5-2016

An Original Computer Game Incorporating Optical Illusions

Jacob Butterfield
Utah State University

AN ORIGINAL COMPUTER GAME INCORPORATING OPTICAL ILLUSIONS

by

Jacob Butterfield

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Minghui Jiang Major Professor	Dr. Vladimir Kulyukin Committee Member
Dr. Dean Mathias Committee Member	

UTAH STATE UNIVERSITY
Logan, Utah

May, 2016

ABSTRACT

An Original Computer Game Incorporating Optical Illusions

by

Jacob Butterfield, Master of Science

Utah State University, May, 2016

Major Professor: Dr. Minghui Jiang
Department: Computer Science

Computer games have a diverse range of genres and implementations. Many of the most popular computer games consist of time-killing puzzles targeted at mobile devices. Each game has a well-defined challenge that a user must solve in order to progress.

The goal of this project is to investigate and implement a new obstacle for use in computer games, namely optical illusions. One illusion in particular, the Fraser spiral, was especially considered. Before this project, no other computer game integrating this optical illusion has been attempted.

Included in this report is an overview of the design, implementation and testing involved in creating this game. A gameplay method is introduced that effectively makes use of the Fraser spiral illusion. Tests were conducted in order to investigate the impact of the illusion on user performance. The results of these tests demonstrate the plausibility of using optical illusions to increase gameplay difficulty.

(41 pages)

CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Computer Games

A computer game is a virtual representation of a physical task for challenge and amusement. One may participate in a computer game in order to improve his skills, compete in professional contests or to simply pass the time.

Popular computer games combine computer graphics with distinct challenges in order to create a difficult, yet enjoyable, experience. Game genres range from complex, life-like simulators of sports, physical combat or role-playing to fun puzzles with bright themes and cartoonish animation. Each year, several games are developed that target these structured categories. The intent of this project is to design, develop, and test a brand new class of computer game, one that integrates optical illusions.

1.2 Optical Illusions

An optical illusion is an image whose true composition is different from its perception [1]. When one views an optical illusion, he may perceive objects or properties that do not exist because of the the brain's attempt to accommodate diverse imagery [2]. Some well-known optical illusions may cause the viewer to discern non-existent images, interpret a three-dimensional image as a two-dimensional image or vice versa, and even fail to recognize the same color in different environments. The optical illusion that provides the basis of this project is known as the Fraser spiral or false spiral illusion [3]. There are several versions of this illusion, as seen in Figure 1.1, and this project relied exclusively on the version labeled (c). This version of the Fraser spiral illusion can be described as a series of concentric circles comprised of square outlines with alternating colors. Each square has an angle relative to its position in the circle with an additional offset. This offset value alternates across each circle,

causing no adjacent circles to be composed of squares with the same rotational correction. When viewed in its entirety, the concentric circles appear to be overlapping. This project attempts to use the false entanglement property as the basis for a challenge in a computer game.

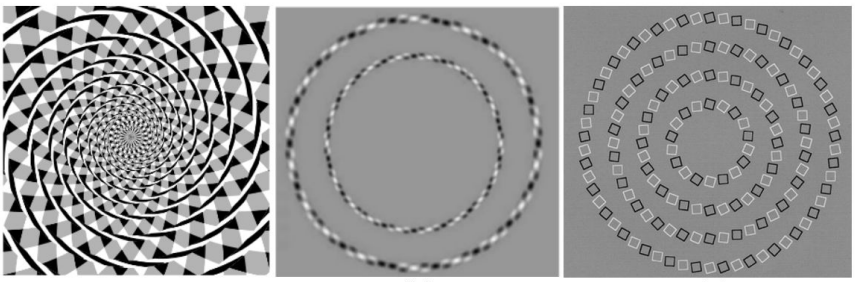

Figure 1.1: A few versions of the Fraser spiral illusion [4] [5] [6]

1.3 Development Overview

The process of creating a computer game that incorporates the Fraser spiral was comprised of several steps. Among these, were design considerations, prototypes, software development and tests. Each of these steps will be discussed in further detail in this report. The initial design and idea were given by Dr. Dean Mathias. From there, a prototype was created using the Unity game engine. After the prototype was approved, a richer version was designed and developed using the web-based Turbulenz game engine [7]. The primary focus of this report will be on the implementation using this tool and the results in testing against a version that lacks the illusion properties.

First, an overview of the technologies utilized in this project will be described in detail. Next, the Turbulenz game engine itself will be examined, including its workflow and libraries. After the technologies have been established, the overall architecture of the game will be given. From there, several proposals for gameplay will be described along with the project development process. Because the core gameplay was the result of experimentation, these two areas of will be discussed together. From there, data generated from empiri-

cal tests will determine the effectiveness of the illusion on user performance. Finally, a conclusion of the project will be given with a recommendations for future work.

CHAPTER 2

Tools Utilized

2.1 Overview

This section explains and details the base technologies used in the creation of this project. Included are descriptions of how each component works, the manner it was used in the project and the reasoning behind its utilization. While not an exhaustive description, the principles explained are intended to give a clear synopsis of each technology.

2.2 JavaScript

Foremost among the tools used in this computer game is JavaScript, which is the programming language that makes up the entirety of the project. JavaScript is a fully featured programming language that can be executed in modern web browsers [8]. It supports several popular programming methods, including object-oriented, imperative and functional paradigms [8]. JavaScript was initially designed to be a scripting programming language. This means that its commands are interpreted and executed one line at a time. Other languages, such as C++, are first translated into machine code and then executed in their entirety on a processor. JavaScript also distinguishes itself from other programming languages because it is dynamically-typed. This means that a single variable may change its type as a program executes [9]. JavaScript is typically associated with web-based programming because of its ability to run in a browser.

When used in conjunction with a webpage, JavaScript code is executed on the client instead of on the server. This means that when a user requests a webpage, it is sent from the server, or the computer hosting the page, to the requester's browser. After it has been loaded, any JavaScript code that may accompany the page is executed on the user's computer instead of on the server. This reduces the computing load on the server and may

enable it to respond to more requests. New technologies have been created to improve the performance of JavaScript code inside the browser. Among these is the V8 engine that operates in the Google Chrome browser [10].

2.3 Chrome Browser V8 Engine

This project was extensively tested and developed using tools available in Chrome, a well-known web browser created by Google. A central feature of this browser is fast performance. Therefore, one of the most important technologies available to web developers is the aforementioned V8 JavaScript engine. The goal of the V8 engine is to improve on performance of JavaScript by compiling its commands into machine code that executes directly on the processor [11]. Although JavaScript was initially designed to be interpreted line by line, the V8 engine allows it to act like a compiled language, such as C++. Having optimized and efficient JavaScript code is essential to game development because of the intensive operations involved, including physics simulations, collision detection, and graphics rendering. Accordingly, the performance provided by Chrome, especially the V8 JavaScript engine, was important to the development and testing of this project.

