TEC | Tecnológico de Costa Rica

May 1, 2024

Our version of SpotIT!

Computing Engineering

Programming Languages

Authors:
Danilo Duque,
Desireé Avilés,
Emmanuel Rojas,
Joshua Jiménez

Professor:
Eddy Ramírez Jiménez

1 Executive Summary

The SpotIt Project is dedicated to crafting an immersive gaming experience through the development of a custom implementation of the SpotIt game. This executive summary provides an overview of the approach taken to create an engaging gaming environment, highlighting the unique features and objectives of our implementation of the SpotIt Game.

The documentation for the SpotIt Project encompasses a comprehensive showcase of all the tools utilized to achieve the implementation of the game. This executive summary offers insight into the methodology adopted to curate an immersive gaming atmosphere, emphasizing the distinctive attributes and goals inherent in the rendition of the SpotIt Game.

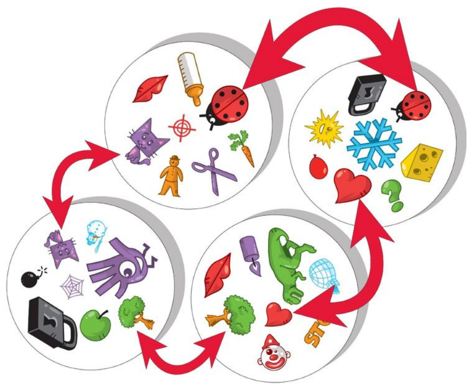
Various tools were leveraged throughout the development process, including computational geometry algorithms for the creation of decks, probabilistic algorithms to enhance game dynamics, utilization of cloud services for scalable infrastructure, integration with databases for player profiles and game statistics, web page development for online accessibility, and object-oriented programming for a modular and maintainable codebase, among others. These tools collectively contribute to the seamless execution of the SpotIt Game vision, enriching the gaming experience for users.

Moreover, detailed explanations of each factor considered in the project's implementation will be presented in the following sections. These descriptions will provide a more in-depth look at the particular methods, tools, and tactics utilized to guarantee the effective creation of the SpotIt Game. Keep an eye out for a comprehensive examination of our methodology and its impact on shaping a gaming experience.

2 Introduction

Spot It, a widely acclaimed card game renowned for its engaging gameplay and vibrant design, was created by Dobble and distributed by Asmodee. This beloved game has garnered global acclaim due to its simplicity and adaptability, making it a favoured choice for various social occasions, family gatherings, and educational environments.

Spot It offers players a variety of modes, each with its own distinctive gameplay elements. Yet, the fundamental goal remains unchanged: swiftly identifying the common image between two cards. What sets Spot It apart is its unique feature: between any pair of cards, there is always one and only one shared image. This becomes particularly challenging as the images on the cards are rotated and scaled, adding an extra layer of complexity to the task of identifying the intersection of two images.



The mathematical foundation of this seemingly simple game is built upon its intricately designed deck. Each card is meticulously crafted to feature a unique arrangement of symbols, ensuring that only one matching symbol is shared between any two cards. Moreover, efforts are made to minimize the number of images and cards required while preserving the integrity of the original gameplay experience. This sophisticated deck design draws parallels with geometry, specifically within the realm of finite projective planes.

Following the exploration of Spot It's unique gameplay mechanics and its mathematical underpinnings inspired by finite projective planes, it becomes evident that the game transcends mere entertainment and delves into the realm of intellectual challenge. This introduction sets the stage for a deeper exploration of the SpotIt Project, where the process of translating this beloved card game into a digital format while retaining its essence and complexity will be discussed.

3 Theoretical Framework

In this section, a comprehensive overview is presented regarding the foundational theories and concepts that underpin the understanding of the tools utilized for addressing the problem at hand. Through an exploration of these theoretical perspectives, the aim is to establish a framework for analysing and interpreting the problem within a broader scholarly context.

3.1 Finite Projective Planes

In the game SpotIt, the process of constructing decks parallels the creation of a finite projective plane. However, one might wonder: what exactly constitutes a finite projective plane?

In mathematics, a projective plane is a geometric structure that extends the concept of a plane. While in the conventional Euclidean plane two lines typically intersect at a single point, exceptions arise with parallel lines that never meet. Introducing the notion of "points at infinity," a projective plane enables parallel lines to intersect at these additional points. Consequently, any two distinct lines in a projective plane intersect at precisely one point.

According to the preceding definition, a projective plane $\mathbb{P}$ is defined as a triple (P, L, I), where P denotes the set of points p of $\mathbb{P}$, L represents the set of lines l of $\mathbb{P}$, and $I \subseteq P \times L$ is the incidence relation satisfying the following axioms:

1. For any two distinct points $p_1, p_2 \in P$, there exists a unique line $l \in L$ such that $(p_1, l)(p_2, l) \in I$.
2. For any two distinct lines $l_1, l_2 \in L$, there exists a unique point $p \in P$ such that $(p, l_1)(p, l_2) \in I$.
3. There exist four distinct points in P such that no line in L is incident with more than two of them.

