**FINAL REPORT**

**NOTE:** “I have attatched the demo prototype with this file please have a look.”

* ***PROJECT TITLE:***

***“*STAKES”**

* ***TEAM MEMBERS:***

The team includes:

**Muhammad Fasih Ul Islam (502518)** played the role of team coordination and resource gathering.

**Muhammad Ammar (501203)** played the role of main program creation i.e the demo.

**Muhammad Ahmad (501395)** played the role of game program testing.

* ***EXCECUTIVE SUMMARY:***

The main learning source nowadays for the youth is, whether we like it or not, movies, games and social media. In this day and age teaching something to students through books would be staying in the past. So we came up with the idea of “STAKES” a business oriented game allowing students to get the experience of stock market, cryto currency and real estate. We have achieved something similar to such a simulator.

* ***PROBLEM STATEMENT AND OBJECTIVES:***

As mentioned, the teaching of business world through games was our main objective as achieving that through old school text books approach is getting difficult these days.

* ***SYSTEM REQUIREMENTS:***
* ***FUNCTIONAL REQUIREMENTS:***

The **key functionalities** include:

1. **Stock Market Simulation:**

Users can view and interact with simulated stock prices in real-time.

Buy and sell stocks based on market conditions.

1. **Cryptocurrency Trading:**

A simulated crypto trading platform where users can trade popular cryptocurrencies.

Crypto prices fluctuate based on pre-programmed algorithms or external feeds.

1. **Real Estate Transactions:**

Players can purchase, sell, or manage properties in the game.

Real estate prices are dynamic and influenced by market trends.

1. **Loan and Banking System:**

Users can take loans with interest rates varying based on their financial status.

Banking options for depositing, withdrawing, or transferring virtual funds.

1. **Diplomatic Relations System:**

The player can view diplomatic relations which would play an important role in market influence.

1. **Progress Saving:**

The program should also be able to save the data of the user so as to start the game from where he left off.

* ***NON-FUNCTIONAL REQUIREMENTS:***

1. ****Performance****

The game should run smoothly at 60 FPS on recommended hardware.

Game operations like buying, selling, or accessing menus should complete in less than 500ms.

1. ****Scalability**:**

The game engine should handle increasing data (e.g., more users or complex calculations) efficiently.

Support for future multiplayer functionality if planned.

1. ****Reliability**:**

The game should recover gracefully from minor crashes, with an autosave feature every 5 minutes.

No more than 0.1% downtime during active user sessions.

1. ****Usability**:**

An intuitive user interface suitable for players aged 12 and above.

Use simple terminology to ensure clarity for users unfamiliar with financial terms.

1. ****Portability**:**

The game must run on Windows PCs with support for macOS and Linux as future goals.

Must work seamlessly on screens with resolutions of 1366x768 and higher.

1. ****Security**:**

User data such as login credentials should be encrypted and securely stored.

Ensure no unauthorized transactions or breaches within the game.

1. ****Maintainability**:**

Modular codebase to allow for easy updates or the addition of new features (e.g., new assets, trading features).

Comprehensive documentation for developers to understand and improve the system.

1. ****Aesthetic Design**:**

Use a visually appealing, modern UI that is easy to navigate.

Employ consistent color schemes and icons representing financial activities.

1. ****Accessibility**:**

Include features like adjustable font sizes and colorblind-friendly themes.

1. ****Storage Requirements**:**

Game installation size should not exceed 1 GB.

Save data files should be under 10 MB per user.

* ***TECHNOLOGY SELECTION:***

#### ****1. Considered Technologies****

**Game Engines**:

Unity and Unreal Engine were considered for their powerful graphics capabilities and built-in physics engines.

Lightweight engines like Godot and custom-built frameworks were reviewed for simplicity and smaller system requirements.

**Programming Frameworks**:

Qt was evaluated for its strong GUI design features and integration with C++.

Alternatives like GTK+ and wxWidgets were analyzed but lacked the same level of flexibility and scalability for creating polished applications.

**Data Storage Options**:

SQLite and flat-file formats (e.g., JSON or CSV) were considered for storing user progress.

Pure in-memory storage using standard C++ data structures was also explored for simplicity.

**Standalone Libraries**:

Graphics and UI libraries like SFML were briefly considered but set aside in favor of Qt for its comprehensive features.