2.4 Git

Another tool that was used extensively in this project is the version-control system Git. When creating software, it is essential that a developer, or a team, has the ability to apply incremental changes on a functional code base. This allows the developer to add new features or make modifications without negatively affecting the software. Once all changes have been added and successfully tested, the code base can be extended to include them. This idea is known as source or version control and Git is a program that facilitates this process. It works by storing a set of user-defined files in a database and then adding new entries to the database as changes are made [12]. When a developer wishes to add a new feature, he may download the most recent code from a computer that hosts the project, make and test the necessary changes, and then upload his modifications to the hosting computer [12]. Keeping track of these changes enables a developer, or a team, to maintain

a functional, up-to-date version of the project, to create and test new features, and to preserve the entire history of the software. For this project, the code was hosted on Github, a popular hosting site for Git repositories. Although the project was developed by only one person, new features were added and committed routinely to the hosting server. As a result, a complete record of the development of the project is accessible. Git is used extensively in industry for code collaboration and management and was an essential tool for this project.

The final technology that was used in this project is the Turbulenz game engine. Because this tool was so tightly coupled with the game, a deeper discussion of it is given in the next section. Overall, this project relies on several underlying technologies in its implementation. Each makes up an integral part of the overall project and its development. Together, they represent a microcosm of industry standard tools and techniques for game development.

CHAPTER 3

Turbulenz Game Engine

3.1 Overview

Turbulenz is a free and open-source game engine for the HTML5 web standard [15]. Like other popular game engines, such as Unity, or Unreal, it provides a number of libraries for essential game components, including animation, physics, audio, and resource management [13] [14] [7]. In addition to these libraries, Turbulenz also comes with a number of tools for hosting resources, debugging and consolidating code. Being a web technology, games designed using this engine can easily be ported to a number of platforms, such as desktop and mobile, while retaining the same code base. In the beginning of this project, the decision to use Turbulenz was based on the functionality it provided and the ability to use it across several platforms.

The Turbulenz game engine is built on several standard web technologies. Instead of targeting a specific console or device, this game engine is designed to run entirely inside of a web browser [7]. At its core, Turbulenz is implemented in JavaScript and provides a simplified interface to browser libraries such as WebGL and the HTML5 canvas [7]. Therefore, any browser that is compatible with these technologies will be able to run a Turbulenz game. This greatly simplifies the development process because no special accommodations need to be implemented for varying architectures, screen resolutions or operating systems. Games that are developed using the Turbulenz engine are written entirely in JavaScript and generally run in a single webpage.

In addition to being fully web-compliant, Turbulenz is a modular game engine [16]. This means that each component, or module, provided by the engine is independent and provides objects and methods to aid in game development. However, which modules are used is a decision made by the developer [16]. This allows each game to utilize only the pieces of the

engine that are essential, while ignoring unnecessary components. For example, because of the two-dimensional nature of the illusion used in this game, only two-dimensional rendering libraries provided by Turbulenz were included, while all of the three-dimensional rendering libraries were excluded from the project. This reduces the code base and overhead of the entire project, allowing it to be lighter and simpler.

3.2 Workflow and Tools

Due to the expansive nature of the Turbulenz game engine, a workflow has been created to improve developer efficiency. This process includes using Turbulenz specific tools to build, test and release new features and versions of the game throughout the course of development. In order to build a game using Turbulenz, an SDK must be downloaded from the official game engine website. Included in this SDK are all game engine modules, build scripts for consolidating code, a simple server to host resources and several examples of game implementations. With the SDK downloaded, the first step is to run the default server using the executable provided. The end goal of Turbulenz games is to reside on a publicly accessible server and to be played by many. Therefore, the default server provided in the SDK not only hosts local versions of the game, but also acts as a prototype for the deployed final product [17]. By default, the local server is designed to run on port 8070. Once the server has been started, the developer can navigate to the URL `localhost:8070` in a browser on the same computer to view a landing page with all Turbulenz applications, or games, that have been configured to work with the engine. An image of this landing page is included in Figure 3.1. Each application has various options for management, including links to play the game locally, metrics on loading performance and deployment options.

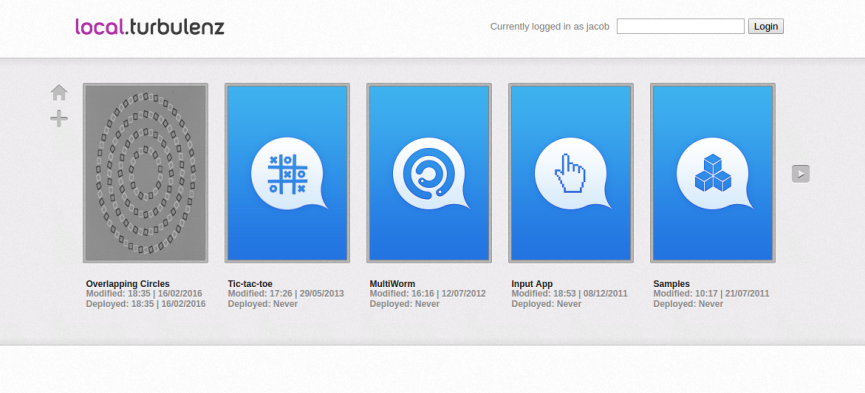

Figure 3.1: The Turbulenz local server landing page

With the Turbulenz server running, local games can be added and modified as their development progresses. The first step in building a Turbulenz game is to import the necessary modules to the main application file. As previously stated, there are several modules provided by the Turbulenz game engine that provide a great deal of functionality. At the time of this project, there are eighty-three modules available for use, all of which have been implemented in JavaScript. Once the specific modules have been selected for the project, they marked for inclusion using JavaScript template syntax, along with any other JavaScript files written by the developer. These dependencies are then compiled into a standalone project using tools provided by the Turbulenz SDK.

All of the tools available in the SDK are included in the Turbulenz development environment. The development environment is an executable that is not to be confused with the Turbulenz local server. This environment provides tools for building and packaging the game, while the local server allows for testing it in the browser. During the game development process, it makes sense to have both the server and the development environment running simultaneously. This allows the developer to modify the game code, rebuild it and then test it in the browser. The tools provided by Turbulenz are manifest as a series of shell commands for performing different actions. Among the most important of these operations are the *maketzjs* and *makehtml* commands. These tools are used once the developer has reached a point where it is necessary to test the latest changes in the browser. The *maketzjs*

command accepts the main application file as an argument, finds all of the dependencies listed in its template markup, and then combines these into a single JavaScript file [18]. The *makehtml* command then creates an HTML webpage to reference this monolithic JavaScript file [18]. Together, these two files compose a game that can be executed in nearly all modern browsers. The completed game can then be tested and marked for improvement, thus reinitializing the development cycle. Therefore, the process of selecting modules, importing them, building the project, testing it and receiving feedback becomes the central workflow of the Turbulenz game engine.