$\mathbb{P}$ is said to be an infinite if $|P| = \infty$ and finite otherwise.

To illustrate, consider standing between two train rails: as you look into the distance, the rails appear to converge, eventually seeming to intersect at a vanishing point on the horizon. This phenomenon reflects the principles of a projective plane.

The degree g of a finite projective plane refers to the count of points incident to a line $l \in L$. This parameter is essential in characterizing the geometric properties and structure of the plane. In the context of a projective plane $\mathbb{P}$ of order q, where q is a power of a prime number, the degree g is precisely $q+1$. This signifies that each line in the plane contains $q+1$ points, a fundamental property that guides the arrangement of elements within the plane.

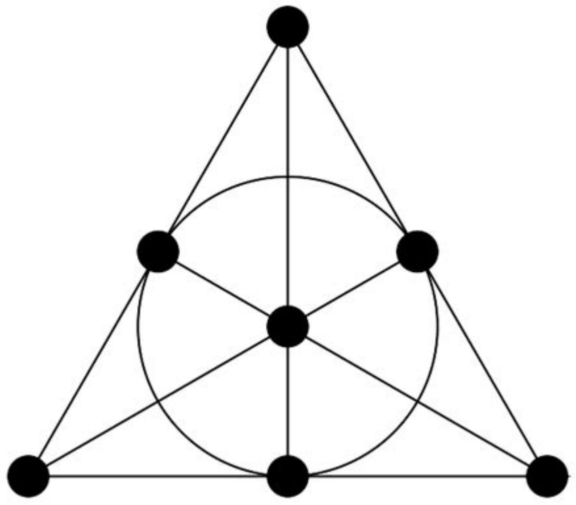


Figure 1: Fano plane, is a finite projective plane of degree 3

Now, consider a finite projective plane $\mathbb{P}$ of degree g. Here, the concept of the degree extends beyond mere enumeration of points and lines; it encapsulates the intricate relationships between these elements. The minimum number of distinct points and lines within such a plane is determined by $g^2 - g + 1$.

Within the context of SpotIt, each card assumes the role of a line, while each image represents a point. This conceptualization effectively constitutes a deck, as ensuring adherence to the axioms of the incidence relation of a projective plane is equivalent to fulfilling the SpotIt deck requirements.

3.2 Cyclic Difference Sets

A cyclic difference set D modulo m is a set of positive integers $\{0, n_2, n_3, \ldots, n_k\}$ less than m, where $|D| = k$, that satisfies the property that all the differences between any pair of elements in D modulo m are distinct. In other words, for every pair of elements n_i and n_j in D, with $i \neq j$, the difference $n_i - n_j$ is distinct from any other difference $n_p - n_q$, where p and q are distinct indices from i and j.

An intriguing property of cyclic difference sets is their ability to readily generate finite projective planes. This is achieved by utilizing a cyclic difference set D of size k, where k also denotes the degree of the finite projective plane $\mathbb{P}$.

The connection between cyclic difference sets and finite projective planes lies in the correspondence between points and lines in these geometric systems. In a finite projective plane of degree k, each point is associated with a line and vice versa. The existence of a cyclic difference set of size k provides a convenient way to define this correspondence between points and lines.

When employing a cyclic difference set D of size k, the process of assigning points and lines is a fundamental step in establishing the geometric structure of the resulting finite projective plane. Let's delve deeper into this process:

Firstly, consider D as a set containing k distinct elements. Each element of D serves as a reference point within the construction process. These points are then utilized to define the points of the finite projective plane.

For each element d_i in D, a corresponding point P_i is established in the finite projective plane. These points are distinct and represent the vertices of the geometric space.

Now, to define the lines of the finite projective plane, we consider the differences between the elements of D. These differences are calculated modulo k, ensuring that they remain within the range of the set D. Let's denote the set of these differences as Δ.

For each difference δ in Δ, a line L_δ is constructed in the finite projective plane. The construction of these lines is crucial in establishing the geometric properties of the plane.

Each line L_δ consists of points P_i such that the difference between any two points P_i and P_j is equal to δ modulo k. This ensures that every line passes through precisely k points, maintaining the necessary properties of the finite projective plane.

Moreover, the cyclic nature of the difference set D guarantees that all differences between its elements are represented within Δ, ensuring the completeness of the set of lines in the finite projective plane.

Through systematic association of points with elements of D and lines with differences between these elements, a well-defined geometric structure is established in accordance with the principles of finite projective geometry.

3.3 Monte Carlo Probabilistic Algorithm

A probabilistic algorithm is a computational procedure characterized by a sequence of steps, wherein at least one step incorporates randomness into its output, influencing the subsequent steps in the algorithm's execution. This injection of randomness distinguishes probabilistic algorithms from their deterministic counterparts, allowing them to navigate through complex problem spaces with a degree of adaptability not present in purely deterministic approaches. By introducing random elements, probabilistic algorithms can explore different paths, potentially leading to more efficient or effective solutions. This stochastic nature enables probabilistic algorithms to tackle problems that may be intractable using deterministic methods alone, offering valuable tools for addressing a wide range of computational challenges in diverse fields such as optimization, cryptography, and machine learning.

