|  |
| --- |
| UAS Software Development Project 1 |
| Ninja Game |
| Software Design Document |

|  |
| --- |
| Team members:  Dair Baidauletov  Fayjus Salehin  Hung Dao |

Contents

[1. Introduction 2](#_Toc322695322)

[1.1 Scope of The Project 2](#_Toc322695323)

[1.2 Document Overview 2](#_Toc322695324)

[1.3 Reference Material 2](#_Toc322695325)

[1.4 Definitions and Acronyms 2](#_Toc322695326)

[2. User Interface 2](#_Toc322695327)

[2.1 Site Map 2](#_Toc322695328)

[2.2 Page Layout and Design 2](#_Toc322695329)

[3. Database 2](#_Toc322695330)

[4. Client Side SW Design 2](#_Toc322695331)

[5. Server Side SW Design 3](#_Toc322695332)

# Introduction

## Scope of the Project

We are making a mini game for entertainment purposes. Ninja game is a simple web based 2D endless running game[1] with a database system that allows player to track top scores. The game is using currently top popular internet technologies and is capable to be played by people all over the world.

## Document Overview

This document describes the main design basis of our project. Our team consists of motivated students diving into a software development process for the first time. The interactive mini-games are easy way to connect with people and they rely hardly on the social interests and age groups, which makes the product universal and unrestricted.

## Reference Material

[1] — en.wikipedia.org/wiki/Platform\_game

## Definitions and Acronyms

# User Interface

## Site Map

Describe how do the pages in the site link to each other.

## Page Layout and Design

Describe the layout(s) of the page design(s) used in your site.

# Database

Describe and illustrate the whole database design of your site. All tables, table structures and relations should be presented.

We have one table, displaying player names and scores.

# Client Side SW Design

After couple of experiments of using the native HTML rendering and animation techniques, we found out that in order to make the game playable, we should use some rendering *Javascript* framework. Our choice fell on modern **p5.js** framework and using it makes such nuances as smooth animation, easy controllers, audio background etc. not exceedingly complicated.

The framework requires some preloading and setup methods to handle variable declaration, assets attachment and assignment operations. The next *draw()* function is autonomously called once per rendered frame and it keeps tracking of all the processes and occurrences in code with possible *if* constructs. This helps us to concentrate on actually making game mechanics and loop flow work properly and with no extra exceptions or errors.

There are 9 javascript classes in the source code — 1 Ninja class for player , 1 abstracts class Collider, 2 subclasses of Collider — Enemy and Pickup, and actual 3 children of Enemy (Shuriken, Kunai, Katana) and 2 of Pickup (Health, Coin).

C:\Users\lenovo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Untitled Diagram.png

The reason for creating an abstract class Collider is in optimization of code. As it is all of the collider shall have *move(), show()* and *crash()* methods, therefore it is more convenient to put them into an abstract class first.

The reason for creating two different Enemy and Pickup classes, although they require similar value of integer, is in creating a logical distinction in uses of mentioned classes. Therefore, the Enemy classes affect the health of Ninja (deteriorates), whereas Pickup may perform any other interactions.

* **Ninja** is the main controllable player-object.

Data fields:

*(int) height, width, x-position, y-position, healt, score*

Member functions:

*move()* and *show()* functions are called in main loop and are respectively responsible for moving the coordinates according to player inputs and displaying the corresponding animation of ninja. The move function also sets the boundaries to the movement of player, illusion of ground and gravity imitation.

* **Collider** is an abstract class created for being extended by class **Enemy** and **Pickup**. The reason for doing so is to optimize code by not repeating similar functions. When created it first randomizes the position of object.

Data fields:

*(int) x-position, y-position, speed, width, height*

Member functions:

*move()* — simple arithmetic function for changing the *x* coordinate according to *speed* parameter;

*crash( Ninja )* — takes the *Ninja* object as a parameter, checks for collision to occur and performs actions for any of particular case by checking the type of an object with ‘*instanceof’* command. Therefore, if player have crashed into a *Coin*, his score is increased, if crashed into *Katana* his health is decreased by a certain value.

* **Enemy** (inherits Collider) is used for imitating obstacles and decreasing Ninja object’s health field when colliding.

Data fields:

All data fields inherited from super class *Collider.* The new data field *damage* is an integer value which is decremented from the player’s health when crash occurs.

Member functions:

All functions inherited from super class *Collider*.

* **Pickup** (also inherits *Collider*) is used for rising the score/health of the object Ninja when colliding with it.

Data fields:

All data fields inherited from super class *Collider.* The new data field *value* of type integer describes the incrementation of health/score.

Member functions:

All functions inherited from super class *Collider*.

Classes **Shuriken, Katana** and **Kunai** inherit class *Enemy.* The methods *show()* are overridden to display different animations. Values of some data fields are changed.

Classes **Health** and **Coin** inherit class *Pickup.* The methods *show()* are overridden to display different animations. Values of some data fields are changed.

# Server Side SW Design

Describe here the design of server side software. For server side, you can describe similar things as for the client side.