3.3 Modules Utilized

The specific modules used for this project dealt with resource management, physics simulations and rendering. Of the many libraries provided by Turbulenz, only six were necessary to use in the development of this game. Foremost among these modules is the *MappingTable* library. The mapping table is a standalone JSON file that maps all of the necessary resources to accessible objects. Sprites, two-dimensional images used for rendering, are an example of these types of resources. Normally, code would have to be written to load each sprite individually. However, the mapping table module abstracts all of this functionality inside an easy-to-use interface. In addition to this library, the project uses two other modules for loading called *Turbulenz Game Services* and *RequestHandler*. To be brief, the *Turbulenz Game Services* library enables the mapping table to request resources, and the *RequestHandler* simplifies these requests [19].

Two other modules included in the project introduce objects for two-dimensional physics simulations. These two libraries are called *BoxTree* and *Physics2DDevice*. Turbulenz also provides three-dimensional physics objects, but these were not necessary in this game. Computer games often emulate elements of the physical world, including gravity, friction and collisions. Fortunately, the *Physics2DDevice* library provides the functionality to simulate each of these. At its core, this module provides a base physics object called a *RigidBody*. Each of these objects has several properties that can be either manually set or automatically updated, such as rotation, velocity, position, force and mass [20]. A collec-

tion of *RigidBody* objects are contained by a *World* object. Much like the physical world, this item may have characteristics that act upon its *RigidBody* objects, such as gravity. In addition to these properties, each *World* also has several methods for managing its associated *RigidBody* objects. These include automatically updating their positions, velocities and rotations, as well as detecting collisions with other objects [21]. Each of these abilities were useful in the development of this game.

The final Turbulenz library used in this project was for sprite rendering and is called *Draw2D*. This module provides a number of objects and methods for scaling, displaying and, of course, drawing. The *Draw2D* base object can be configured to scale all images to the current screen size while retaining the same aspect ratio, allowing the game to work on a variety of platforms. Additionally, the viewport of the entire screen can be manually set. This means that only objects with positions inside of said viewport will be rendered. Furthermore, the *Draw2D* library provides a simple interface, called *Draw2DSprite* for two-dimensional sprites that can be associated with a *RigidBody*. This allows each sprite to adopt some of the properties of its corresponding *RigidBody*, such as rotation and position. Automatically updating a sprite based on its underlying physics object greatly simplifies otherwise complex rendering code.

Turbulenz provides a number of useful modules for use in game development. Fortunately, each of its libraries is loaded on a need-only basis, reducing overhead. In total, only a small amount of the total functionality provided by this game engine was utilized. However, even this minute adaptation simplified development and testing.

CHAPTER 4

Project Architecture and Implementation

4.1 Overview

This project was built using a number of common software design principles. Each concept employed was chosen because it fit the project specification. Therefore, the overarching architecture differs from each individual component therein. This section first explains the high level structure of the game and then describes the specifics of each component. A synopsis of each software engineering technique enlisted is given, along with the details of its implementation.

4.2 Modules

4.2.1 Modular Design Pattern

At the highest level, this project follows the modular programming pattern. This design method encourages the creation of several independent software components that act as building blocks for an application [22]. A single component may build on other modules, but no two items should depend upon each other. Generally, each component provides a static interface that can be referenced by other modules [22]. An application may be an overarching central module that acts as a container for any supporting items and dictates when they are used [22]. Additionally, an application may house resources needed by other modules, allowing these resources to be reused throughout the entire program.

There are several reasons to design software using this pattern. First, each individual component may be developed apart from the other units, allowing a developer, or team, to create and maintain modules separately [22]. Additionally, any components that may reference another module are not affected when that module changes its inner functionality [22].

In the worst-case scenario, a module's interface changes. However, it is much easier to adjust a few method calls, than to refactor an entire program. Finally, modular programming allows components to be reused across different applications [22]. A well-designed module may be incorporated into a number of different programs because of its independent nature.

4.2.2 Project Modules

This project has several modules that are contained within a main application. Each of these components provides a consistent and easy-to-use interface and is designed to run independently of other items.

4.2.3 Application

The central component of the game is the *Application* module. It is the main entry point for the entire game. The *Application* object provides an interface consisting of two methods, `create` and `shutdown`, which serve to abstract its entire functionality. This module also encompasses and manages all other the auxiliary components and game resources.

The `create` method of this module first instantiates all of the Turbulenz objects used throughout the game. This allows game engine components, such as input handlers and renderers, to be shared by the various modules. Once these objects have been created, the resources from the mapping table are loaded into the application. In this project, the only resources that need to be loaded are sprites. They may be requested and sent from the server, or they may come directly from the browser cache. After all of the resources have been loaded, the auxiliary game modules are initialized by calling each of their `init` methods. As mentioned earlier, the *Application* object is a container for supporting modules, one of which is the *Menu* component. Therefore, after all of the resources have been loaded and the modules initialized, the `show` method of the *Menu* is called, causing a menu to be displayed to the user.

In contrast to the `create` method, the `shutdown` function clears all resources and objects used in the game. This subroutine is automatically called when the webpage is closed. When shutting down, the *Application* component iterates through each game module and

calls their respective `shutdown` methods. This ensures that the application exits properly, regardless of what the user is doing when the page is being closed. Once each module has been stopped, the Turbulenz engine is flushed. This causes the browser to release unused memory that may have been allocated for the game [23]. Any remaining Turbulenz objects are then set to the special JavaScript value `null`, allowing all memory they occupied to be freed as well. In summary, the `shutdown` method provides an easy way for the browser to quickly clean up after the game has been exited.

4.2.4 Auxiliary Modules

Now that the main *Application* has been explained, each supporting component is examined. The four auxiliary modules are *Menu*, *GameOver*, *HighScores*, and *Game*. As a general rule, each module represents a different screen that is shown to the player. For example, the *HighScores* component displays the best user performance, while the *GameOver* module informs the user that the game has been terminated. Each of these items has its own interface, and is designed to work independently of the other components. However, there are several direct references between modules that allow the user to navigate the application. As previously stated, each of these ancillary modules has its own `init` and `shutdown` methods, which bring consistency to the program. A `show` method, which displays a component's associated screen, is also common among these items. While these supporting components share a few methods, they also have numerous important distinctions, and will thus be analyzed individually.