Now, a Monte Carlo algorithm represents a specific category within the realm of probabilistic algorithms. In essence, it comprises a sequence of steps where randomness plays a pivotal role in the approximation of an optimal result, albeit without guaranteeing its attainment.

3.4 Amazon Web Services

AWS stands for Amazon Web Services. It is a cloud computing platform provided by Amazon.com that offers a wide range of infrastructure services such as storage, computing power, databases, networking, and analytics, among others. AWS enables businesses and organizations to utilize computing resources over the Internet, eliminating the need to invest in costly physical hardware and infrastructure maintenance. It is one of the most popular and widely used cloud computing platforms globally.

Several services were employed in the implementation of the SpotIT project, including:

3.4.1 RDS

Amazon RDS (Relational Database Service) is a cloud-based service provided by Amazon Web Services (AWS) that simplifies the setup, operation, and scaling of relational databases. With Amazon RDS, users can easily deploy, manage, and scale popular database engines such as MySQL, PostgreSQL, Oracle, SQL Server, and MariaDB without needing to manage the underlying infrastructure. This service automates routine database tasks such as hardware provisioning, database setup, patching, backups, and scaling, allowing users to focus more on application development and less on database management.

3.4.2 S3

Amazon S3 (Simple Storage Service) is a widely-used cloud storage service provided by Amazon Web Services (AWS). It allows users to store and retrieve data of virtually any size at any time, securely and reliably. Amazon S3 offers scalable storage infrastructure with high availability and durability, making it suitable for a wide range of use cases, from simple backup and archival to serving static website content and hosting entire applications.

With Amazon S3, users can upload files, objects, or data sets, which are stored as "buckets" in the cloud. These buckets act as containers for the data and can be organized and managed as needed. Amazon S3 provides features such as versioning, encryption, access control, and lifecycle management to help users securely store, manage, and protect their data.

Amazon S3 is designed to be highly scalable and cost-effective, with users only paying for the storage they use and any additional data transfer or operations performed. It offers a simple and intuitive interface, as well as robust APIs and SDKs for seamless integration with other AWS services and third-party applications. Overall, Amazon S3 is a versatile and reliable solution for storing and managing data in the cloud.

3.4.3 IAM

IAM (Identity and Access Management) is an AWS service for securely managing access to resources. It controls user identities, permissions, and policies, ensuring only authorized users and systems can access AWS resources. IAM offers user, group, and role management, along with fine-grained access control through policies. It also supports features like multifactor authentication and auditing for maintaining security and compliance in the AWS environment.

3.4.4 EC2

Amazon EC2 (Elastic Compute Cloud) is a web service provided by Amazon Web Services (AWS) that allows users to rent virtual servers, known as instances, on which they can run their own applications. EC2 provides scalable computing capacity in the cloud, allowing users to quickly scale up or down based on their computing needs.

With Amazon EC2, users can launch instances of various types, sizes, and configurations, depending on their specific requirements. These instances can be configured with different operating systems, software packages, and security settings, providing users with flexibility and control over their computing environment.

Amazon EC2 offers a pay-as-you-go pricing model, where users only pay for the compute capacity they use, making it cost-effective for a wide range of workloads, from small development projects to large-scale enterprise applications.

EC2 instances can be used for a variety of purposes, including web hosting, application development and testing, data processing, machine learning, and high-performance computing, among others. It provides a reliable and secure infrastructure for running virtually any type of workload in the cloud.

3.5 MySQL

MySQL is an open-source relational database management system (RDBMS) widely used for storing and managing structured data. Developed by Oracle, it offers features like transactions, indexing, and SQL querying. Known for scalability and ACID compliance, it supports various storage engines. MySQL is popular in web applications and has broad platform support.

3.6 JavaScript

JavaScript is a high-level programming language commonly used for creating interactive and dynamic web content. It is a versatile language that runs on the client-side within web browsers, allowing developers to manipulate webpage elements, respond to user actions, and dynamically update content without requiring page reloads. JavaScript is also used on the server-side through platforms like Node.js, enabling developers to build scalable and performant web applications. With its widespread adoption and extensive ecosystem of libraries and frameworks, JavaScript has become a cornerstone of modern web development.

3.7 Java

Java is a high-level programming language renowned for its portability, reliability, and security. It's extensively used for building enterprise-level applications, mobile apps (Android), and web services. Java's "write once, run anywhere" principle allows code to be executed on any device with a Java Virtual Machine (JVM). This platform independence, along with its robust standard library and strong community support, has made Java one of the most popular programming languages worldwide.

3.8 C

C is a powerful and efficient procedural programming language known for its versatility and performance. Developed in the early 1970s, it remains widely used today in various applications, including system programming, embedded systems, and high-performance computing. C's simplicity, portability, and close-to-hardware access make it ideal for developing operating systems, device drivers, and low-level software components. Despite its minimalistic syntax, C offers robust features such as pointers, memory management, and low-level access to computer hardware, making it a preferred choice for developers seeking control and performance in their applications.

