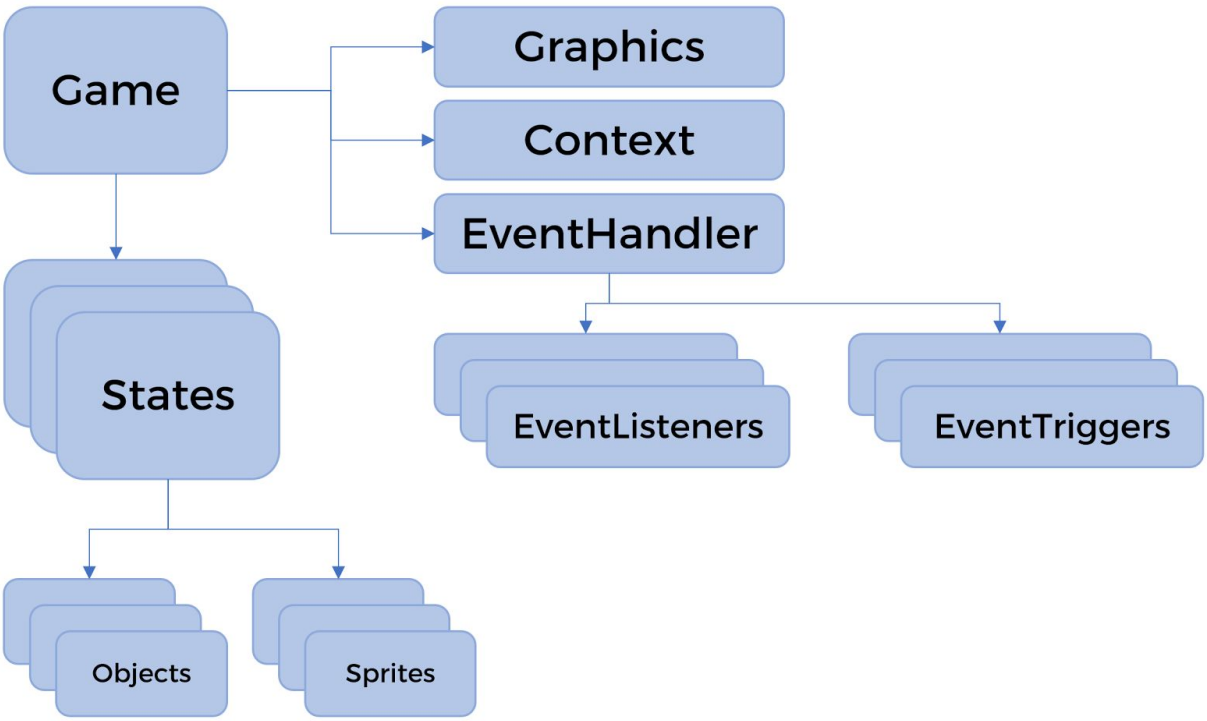
The Context contains hard data such as the frame rate and window size.

The State functions as a template where the user defines virtual functions to overwrite the default functions init(), update(), render(), etc. These functions are executed at the proper times, containing all of the primary logic for how the game operates. Init() sets up the starting condition and objects, update() is called every frame to make the game run, and render() tells the game what to draw to the screen. These are all associated with the state.

The state can be changed based upon registered events. These can be key events, which include mouse interactions, or custom events that occur within the game. The user defines EventListeners with functions that execute when the event occurs. The game contains a main loop which sets the frame rate according to user specifications. A simple timer tracks seconds passed and throttles the rate accordingly.

The Graphics object in the Game contains the necessary data for drawing the game on the screen. This can be redefined for different graphics libraries but it is currently configured for SDL2. All game objects are derived from Object. Object contains the necessary information, such as the image, size, and position. The Player class extends Object to include any information that is specific to a user-controlled player character. The developer extends Player to create game-specific players and the user extends Object to create non-player objects. The update() function will perform object actions and the render() function will determine which objects are drawn to the screen.

4.2 Decomposition Description



4.3 Design Rationale

For game design object orientation is very useful. It is natural and intuitive to refer to items within a game as ‘objects’ and ‘players’. Each object type contains instance variables and member functions designed for the functionality of that object. It is important that a game contain different states to indicate how the game should behave based upon the condition of the game, so the state object follows. The game needs to know when to transition between states so we gave the user the ability to listen for

events and act accordingly. The keypress events are different than game events but we coupled them in the EventListener class by allowing the user to define a generic event. It would have been easier to integrate graphics into the library but we wanted to allow for different libraries and provide that freedom to the user.