4.2.5 Menu

The first auxiliary module to describe is the *Menu*. As expected, the *Menu* is the landing screen of the game. It gives the title of the game as well as navigation options. The `init` method of this component sets private data members and creates menu items. A menu item is a combination of positional information, a sprite, and a function callback. All of the menu items are stored in a dictionary by the name of their sprite. This allows them to be accessed using static keys. The associated positional data takes into account the location

of the sprite on the screen as well as its size. This allows the *Menu* to detect whether a user has clicked on a particular item, which will cause the associated function to be executed. For example, when the *New Game* menu item has been clicked, the *Menu* is canceled and the *Game* screen is displayed. After all of the menu items have been initialized, the main screen can be shown.

The `show` method of the *Menu* module is associated with rendering an updated version of each menu item. Before rendering, this subroutine adds click and touch event listeners to the screen. An event listener is a function that is automatically executed when a particular event occurs, such as a mouse click in this case. Once the listeners have been registered, a continuous loop is started. This loop has two important functions called `getInput` and `render`. When the `getInput` method is called, it updates the input queue object to check for any mouse clicks from the user. If there are any, the callbacks registered with the mouse clicks are executed. This pattern allows input to be processed synchronously with rendering. The `render` method on the other hand is responsible for drawing each menu item. It uses Turbulenz graphics objects to clear the previous screen and then render the sprite associated with each menu item in its correct position. Running the render method in a loop allows the menu to be responsive and scale sprites when the browser is resized. The screen shown by the menu is included in Figure 4.1.

When a user selects a menu item the menu is canceled and the item's associated module is shown. Canceling the module involves stopping the continuous loop and then removing the mouse click event listeners. The loop is stopped because it is assumed that the newly loading module will render its own components. Likewise, the event listeners are removed in order to avoid conflicts with other modules. The *Menu* object has references to the *Game* and *HighScores* modules, allowing the user to start a new game or view the high scores directly from the menu.

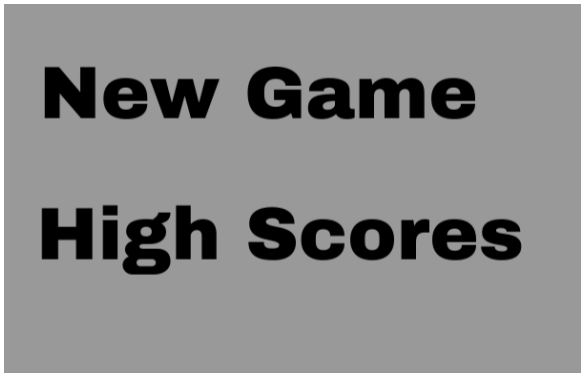

Figure 4.1: The Menu screen

4.2.6 Game

The next supporting component to examine is the *Game*. This module is best described as a container and controller of levels. The *Level* objects have a different architecture from the other modules, and will be described later in this report. On initialization of the *Game* module (by calling its `init`) method, this component prepares its member Turbulenz objects and then populates a private list of *Level* object references. This construct allows the program to create and load each *Level* object dynamically, which reduces the overall amount of memory required by the game. Once the list of *Level* references has been populated, it is never modified throughout the duration of the program. A private `_level` variable stores a reference to the currently operating *Level* object, and is updated as the game progresses. There is an additional private variable called `_levelIndex` that stores points to the current level in the private list of levels. This counter is incremented as the user passes each level.

The `play` method is the main entry point to the *Game*. Much like the `show` subroutine of the *Menu* component, this function first adds mouse click listeners to the screen. After that, it also initializes the first *Level* object. Each of these objects is instantiated using the private method `loadLevel`. When a level is created, the game first checks to see if there are any more levels in the list by comparing the `_levelIndex` counter to the total number of *Level* objects. If the user has completed all levels, then the *Game* module is canceled and the *GameOver* screen is shown. Otherwise, the next *Level* object is instantiated and

the `_level` private variable is updated. Once the first level has been loaded, the game loop is started. Using Turbulenz tools, the game loop is set to run at 60 frames per second (FPS) which is common for video games. A game loop is a construct for ensuring that a game is continually receiving input, updating physics objects and logic, and rendering the most recent game world [24]. In this project, the methods for updating game objects and rendering are abstracted by the *Level* items. These methods and others are common to all levels due to their code structure. An screenshot of the first level being rendered is included in Figure 4.2. Each *Level* object has the ability to report its score and completion time. Once a level has been flagged as passed, the *Game* module increments the `_levelIndex` variable and loads the next level by calling the `loadLevel` method. In addition to checking for level completion, the *Level* objects also cause the *GameOver* module to be shown when the user has run out of time and the game is over.

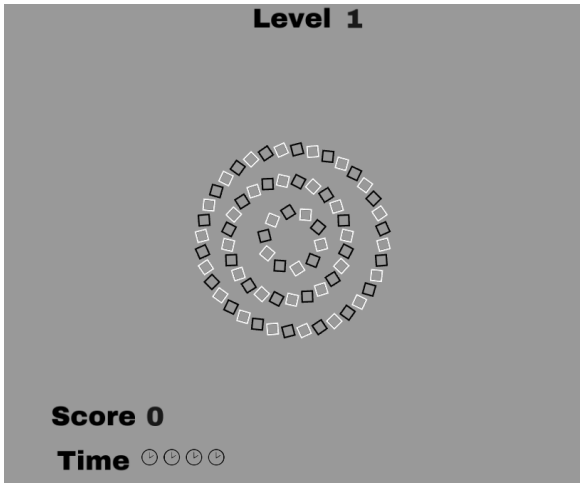

Figure 4.2: The first level screen

When the user has completed the game or has run out of time, the *Game* module transitions to the *GameOver* screen. First, the game loop is canceled using Turbulenz APIs and the mouse click listeners are removed. Any references to a currently loaded level are also disposed of. This saves memory and allows the game to be loaded from the beginning when a new game is started. Finally, the cached score is passed to the *GameOver* module, which then takes effect.

4.2.7 Game Over

The *GameOver* module is the final screen of the game. It serves several purposes. First, it informs the user of his final score. Additionally, this component is responsible for saving the highest scores. As new scores come into this module, they are compared with any cached high scores from the browser. A JavaScript API called `localStorage` allows values to be saved directly directly to the browser and retrieved using a key. Any values already saved to `localStorage` are brought out and compared with the latest new score. The top three scores are then placed back in browser cache as the definitive high scores. Also, this module provides a link back to the *Menu* module. After playing, the user can either start a new game or view the high scores.

Much like the *Menu* module, this component creates several items before it is shown. These include a sprite with the text *Game Over* as well an option to return to the menu. Additionally, a sprite to display the final score is generated in the `show` method. A screenshot of the *GameOver* screen is included in Figure 4.3.