3.9 Separating Axis Theorem

The Separating Axis Theorem (SAT) is a method used to determine whether two convex objects intersect in Euclidean space. It works by examining a series of axes (or directions) in space to see if any of these axes can separate the two convex objects. If an axis is found that can separate the convex objects, then there is no intersection between them. However, if no separating axis is found, it concludes that there is an intersection between them. This theorem is particularly useful in collision detection in computer graphics and physics simulations due to its efficiency and ease of implementation, and it can be applied to a wide range of shapes and convex objects in Euclidean space.

3.10 Object Oriented Programming

Object-oriented programming (OOP) is a cornerstone in modern software development, offering a robust framework for building scalable and maintainable applications. At its heart lies the concept of objects, where data and behaviour are encapsulated into cohesive units. This paradigm promotes a modular approach to software design, allowing developers to break down complex systems into manageable components.

1. Encapsulation ensures objects hide internal details, enhancing security and data integrity.

2. Inheritance promotes code reuse and hierarchical relationships among classes.

3. Polymorphism enables treating different object types uniformly, simplifying maintenance and promoting extensibility.

4. Abstraction simplifies complex systems into generalized models, aiding high-level design without getting bogged down by implementation details.

4 Solution Description

This section is divided into three parts, each offering a comprehensive explanation of the resolution process for individual sub-problems. The integration of these three sub-solutions culminates in the successful implementation of our version of SpotIT.

Generation of Decks

The creation of in-game decks presented a significant challenge, comprising two distinct sub-problems. Firstly, it involved generating Finite Projective Planes to represent the deck. Secondly, it required the arrangement of images, including rotation and scaling, for each card within the deck.

Finite Projective Plane Generation

As previously discussed in the theoretical framework, a SpotIT deck can be represented isomorphically as a finite projective plane, with the degree equal to the number of images per card in the deck.

Hence, the process of creating a deck can be streamlined to generating a finite projective plane that adequately represents it. While a swift algorithm exists for constructing finite projective planes of degree p where p is a prime number, this situation requires exploring degrees that are powers of a prime number less than 10.

By imposing this requirement, the algorithmic complexity increases significantly, leading to an NP-complete problem. This solution can be reduced to graph coloring since each card requires a unique combination of images, resembling vertices in a graph that must be colored without adjacent vertices sharing the same color.

In this context, cyclic difference sets play a pivotal role. This is due to the fact that the creation of a finite projective plane of degree d can be expedited by the availability of a cyclic difference set of size d. Therefore, by pre-computing and storing these sets in memory, the generation of decks can be significantly accelerated.

Let C be a finite set representing the entire deck of cards, and let D be a cyclic difference set of size k contained within C ($D \subset C$). This means for every element $d \in D$, there exists a corresponding element $c_d \in C$ such that $d = c_d$. We also define an equivalent set D'.

The algorithm proceeds by performing $|C|$ rotations, where each rotation introduces a new "image" in a card. During the i^{th} rotation ($1 \leq i \leq |C|$), we increment each element in D' by 1 (modulo $|C|$). This ensures elements stay within the valid range for cards. Each rotation results in a unique intersection between D and the rotated set D', containing exactly one element. This element represents the card that will not have an image introduced in the i^{th} rotation.

By performing $|C|$ rotations, each card in C will have k corresponding images. Furthermore, the resulting deck structure becomes isomorphic to a finite projective plane of degree $|D|$.

Here is an implementation of the algorithm in C++:

```cpp
void createFPP(vector<int> &C, vector<int> &D){

    vector<int> D1(D);
    for(int i = 0; i<C.size(); ++i){
        for(int j = 0; j<D.size(); ++j)
            for(int k = 0; k<D.size(); ++k)
                if(D1[k] != D[k])
                    C[D1[j]] |= 1<<i;

        for(int j = 0; j<D.size(); ++j)
            D1[j] = (D1[j] + 1)%C.size();

    }

}
```

Distribution of the images on the card

In the original game of SpotIT, each card features a configuration comprising n images. These images are not only rotated but also scaled by seemingly random factors, contributing to the unique and dynamic appearance of each card. This rotation and scaling process adds an element of unpredictability and challenge to the game, as players must quickly identify matching images despite variations in orientation and size across different cards.

A requirement for positioning the cards is to maximize the area occupied by images on each card. This ensures that the game remains challenging without becoming overly complicated, while also maintaining the cards' aesthetic appeal.

The solution algorithm commences with the identification of a recurring pattern in SpotIT cards. It is observed that the majority of cards adhere to one of two configurations: either the images are arranged in a manner reminiscent of the hours on a clock, with subsequent scaling and rotation, or a single image is centrally positioned, surrounded by others in a manner akin to the previously mentioned pattern.

Adhering to this guideline is fundamental for cultivating a thorough comprehension of the algorithm. Let's now delve into the intricacies of scaling and rotating each image.