5. DATA DESIGN

5.1 Data Description

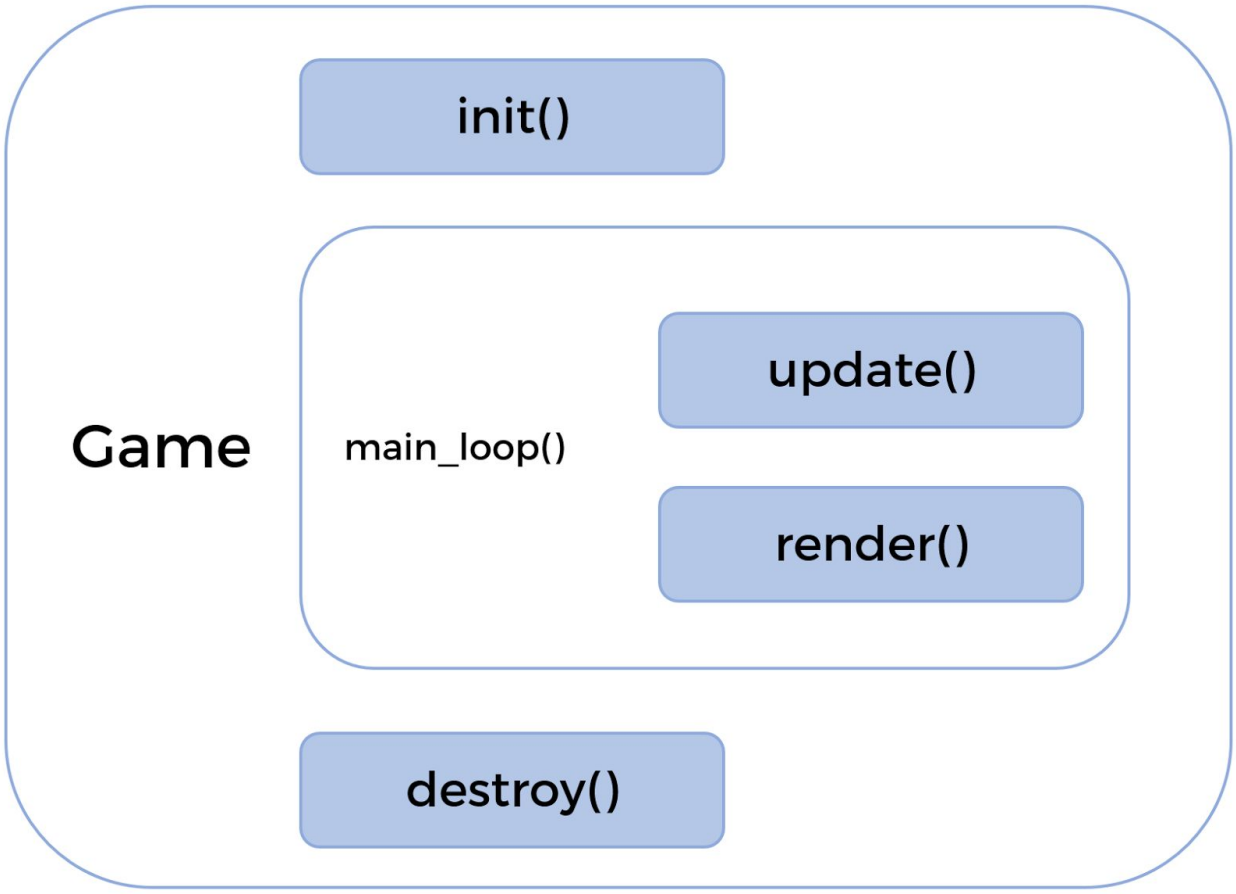
There is no external storage of data outside of running memory. As described previously, the primary Game object contains all of the game data.

6. COMPONENT DESIGN

6.1 State Life Cycle

A game consists of a sequence of transitions between states. A game starts in the start state, which contains EventListeners which trigger transitions to alternate states. The game can be represented as a state machine with transitions corresponding to events. When a state transition is initiated, the prior state EventListeners are destroyed and the new EventListeners are registered.