Figure 4.3: The game over screen

4.2.8 High Scores

The final auxiliary component to analyze is the *HighScores* module. The main function of this module is to display the top local performances in the game. The structure of this component is very similar to that of *Menu*. When initialized with the `init` method, this module sets private variables and creates items to be rendered on the screen. However,

because the `init` method is only called once in the duration of the application, the high scores are not retrieved from `localStorage` during its execution. Instead, they are loaded during the `show` method to ensure that they are current.

The `show` method of the *HighScores* module is responsible for loading top scores and displaying them to the user. When called, the subroutine adds mouse click event listeners to the screen and then retrieves the high scores. Once the scores have been loaded, they are given associated sprites. A continuous loop is then started. Much like the *Menu* module, the continuous loop allows the *HighScores* module to scale sprites on window resize events and also accepts user input. Unlike the *Menu* module, however, this module only has one navigation item (a link back to the menu screen). An example of the *HighScores* screen can be seen in Figure 4.4.

Figure 4.4: The high scores screen

4.3 Levels

4.3.1 Object Oriented Programming

The architecture of the *Level* objects greatly differs from that of the game modules. Instead of following a modular design pattern, the Level objects employ an inheritance hierarchy. Inheritance is an ability of object-oriented programming that allows objects to

build on existing components [25]. In classical inheritance, each object template is called a class. Classes that provide common functionality are referred to as base classes, while classes that build on these are called derived classes [25]. Generally, a derived class is able to use the methods of its corresponding base class. Distinct classes that extend the same base class can be treated as the same type of object. JavaScript employs a special type of this concept called prototypical inheritance. While the central idea of inheritance, that objects build on other objects, is still in play, there are important distinctions between classical inheritance and inheritance using prototypes. The first thing to consider in prototypical inheritance is that all objects are built on prototypes. A prototype is another object that exposes its properties and methods for extension [26]. Because each prototype is also an object, it may have its own prototype as well, and so on [26]. Additionally, prototypical inheritance allows a new object to build on an instance of another object instead of its class, or template [27]. This allows each object of a certain type to adopt properties from other types without affecting its object template. In contrast, objects created using classical inheritance must share a template if they are of the same type. While prototypical inheritance differs from classical inheritance in some respects, both techniques share some of the same concepts.

Object-oriented programming allows for code reuse and abstraction. Although the implementation may differ, classical and prototypical inheritance allow for these advantages. Therefore, in this section, the terms class and object are used interchangeably. A fundamental concept of object-oriented programming is encapsulation. When the abilities of an object are exposed without its underlying intricacies, it is said to be encapsulated [28]. Similar to the modular design pattern, this allows an object to be modified internally without affecting its use [28]. Another essential idea of object-oriented software development is polymorphism, which is a by-product of inheritance. This concept allows a program to treat different objects in a similar manner if they extend the same base class [29]. This is because objects that inherit from the same base class intrinsically share its methods. In addition to having subroutines from the base class, each derived class can override these to better match its functionality. Virtual methods, base class functions which have no de-

fault implementation, are an extension of this idea. All of the aforementioned concepts are available in both classical and prototypical inheritance and were used extensively in this project.

4.3.2 LevelBase

The *LevelBase* module is the prototype of all other *Level* objects. It is designed to fully encapsulate the gameboard layout and rendering, as well as all game logic and scoring. An overview of the *LevelBase* API will be given in this section. It should be noted that its interface is extended by each *Level* object in order to differentiate itself from other levels.

On instantiation via its constructor, *LevelBase* sets constants, sprites and Turbulenz objects as private data members. These are accessible to any of its methods, as well as to the methods of any derived classes (other level objects). In addition to the constructor, *LevelBase* exposes an `init` method that is executed before the the object may be used. This function sets up the gameboard and starts its timer. The composition of the gameboard is a series of Turbulenz *RigidBody* objects contained by a private *World* data member. Each *RigidBody* represents an individual square that makes up one of the circles in the illusion. Geometric logic determines the position and rotation of each *RigidBody*, which is also assigned a black or white sprite, thus recreating the illusion.

Two other important methods exposed by the *LevelBase* object are called `update` and `render`. `Update` is responsible for decrementing the timer, updating physics objects, applying effects and determining if the timer has expired. To add any effects on a recurring basis, a virtual method named `\_updateBlink` is called from `update`. Some of the levels implement this virtual method, however its implementation is not required. Similar to the `update` function, `render` is responsible for drawing all physics objects as well as informative sprites, such as level number, score and time remaining. Because the gameboard is contained in Turbulenz *World* data member, the `render` method simply iterates through its *RigidBody* objects and draws each individually. In addition to these methods, the *LevelBase* object has an input handling function called `onTap`. Clicks (and taps) on the screen are central to gameplay. Therefore, whenever a click is registered, the private *World* object is queried to

determine if the click struck any of its *RigidBody* objects. If so, additional logic determines the validity of the input in conjunction with game rules. Finally, the *LevelBase* module has several "getter" and "setter" methods. These functions expose the level state, score and time remaining. The *Game* module relies on these methods in order to control the levels. Because all of the levels inherit from the *LevelBase* object, these can act polymorphically. This allows the *Game* module to execute the same functions on each *Level* object, regardless of its specific type. Overall, the *LevelBase* component provides a simple and extensible interface for all *Level* objects.

4.3.3 Derived Levels

Each derived *Level* follows the same pattern of extending a base object. These objects are denoted by the name *Level* and then a number, such as *Level1* or *Level8*. The first five levels extend the *LevelBase* object directly, while the next ten build on these. Each *Level* extends the `init` method in order to create a list of circles that need to be drawn and then calls the base `init` function to initialize any other level variables. Because each level is associated with a particular challenge, they may override how the level is updated or how input is handled. After every five levels, the challenge is repeated, only this time with an additional circle to enhance the illusion. This makes the implementation of the later levels especially simple.

Altogether, this project employed a number of common software design principles. As a whole, the software consists of distinct modules that can work independently of each other. The central *Application* component is a container for all of the auxiliary modules. In addition to the modular design pattern, common object-oriented concepts are applied in the structure of the levels. These include inheritance, polymorphism and method overriding. The overall design of this project improves maintainability and encourages extensibility.

CHAPTER 5

Gameplay Evolution

5.1 Overview

The finalized gameplay was a created through a process of concept and experimentation. While the idea of using the Fraser spiral was consistent throughout development, the core gameplay varied considerably throughout the course of this project. As the game framework was created, new adaptations and improvements were hypothesized and tested. All gameplay concepts stemmed from an understanding of how the illusion is perceived. Additionally, each version was designed especially for mobile devices because of the massive market for smartphone and tablet games. Several gameplay ideas never made it to a demo due to their difficult implementation and high risk of failure. Another major issue in the development of this project was due to the lack of metrics to quantify the prevalence of the illusion in gameplay. That is, while some gameplay methods may be more challenging than others, there is not a solid mechanism to determine how much the illusion causes the increased difficulty. Accordingly, game development relied on user feedback regarding the effectiveness of the illusion. In summary, the development of this project was an attempt to find a balance between an optical illusion and simple, enjoyable gameplay.