When addressing scaling, let's consider an image I depicted as a quadrilateral with area A, necessitating enlargement by a factor of f. To achieve this, each coordinate undergoes multiplication by the square root of the scaling factor, denoted as $\sqrt{f}$. Consequently, scaling a dimension by $\sqrt{f}$ results in the area being multiplied by f. This function was implemented to maximize the expansion of a particular image while ensuring avoidance of intersections with other images on the card. To expedite this process, a binary search algorithm was introduced, enabling logarithmic time searches to determine the largest factor by which the image could be scaled without encountering collisions with other images.

Next, let's explore the process of rotation, which involves adjusting each image's orientation to align with the desired configuration. This entails multiplying each point belonging to an image I by the 2D rotation matrix R:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Here, (x, y) represents the coordinates of a point in the original image, θ denotes the rotation angle, and (x', y') corresponds to the coordinates of the rotated point.

Two types of rotation were utilized: rotation from the center of the image and rotation from one corner of the image. This was implemented to broaden the scope of solutions accessible to the nondeterministic machine for addressing sub-problems.

Additionally, it's worth noting that the algorithm randomly selects images, ensuring a probabilistic approach where each image has an equal chance of scaling or rotating, enhancing the algorithm's versatility and ensuring fair treatment of all images in the deck.

Moreover, the algorithm attempts to rotate the image as long as a collision with another image exists. In the absence of a collision, the algorithm returns the polygon with the rotation applied up to that point.

These two robust methodologies, in conjunction with the SAT, significantly enhance both the efficiency and accuracy of the algorithm tasked with approximating the optimal configuration of the card. By leveraging the principles of scaling and rotation alongside the capabilities of the SAT, the algorithm can adeptly manipulate the layout of the card to achieve the desired configuration with precision and efficiency.

Persistent Data Management

For the complete implementation of the SpotIT project, it was essential to ensure persistent data management for various entities. This decision stemmed from the recognition that without a robust database framework, extensive calculations would be required before users could initiate gameplay. Such delays would significantly diminish the user experience.

For instance, in the deck creation process, the absence of persistent data would require users to endure prolonged waits for the finite projective plane to be generated and then for each card in the deck to be configured.

In response to this requirement, a relational database framework was implemented. To achieve this, a thorough analysis of the project's entities was conducted. This analysis involved identifying the key components for storing and managing data effectively.

The analysis process commences with a meticulous delineation of the project's entities. After careful consideration, the project team determined that the most pertinent entities for the project were a deck, a card, and an image. By establishing relationships among these entities, the project could be comprehensively realized.

The deck entity serves as a repository for numerous cards, all sharing a common thematic motif. Each card is associated with a single deck.

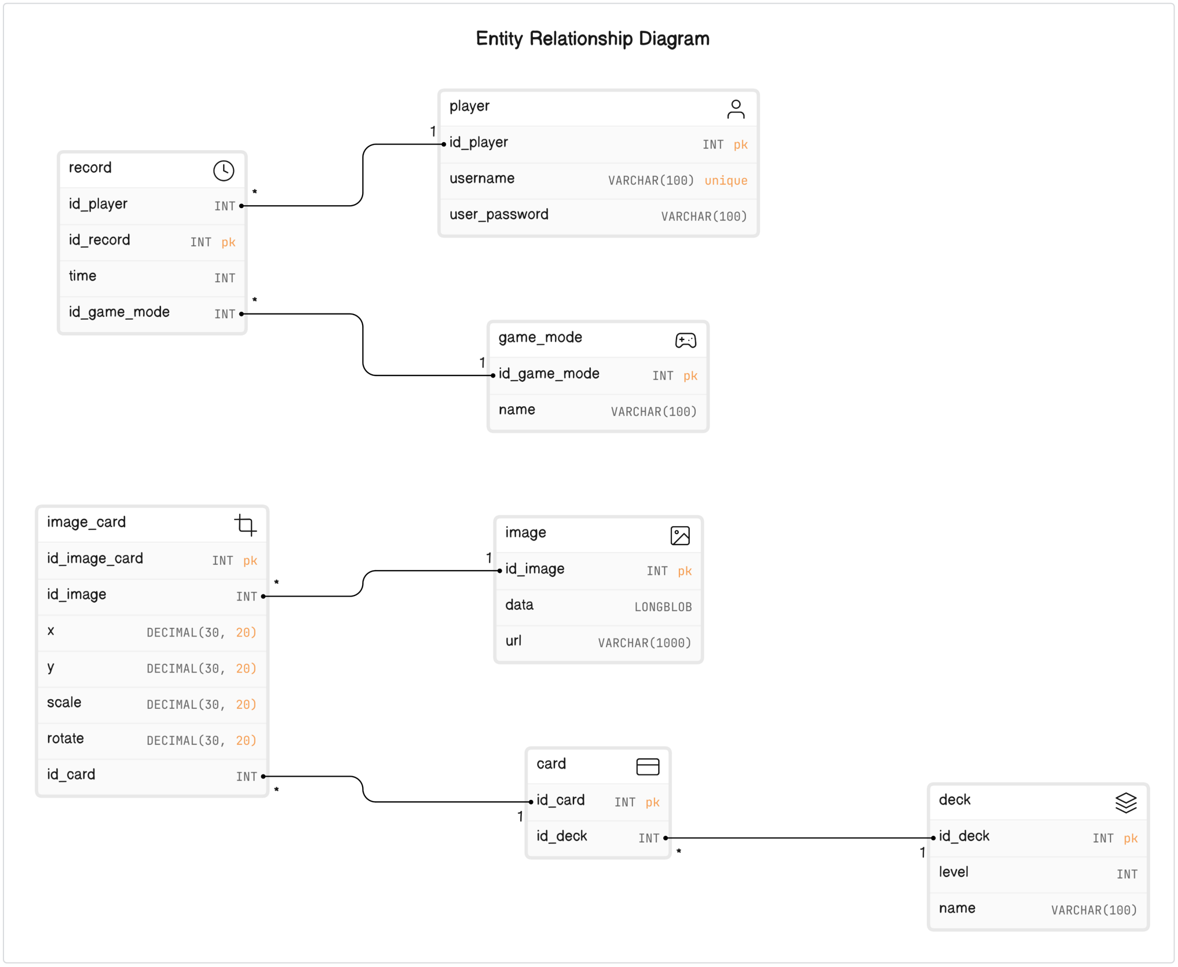
The card entity comprises collections of images and is linked to a deck. To manage the many-to-many relationship between cards and images, an intermediary table, often referred to as a junction or associative table, is employed. This table facilitates the association between cards and images, allowing for efficient retrieval and management of image-card connections.