6.2 Game Life Cycle



The game life cycle is based on the standard C++ principle of RIAA. However due to the requirements to allow for external graphics libraries and other third party libraries Game has both the init() and destroy() functions.

6.3 Event Handling

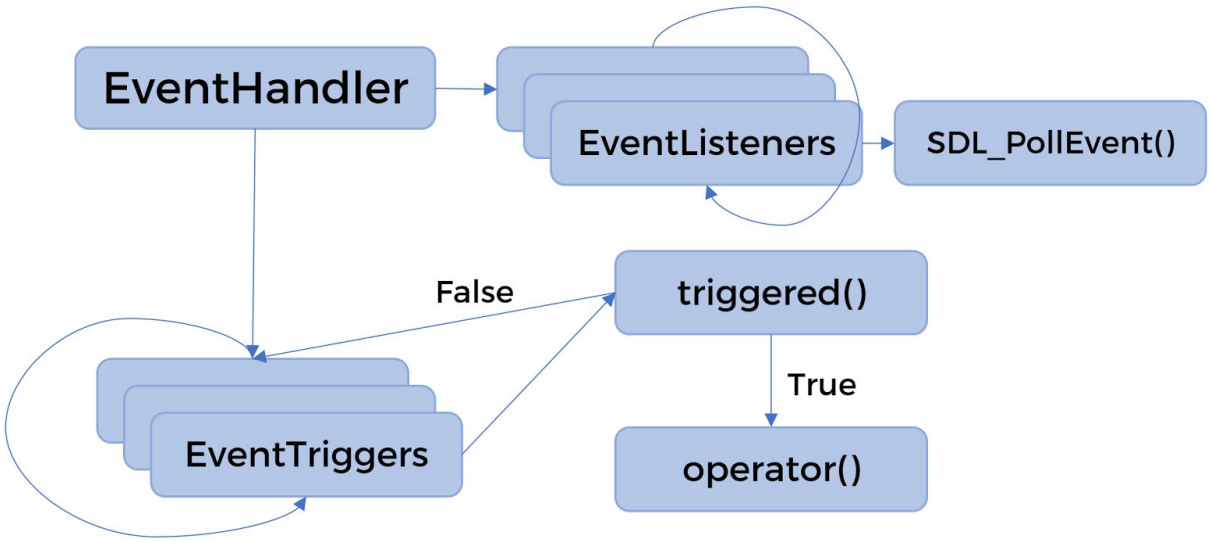
SDL_Events are keypresses, mouse interaction, and other similar events. The EventHandler class registers which events are “subscribed to”, and contains functions checkListeners(), and process(), which listen for and process events, respectively. The user defines EventListener objects which designate a function to execute when the corresponding handler is activated.

It is an important decision of the user to ensure that the correct EventListener class is used. The EventTrigger class represents generic predicate that are checked for game certain conditions to manifest in the state that the user can defines. If the predicate is

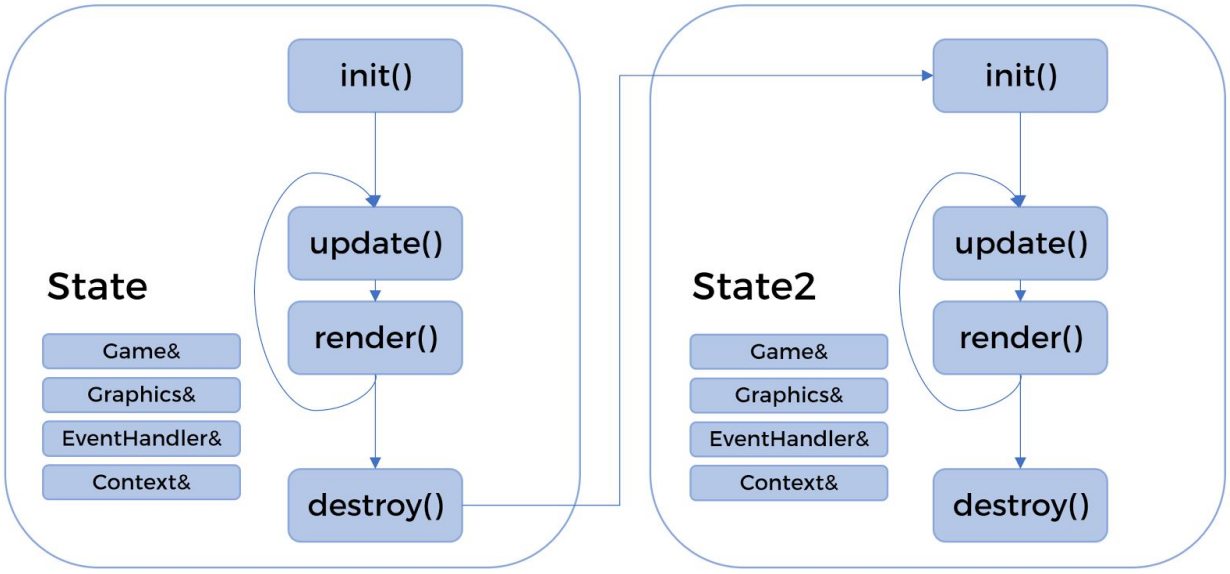
met then the EventTrigger's operator() will be executed in the Game's event checking functionality.