5.2 Original Concept

The original design for this game differed greatly from the finished product. Initially, the idea was to have a dynamic, user-controlled object attempting to navigate in between any of the two circles that comprise the illusion. Much like other computer games, this user object was designed to collect coins and other items without touching the boundary of either the inner or outer circles. Striking any of the circles would cause the user to lose life points until all had expired and the game was over. Control of the main object was

supposed to be a function of device orientation, also known as tilt. A screenshot of this game on a mobile device can be seen in Figure 5.1. It was supposed that the entanglement effect of the circles caused by the Fraser spiral would confuse the users as they attempt to guide the object around the level. Unfortunately, the illusion effect was essentially nullified by the gameplay. It was quickly discovered that as the user tracks his/her object throughout the map, the interweaving effect disappears around it. In other words, the spiral illusion is most effective when the image is being view as a whole, instead of just one area. Although disappointing, the failure of this initial idea paved the way for the remainder of this project. Instead of simply implementing a preconceived game concept, this project would require experimentation in order to find suitable gameplay.

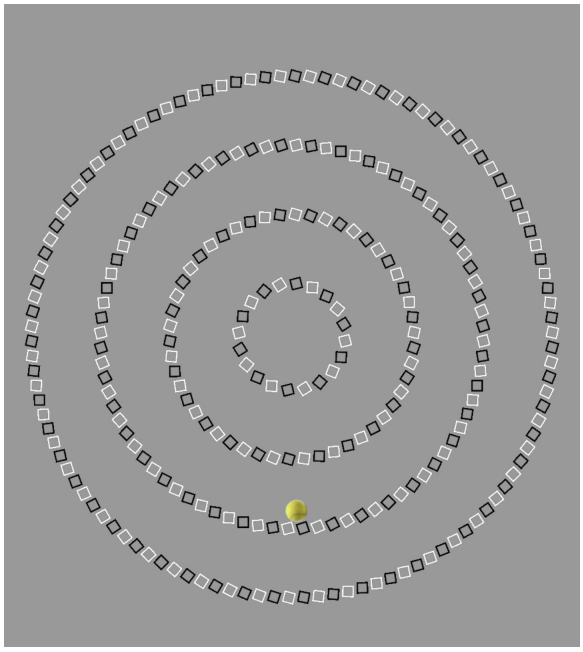

Figure 5.1: The first iteration of the game.

5.3 Alternative Gameplay Concepts

After the initial idea was rejected, several new designs were generated in an attempt to find gameplay that successfully utilized the illusion. One early approach was to allow one of the circles involved in the illusion to be adjusted according to device orientation.

The user would tilt the device in an attempt get the circle to be concentric with all others. The idea behind this concept was that the user would take in more of the image (and therefore the illusion) while trying to position the stray circle. The main issue with this idea was geometry. Because the circles are concentric, positioning one part of the user-controlled circle would correctly arrange the entire circle. As a result, this gameplay model was rejected because it could not consistently force the user to perceive the illusion.

Another, more successful version of gameplay that was implemented and tested involved hiding a rogue white square at a random location between any two of the circles. The user simply had to locate and tap on this square in the allotted time. Once the correct position was determined, the square was respawned in a different region. Unfortunately, the change in this square's position could be detected if it was relocated immediately. This situation was remedied by blurring the entire image before relocating the square, which was sufficient for perception to reset. The strength of this version is that it required the user to view the entire image in searching for the rogue square, and thus perceive the illusion. To improve the difficulty of this version, additional effects were added to the illusion in order to distract and disorient the user. Much like the overall gameplay, these effects were the result of experimentation. One attempt to confuse the user was filling in a few random squares in each circle about every second. It was thought that the added motion would disturb the scanning process and this was met with moderate success. A screenshot of this version of the game with the filled in circles in included in Figure 5.2. Another enhancement was rotating adjacent circles in opposite directions. This effect was highly successful at disorienting the user, which could have caused him/her to taken more time to find the square. One variation of this enhancement was causing the errant square to rotate along with one of the circles. While the square was no longer stationary, it sometimes would stand out because it was traveling opposite another circle. Another variation of this effect was reversing the direction of rotation after each click. Finally, a combination of rotation and blinking squares was also attempted. Overall, this version of the game was the most successful up to this point, and it opened the door to the final gameplay concept.

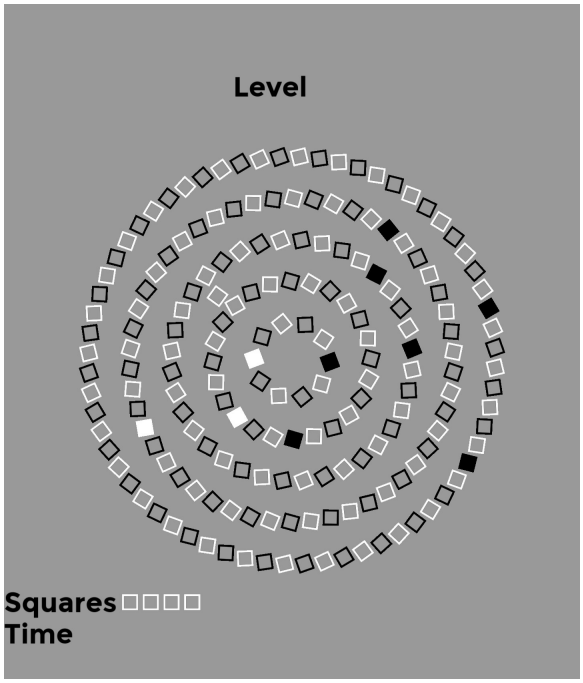

Figure 5.2: Another version of the game with an added effect

5.4 Finalized Gameplay

The game that is presented with this project was the result of experimentation and modification of other concepts. The idea for the final version was a result of the aforementioned observation that the entire image must be viewed as a whole in order to perceive the illusion. Although the previous gameplay iteration forced the user to scan the whole figure, the illusion did not cause much of a challenge. As a result, a modification was made to the geometry of the image by adding a positional offset to one of the circles. Because the illusion feigns a spiral, it already appears that the circles are not concentric. Therefore, an additional offset applied to one circle may go unnoticed. Once it was determined that this alteration was legitimate, the gameplay was borrowed from the previous version. However, instead of tapping one square, any part of the offset circle could be selected. The effects from the previous version were also maintained in order to provide distraction and avoid monotony. The game was extended by adding an additional circle to the illusion every five levels and then repeating the effects in order. Only fifteen levels were considered for the

sake of avoiding redundancy. Finally, in order to prevent the user from detecting changes in the game board, a solid square was added to appear after each click or tap. This was intended to be a more effective form of the blurriness applied after each tap in the earlier gameplay version. Finally, a time-based scoring method was implemented. With the core gameplay in place, the project was essentially completed. However, to be thorough, a web application was developed for testing and will be explained in the next section.