On the other hand, the image entity consists of PNG files, each associated with multiple cards. This approach facilitates the reuse of images across decks of the same theme, optimizing database storage space.

By combining the entities, the program's entire functionality could be effectively stored in a database, facilitating the reuse of precalculated resources.

For user data persistence, three tables were deployed. The first is the player table, which stores individual user data such as usernames and passwords. Additionally, a game mode table was implemented to assign names to each game mode. Subsequently, an intermediary table was utilized to determine the champion of each game mode by storing the best record achieved among all players. This intermediary table stores information such as the highest score achieved and the time taken to complete each game mode.

Figure 1



After a comprehensive analysis of the database requirements, the project team opted for Amazon Web Services (AWS) as the preferred deployment platform, considering its intuitive interface and cost-effectiveness compared to alternatives like Google Cloud and Oracle Cloud Infrastructure.

The database architecture was meticulously crafted using a combination of AWS services, with Amazon RDS serving as the backbone for robust database management. Specifically, the RDS implementation was based on MySQL, utilizing a t2.micro instance type. Despite its modest specifications, the t2.micro instance proved more than sufficient for the project's needs, offering 20GB of usable storage capacity and running 24/7.

In addition to RDS, Amazon S3 played a crucial role in storing and organizing the 146 images required for the application. Leveraging S3's scalable storage capabilities ensured seamless accessibility to the images from anywhere, further enhancing integration with the database. Notably, the URL field in the image table was populated with links pointing to the corresponding images stored in the S3 bucket, simplifying the retrieval process and optimizing resource management.

To facilitate team access to the database, the team leveraged AWS Identity and Access Management (IAM) service to easily manage and assign roles to each team member. IAM provided a streamlined approach to controlling access permissions, ensuring that team members had the appropriate level of access while maintaining security and compliance standards.

Throughout the deployment process, adherence to AWS best practices and security standards remained a top priority. Regular monitoring and maintenance activities were conducted to ensure optimal performance and address any emerging issues promptly.

In conclusion, the strategic deployment of AWS services, including RDS, S3, and IAM, provided a robust and scalable database infrastructure for the SpotIT application. This approach laid the groundwork for efficient data management, seamless user experience, and streamlined collaboration among team members.

SpotIT Layered Design

This project was developed employing the Model-View-Controller (MVC) pattern, a widely-recognized software architecture that effectively segregates presentation logic from business logic and user interaction. This architectural paradigm comprises three fundamental components:

Model

In the context of the MVC architectural paradigm, the Model assumes a pivotal role, serving as the bedrock that encapsulates the fundamental elements of an application's data structure and business logic. It meticulously organizes data management and operational functionalities, maintaining a strict separation from the user interface and any specific presentation layers.

Within the framework of the SpotIT project, this indispensable component encompasses two key dimensions: persistent data management and the implementation of game rules. While the former aspect has been previously elucidated, the ensuing discourse will delve into explicating the rules governing the SpotIT domain and delineating their realization through the tenets of Object-Oriented Programming.

The Player class is meticulously crafted to embody a participant in a Spotit match across any game mode. This class encapsulates pivotal attributes such as the player's score and furnishes essential functionalities like score incrementation and score comparison between players. Each player instance is endowed with a unique identifier, thereby facilitating their distinct identification within a lobby setting.

The Game class emerges as the linchpin for orchestrating game logic within Spotit. It encompasses a suite of features designed to validate player moves, adjudicate scores based on move validity, and ascertain victors upon match culmination. This class leverages a singular boolean attribute to efficaciously streamline scoring mechanisms across diverse game modes. In Spotit, scoring pivots around two primary dynamics: the reduction or expansion of cards in hand. The boolean attribute adeptly modulates scoring adjustments contingent upon the nature of the move.