There are also important EventHandling built in such as keypress, mouse events, and OS hooks. The above mentioned events are system dependant since different operating systems have incompatible interfaces for finding out these states. Tedm allows for OS agnostic handling to allow the user to create portable games that can be played on any hardware.

The decision of which event to use is important in the design of the user's game. Events are powerful in Tedm but leads to abuse if the game is not properly designed around them.



6.4 State Life Cycle



The State life is the most important aspect of Tedm and governs the user defined behavior as called by the game class. The state is where all the game objects live, such as Players, Non-Player Characters, graphics, and any other data that needs to be determined by the state.

The init() of each state is where the user must defined its working environments. The init function should contain player adding and constructing. EventListeners/EventTrigger that apply to this state should be registered within the game reference. This is important as the user does not want an event to be triggered that only makes sense in the context of a different state. Init() also must handle setting up the graphic to be displayed as drawable objects will be drawn through the states render function. Init() should set up the context properly for the game object, such as changing the framerate and the member data of the game's Context. Lastly init() should be where the user defines everything else that is needed by the update() function and all useful EventListeners

Update() handles what many would assume to be the Game's main loop logic. The state's update() is what determines what is a turn (a time delta, or a specified event such as in turn based RPGs) and what needs to execute during each iteration. Update() is in charge of updating the state of the State and changing relevant states of the game objects with State. It is common to handle collision detection and handling within update, but it is also acceptable to handle collision detection within EventHandlers, the

proper handling method needs to be thought out by the user in designing how handling will be handled. Can you handle it?

Render() is in charge as drawing the state to the screen. The frequency of render() being called is determined by the targeted frames per second contained within the game's Context. Drawing includes the background, environment, and all drawable game objects. As default all game objects have a draw() member function and it is advisable to utilize this function when drawing game objects to screen.

7. HUMAN INTERFACE DESIGN

7.1 Overview of User Interface

Tedm very little interaction from the user to do complex operations and executions throughout the code. The key aspects the user must overcome is the creation of EventHandlers, and State. State is the main interaction with the game class contains the current slate, as it ever so cleverly named, of all game objects and determines transitions. State is used by Game through its virtual function pointers, so it is imperative that these be assigned appropriately, particularly init(), update(), and render() as these are guaranteed to be called by Game. EventHandlers is a collection of functors that the user must define when a certain trigger has occurred. Commonly these will be used for creating predicates for conditions met within state, such as collision, keypresses, and quitting.

7.2 Multitude of Events

A user has at their disposal a series of events that give them a variety of different input methods, including mouse clicks, key presses, mouse movement, quitting, and customized triggered events.

Builtin are Events that are triggered by Key presses, and mouse operations. This is extremely helpful as it offers portable Key State and Mouse State interface, since Linux and Windows are incompatible in their ABI. Certain events are hooked into OS operations similar to ExitEvents. These listeners allow for the user handle OS events properly while still safely releasing all resources and putting the game state in a safe state to exit.

EventTriggers are where the real power of event handling take place within Tedm. This is a special functor that has access to data that is arbitrarily passed to it. The functor has a predicate

signature as bool triggered(). If the predicate is true then the operator() is called to do event handling.

No other game creation library has such a robust event handling system and really is why Tedm should be an viable contender in the realm of game engines.