CHAPTER 6

Testing and Results

6.1 Overview and Framework

This project was primarily concerned with building a game using the Fraser spiral illusion. To validate, a test was developed to gather statistics concerning the effectiveness of the illusion on user performance. In order to conduct this test, a version of the game was developed that removed the illusion while maintaining the core gameplay. In this form, all of the square outlines that composed each circle were replaced with solid black squares and they were rotated inward. An image of one of the levels of this game version is shown in Figure 6.1. With both versions complete, the next matter was to gather data from users. A number of modifications were made to the underlying game in order to collect statistics.

The *LevelBase* object was modified to record the number of incorrect guesses in each level as well as the time required by a user to find the errant circle. Also, the countdown timer was prevented from canceling the game, meaning that a user was not restricted by time. Additionally, an HTTP request was added to the *Game* module to send statistics to the hosting server after each level. In addition to recording user performance, metadata concerning each level was sent to the server. This included the number of circles in the level as well as any effects that were added. The completed testing framework was uploaded to a public hosting site, accessible to anyone with an Internet connection. Finally, in order to compare user performance, a unique identifier was loaded into the browser as the game is loaded. This identifier remains in the local storage of the browser, staying the same regardless of how many times the player visits the site. The uploaded statistics were written to a CSV file for further processing.

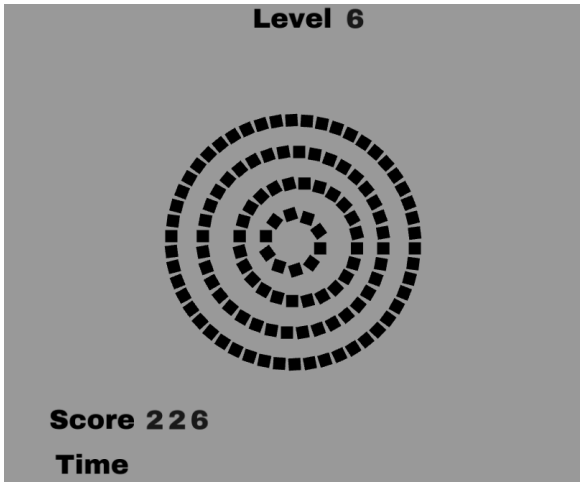

Figure 6.1: The test version of the game, lacking the illusion

With the test framework fully functioning, volunteers were required. Both versions of the game were presented to students and faculty in a number of computer science courses who were invited to participate. In addition to these students, the game was shared on social media in order to bump up the number of testers. All statistics presented in this report were gathered over a one-week period.

6.2 Results

The test results in this section establish the effectiveness of the illusion on user performance. Data gathered comparing player efficiency between the games with the illusion and without it is examined on two metrics. In both cases, the game version that incorporated the illusion was associated with poorer performance, meaning that the illusion made the game more difficult. To compare the two game types, the dataset was reduced to include only users who played each version all of the way through to the end. In addition to this restriction, performance is only compared on levels without any effects. The justification for this condition is that the various effects differed between the two forms of the game.

The first comparison was made on the average time required to complete each level. It was expected that the presence of the illusion would increase the amount of time required to pass a level. Without the illusion, the mean time to complete a level was 20.18 seconds. In

contrast, the same users needed 40.32 seconds to pass a level where the illusion was present. A bar chart highlighting these differences is shown in Figure 6.2. Although not enough data was generated to label these results as statistically significant, a striking difference is apparent. The gameplay was identical between both versions, but each level took about twice of long to complete when the illusion was present.

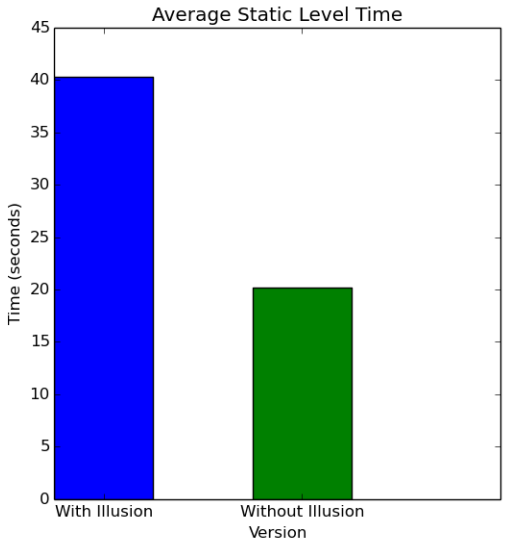

Figure 6.2: The average time required to pass each static version of the game

The other metric used to compare the two game versions was the average number of incorrect guesses on each level. An incorrect guess is defined as any time the player clicks on an incorrect circle in the game. It is expected that because the illusion confuses the user, there would be more incorrect guesses when it is present. The data agree with this conjecture. On average, each player made only about two incorrect guesses on levels without the illusion. In comparison, an average of seven incorrect guesses were made on levels where the illusion was present. This statistic is especially interesting because each level only requires the user to select five correct circles. To summarize, on average the illusion causes each player to make more incorrect guesses than correct ones. In conclusion, although the data was insufficient to find statistically significant correlations, it illustrates that the presence of the illusion in the game board increases difficulty.

With the success of the illusion demonstrated, the next part is to analyze the impact of the various effects on user performance. To review, there were four different enhancements added to the core gameplay, in addition to a version without any effects. These are referred to in the data as blink, rotate, blink-rotate, reverse-rotate and static. Blinking enhancements involved randomly filling in the outlines of the squares that composed each circle. Rotational effects comprised of revolving each circle in an opposite direction. In particular, the reverse-rotate effect would change the rotational direction of each circle after every mouse click. When no enhancements were used in a level, it was referred to as static. The only data considered for this analysis came from users who completed the entire illusion version of the game. A comparison of the effectiveness of each enhancement on average time to complete each level can be seen in Figure 6.3. Interestingly, the levels with no effects (static) were among the most difficult to pass, while levels with any rotational effects were actually easier. Although revolving the circles was disorienting, it perhaps made the errant circle stand out more and, therefore, easier to identify. Additionally, levels featuring the blinking effect took slightly longer than their non-blinking counterparts. In fact, the levels that took the longest to complete on average were static but with the blinking enhancement. This shows that although the illusion was potent at making the game more difficult, adding effects can alter user performance.