The Lobby class assumes a pivotal role as the amalgamation of the Player and Game classes. It functions as an intermediary between the Main class and more specialized functionalities, utilizing a cohort of players in conjunction with a Game object to denote the specific game mode employed within a given lobby. This class serves as a conceptual abstraction of a real-life lobby, encompassing comprehensive management of players within its purview.

Lastly, the Main class epitomizes the apex of the model section of the program. As a singleton, it assumes central importance in overseeing all active lobbies within the server. This class processes incoming JSON payloads containing information pertinent to various requests issued by the controller, subsequently executing corresponding actions in accordance with the request details. Endowed with a plethora of functionalities such as move verification, winner calculation, and lobby creation, the Main class serves as the nerve center for managing the intricate dynamics of Spotit gameplay.

Controller

In the context of the MVC pattern, the Controller serves as a vital intermediary between the views (user interfaces) and the model (application logic and data). Its primary function is to receive user actions from the views, interpret them, and translate them into relevant operations within the model. In the SpotIT project, the controller plays a fundamental role in managing user interactions and coordinating underlying operations in the model, thereby ensuring a clear separation between application logic and its presentation.

In this project, the controller was meticulously crafted through the implementation of a multithreaded HTTP server in Java. At the heart of this architecture lies the **HttpServer** class, which functions as the backbone for handling incoming requests from the view. This server remains vigilant, awaiting queries from the user interface, facilitated by an instance of the **ServerListenerThread** class. It is this dedicated thread that actively listens for requests from the view, ensuring seamless communication between the user and the application.

Upon receiving a request, the **ServerListenerThread** swiftly delegates the task to be processed in a separate thread, utilizing an instance of the **HttpWorker** class. This decoupling of responsibilities ensures efficient utilization of system resources and enhances responsiveness. The **HttpWorker** class assumes the responsibility of encapsulating the received request and dispatching it to the model for further processing.

By orchestrating this intricate dance of threads and classes, the controller adeptly manages the flow of data and commands between the user interface and the underlying application logic. This robust architecture not only ensures the timely handling of user requests but also fosters scalability and maintainability within the project.

View

In the View section, the focus is on the user interface and how interactions with the user are managed. Here's an explanation of how this part of the SpotIT project was implemented:

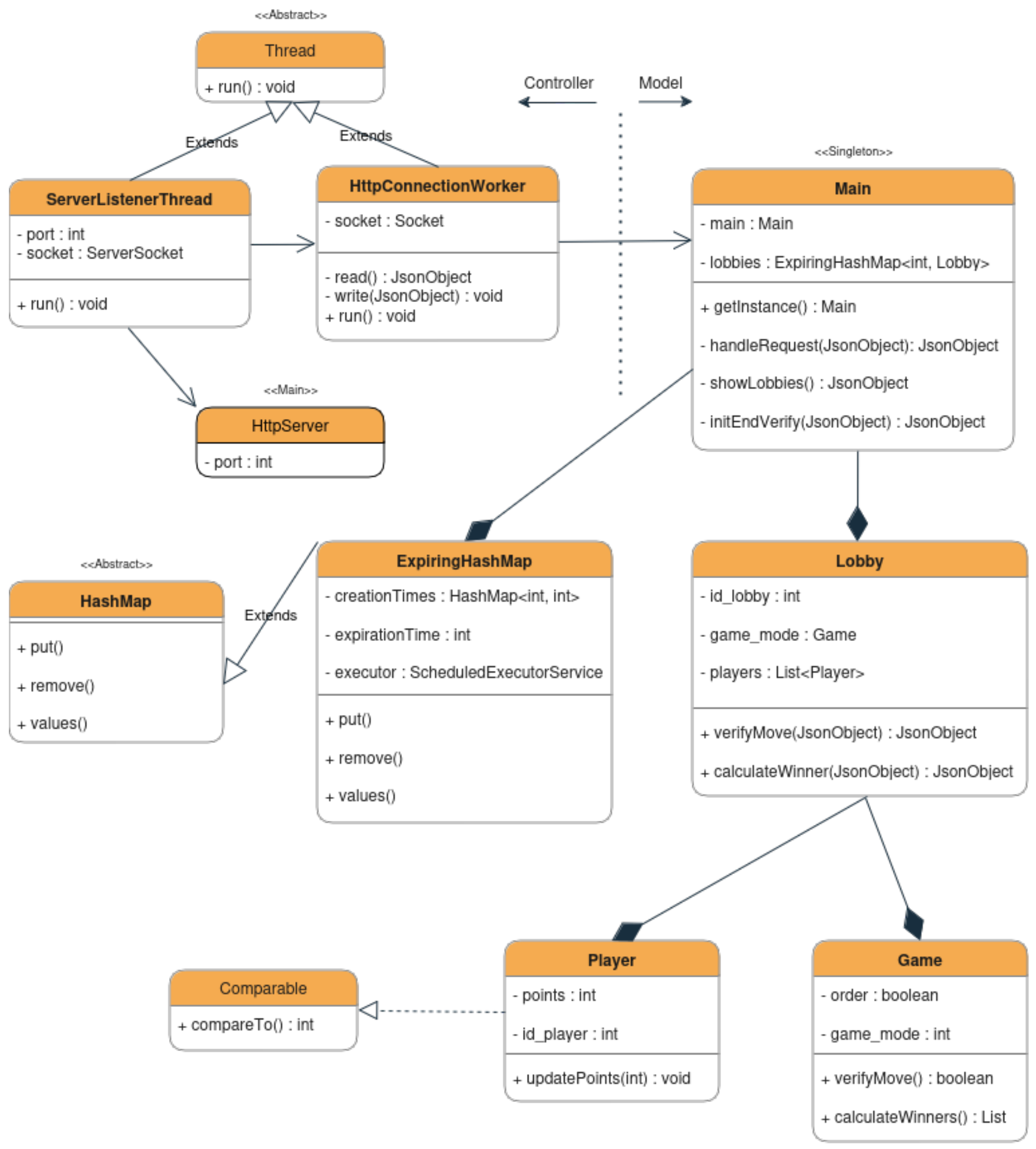
The view in the SpotIT project was developed using modern technologies like React and Vite to efficiently manage the user interface. These technologies enable the creation of dynamic and engaging user interfaces, which is crucial for a satisfactory user experience.

Additionally, server calls were employed to perform verifications and operations related to the game logic. These server calls were handled asynchronously to ensure optimal performance and a seamless user experience.

Furthermore, Firebase was used as a non-relational database to store important game data such as user information, scores, and game settings. Firebase offers seamless integration with web and mobile applications, as well as real-time data synchronization, making it an ideal choice for applications requiring efficient real-time data management.

In summary, the View section of the SpotIT project was developed using modern technologies and efficient approaches to ensure a smooth user experience and effective management of user interactions with the application.

Class Diagram



Conclusions

In conclusion, the development journey of the SpotIT project has been an exhilarating challenge, spanning across various domains of mathematics, including set theory and geometry, alongside intricate database designs and web development intricacies. Delving into the world of interactive gaming applications presented a thrilling endeavour, where the fusion of mathematical concepts with real-world application sparked innovation and creativity.

From designing complex finite projective planes to implementing robust database architectures, each step in the development process posed unique challenges and opportunities for exploration. The integration of cloud services such as Amazon Web Services added a layer of scalability and accessibility, while object-oriented programming principles facilitated modular and maintainable codebases.

Furthermore, the investigation into the relationship between cyclic difference sets (CDS) and finite projective planes (FPP) provided valuable insights into abstract algebraic structures and their practical applications. This research not only deepened our understanding of mathematical theory but also informed the design and implementation of algorithms within the SpotIT project.

Moreover, the utilization of Monte Carlo algorithms for card generation introduced a dynamic and probabilistic element to the gameplay, enhancing the overall user experience. This project has not only showcased the versatility and applicability of mathematical concepts in software development but has also provided valuable insights into the intricacies of collaborative team efforts.

As we reflect on this journey, it becomes evident that the SpotIT project has not only contributed to the advancement of interactive gaming applications but has also fostered a deeper understanding of mathematical principles, database management, cloud computing, and object-oriented programming. It serves as a testament to the power of interdisciplinary collaboration and the endless possibilities that arise when technology and creativity intersect.

Learnings

Danilo Duque

This project serves as proof that mathematics exists even in places we would not anticipate. The relationship between finite projective planes and SpotIT decks makes the game much more interesting because by understanding its magnitude, you can also see its beauty.

Joshua Jiménez

In my case, a learned what the projective finite planes are, and all the maths and logic behind the Spot it game. I also learned more about databases and how can I connect a Java, C or JS program to our database. Finally, now I understand more about image manipulation and geometric algorithms, which helped us a lot of making this project.

Desireé Avilés

I found it intriguing how a seemingly simple game encompassed such a significant amount of mathematical principles. It has truly been a captivating journey exploring the intricate mathematics woven into the fabric of this game.

References

[1] All images used. [Online]. Available: `https://spotit-images.s3.us-east-2.amazonaws.com/`

[2] Amazon Web Services. [Online]. Available: `https://aws.amazon.com/`

[3] Eraser. [Online]. Available: `https://app.eraser.io/`

[4] Oracle Corporation. [Online]. Available: `https://www.oracle.com/`

[5] GitHub. [Online]. Available: `https://github.com/`

[6] Multramate. (s.f.). Finite Projective Planes. [Online]. Available: `https://multramate.github.io/projects/fpp/main.pdf`

[7] Informatics Lab. (s.f.). Cyclic Difference Sets. [Online]. Available: `https://www.inference.org.uk/cds/index.htm`

[8] GeeksforGeeks. (n.d.). MVC Framework - Introduction. [Online]. Available: `https://www.geeksforgeeks.org/mvc-framework-introduction/`

Source Code

To access the source code of this project, please visit our repository at `https://github.com/SpotIt-Game`