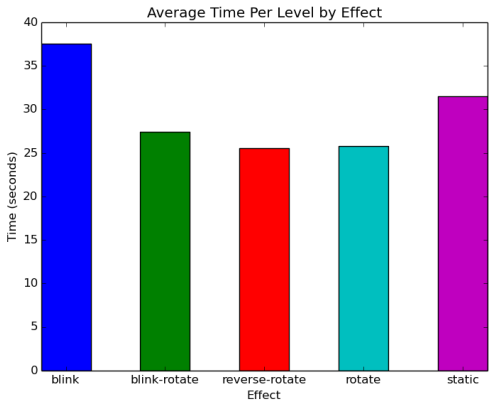

Figure 6.3: A comparison of each effect in relation to average time to pass a level

CHAPTER 7

Future Work

This project involved building a new classification of computer game using standard tools and programming practices. Although this version of the game is complete, there are several related areas to explore for further research.

One application of this project is to convert it to a native mobile application. While a number of tools exist for facilitating this transition, it may be useful to explore alternative technologies in order to improve performance. For example, more robust game engines, such as Unity, may be able to create a stronger experience. The game will certainly have to be tested on a number of mobile devices before it can be released as well.

Additionally, there are other gameplay concepts incorporating the illusion that can be experimented with. In this project, the final form of gameplay was the result of hypothesis and testing. Such a process promotes continuation. It may even be worthwhile to combine several gameplay methods together.

Finally, other, more difficult, forms of the Fraser spiral illusion may be integrated to later levels. As seen in Figure 1.1, the version used in this project is hardly exclusive to the illusion. This goes along with investigation of additional level effects. More advanced game building tools, such as shaders, may increase the potency of the simple effects introduced in this game as well.

CHAPTER 8

Conclusion

Building a complete game using an optical illusion is entering untested waters. Development of the game relied on classic software engineering principles and techniques. Additionally, tools and modules provided by the Turbulenz game engine greatly aided in this process. Although the initial design for the project was ultimately rejected, an effective version was eventually discovered. This issue illustrates the inherent difficulty of such an undertaking. With a completed version of the game, data show the effectiveness of the illusion on user performance. In essence, the completion of this project is a proof of concept that optical illusions can be incorporated into a computer game. That being said, there exists an expansive opportunity for continuing work.

REFERENCES

[1] [Online]. Available: http://www.merriam-webster.com/dictionary/optical%20illusion

[2] M. Bach and C. Poloschek, "Optical Illusions," Advances in Clinical Neuroscience and Rehabilitation, 2006. [Online]. Available: http://www.acnr.co.uk/pdfs/volume6issue2/v6i2visual.pdf

[3] M. Barile, "Frasers spiral," in WolframMathWorld, Wolfram Research, 2002. [Online]. Available: http://mathworld.wolfram.com/FrasersSpiral.html.

[4] "Fraser spiral illusion," Wikimedia Foundation, 2015. [Online]. Available: https://en.wikipedia.org/wiki/Fraser_spiral_illusion.

[5] I. Salfa, "Irregular rings optical illusion," wordlessTech, 2014. [Online]. Available: http://wordlesstech.com/irregular-rings-optical-illusion/.

[6] "15 optical illusions that will blow your mind," Designer Daily: graphic and web design blog, 2015. [Online]. Available: http://www.designer-daily.com/15-optical-illusions-that-will-blow-your-mind-51985.

[7] "Turbulenz Games Platform: Technology Introduction," Turbulenz Labs, 2011. [Online]. Available: http://biz.turbulenz.com/static/presentations/Turbulenz-Whitepaper-2011.pdf.

[8] "JavaScript," in Mozilla Developer Network, 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[9] "JavaScript Data," in w3schools.com. [Online]. Available: http://www.w3schools.com/js/js_datatypes.asp.

[10] "Chrome V8," in Google developers. [Online]. Available: https://developers.google.com/v8/.

[11] T. Laurens, "How the V8 engine works?," 2013. [Online]. Available: http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/.

[12] "Git - git basics," 2nd ed. [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics.

[13] 2016, "What is unreal engine 4," in Unreal Engine, 2004. [Online]. Available: https://www.unrealengine.com/what-is-unreal-engine-4.

[14] U. Technologies, "Unity - game engine, tools and multiplatform," 2016. [Online]. Available: https://unity3d.com/unity.

[15] "Developers," Turbulenz Limited, 2012. [Online]. Available: http://biz.turbulenz.com/developers.

[16] "What is the status of Turbulenz for mobile?," in Turbulenz News, Turbulenz Limited, 2013. [Online]. Available: http://news.turbulenz.com/post/52212393956/what-is-the-status-of-turbulenz-for-mobile.

[17] "9.1. Introduction," in Turbulenz 0.28.0 documentation, 2014. [Online]. Available: http://docs.turbulenz.com/local/user_guide.html.

[18] "25.3. Maketzjs," in Turbulenz 0.28.0 documentation, 2010. [Online]. Available: http://docs.turbulenz.com/tools/game_tools.html.

[19] "21.39. The RequestHandler object," in Turbulenz 0.28.0 documentation, 2010. [Online]. Available: http://docs.turbulenz.com/jslibrary_api/requesthandler_api.html.

[20] "18.3. The RigidBody object," in Turbulenz 0.28.0 documentation, 2010. [Online]. Available: http://docs.turbulenz.com/jslibrary_api/physics2d_body_api.html.

[21] "18.12. The world object," in Turbulenz 0.28.0 documentation, 2010. [Online]. Available: http://docs.turbulenz.com/jslibrary_api/physics2d_world_api.html.

[22] "What is modular programming?" Techopedia.com, 2016. [Online]. Available: https://www.techopedia.com/definition/25972/modular-programming.

[23] "17.2. The TurbulenzEngine object," in Turbulenz 0.28.0 documentation, 2010. [Online]. Available: http://docs.turbulenz.com/jslibrary_api/engine_api.html.

[24] "Gamedev glossary: What is the 'Game Loop'? Game Development Envato Tuts+, 2012. [Online]. Available: http://gamedevelopment.tutsplus.com/articles/gamedev-glossary-what-is-the-game-loop-gamedev-2469.

[25] M. Yaiser, "Object-oriented programming concepts: Inheritance," Adobe, 2012. [Online]. Available: http://www.adobe.com/devnet/actionscript/learning/oop-concepts/inheritance.html.

[26] "Inheritance and the prototype chain," Mozilla Developer Network, 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[27] I. Kantor, "Prototypal inheritance," 2011. [Online]. Available: http://javascript.info/tutorial/inheritance.

[28] M. Yaiser, "OOP concepts: Encapsulation," Adobe, 2011. [Online]. Available: http://www.adobe.com/devnet/actionscript/learning/oop-concepts/encapsulation.html.

[29] V. Beal, "What is Polymorphism?," in Webopedia, 2016. [Online]. Available: http://www.webopedia.com/TERM/P/polymorphism.html.