Additional libraries for advanced functionality (e.g., sound or animations) were reviewed but deemed unnecessary for this version.

#### ****2. Justification for Chosen Technologies****

**Qt for Application Development**:

**Reason**: Qt was selected as the primary framework for its comprehensive tools for creating graphical applications. Its robust support for event-driven programming suited the click-based gameplay design.

**Advantages**:

Simplifies GUI creation with a visual designer.

Works seamlessly with C++ for efficient and fast execution.

Provides features for animations and transitions, enhancing the user experience.

**C++ as the Core Programming Language**:

**Reason**: C++ was chosen for its performance, allowing the game to handle various financial calculations efficiently without noticeable lag.

**Advantages**:

Enables fine control over memory and processing resources.

Compatible with Qt, ensuring seamless integration for GUI and logic.

**No External Dependencies**:

**Reason**: To keep the application lightweight and portable, no internet connectivity, AI, or database systems were incorporated.

**Advantages**:

Makes the game self-contained and offline-compatible.

Reduces complexity, ensuring focus on core gameplay mechanics.

**Standard C++ Data Structures**:

**Reason**: Built-in C++ data structures such as vectors, maps, and arrays were used to manage in-game data like user progress and market states.

**Advantages**:

Minimizes external dependencies.

Provides high performance and low overhead for managing game state.

**Standalone Desktop Application**:

**Reason**: The game was developed as a standalone PC-based application to ensure compatibility with systems typically used by the target audience (students).

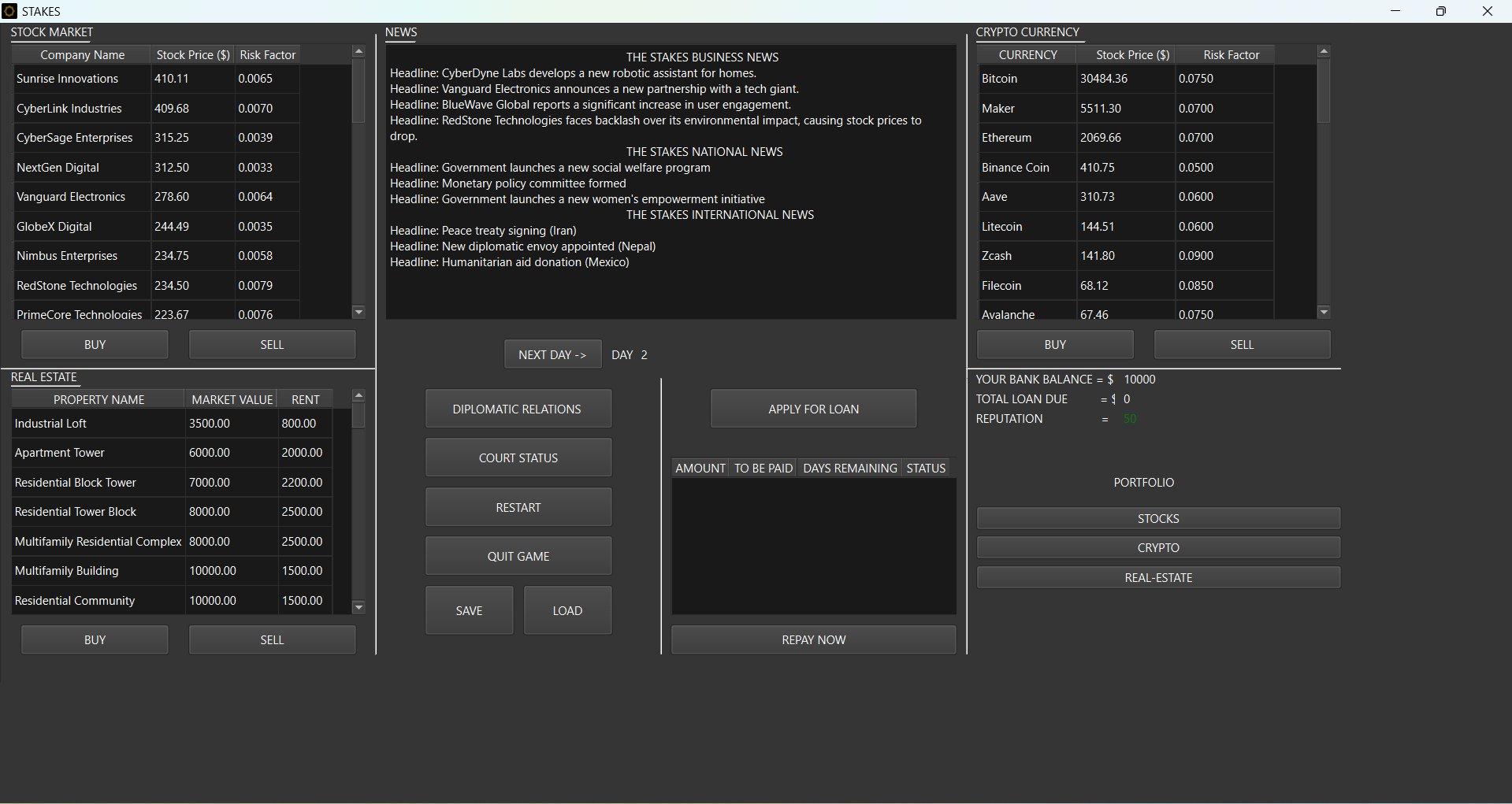
**Advantages**:

Simplifies installation and execution.

Avoids reliance on external servers or connections.

* ***SYSTEM DESIGN:***

**UI:**



### ****Core Components and Functionality****

#### ****1. Stock Market Module****

* **Displayed Area**: Top-left of the interface.
* **Functionality**:
  + Lists companies with their **Stock Price** and **Risk Factor**.
  + Players can choose to **BUY** or **SELL** stocks based on these values.
  + Risk factors simulate volatility and potential returns/losses.
* **Design**: Uses a scrollable table to display data dynamically for multiple companies.

#### ****2. Cryptocurrency Module****

* **Displayed Area**: Top-right of the interface.
* **Functionality**:
  + Displays cryptocurrencies (e.g., Bitcoin, Ethereum) along with their **Stock Price** and **Risk Factor**.
  + Players can execute **BUY** or **SELL** actions for cryptocurrencies.
  + Similar to the stock market, risk factors guide decisions for high-risk, high-reward investments.

#### ****3. Real Estate Module****

* **Displayed Area**: Bottom-left of the interface.
* **Functionality**:
  + Provides a list of properties, their **Market Value**, and **Rent** (return on investment).
  + Players can invest in properties or liquidate assets by selling.
  + Rent acts as a source of recurring income, balancing long-term investment strategies.

#### ****4. News System****

* **Displayed Area**: Center-top of the interface.
* **Functionality**:
  + Displays news headlines grouped into categories like **Business**, **National**, and **International**.
  + Headlines provide clues about changes in market trends and opportunities.
  + Simulates real-world events impacting financial markets.

#### ****5. Financial and Loan Management****

* **Displayed Area**: Center-right of the interface.
* **Functionality**:
  + Shows the player's **Bank Balance**, **Total Loan Due**, and **Reputation**.
  + Players can:
    - **Apply for Loans**: Borrow money for investments, with repayment terms listed in days.
    - **Repay Loans**: Manage debt to maintain a good financial standing.
    - **Track Loan Status**: Check amounts due and repayment timelines.
  + Reputation is affected by timely loan repayments and other actions, influencing gameplay.

#### ****6. Portfolio Management****

* **Displayed Area**: Bottom-right buttons (Portfolio section).
* **Functionality**:
  + Allows players to track their holdings in **Stocks**, **Cryptocurrencies**, and **Real Estate**.
  + Simplifies access to detailed information about owned assets.

#### ****7. Diplomatic Relations and Court Status****

* **Displayed Area**: Center buttons (Diplomatic Relations and Court Status).
* **Functionality**:
  + Players can engage in diplomatic actions or view court-related scenarios.
  + Adds strategic layers to gameplay, with potential penalties or advantages based on choices.

#### ****8. Game Controls****

* **Key Controls**:
  + **Next Day**: Progresses the game by one day, updating markets and loan statuses.
  + **Restart**: Resets the game to its initial state.
  + **Save/Load**: Saves or restores the game's current state, allowing continuity across sessions.
  + **Quit Game**: Exits the game.
* ***IMPLEMENTATION:***

**Graphical User Interface (GUI)**:

**Implementation**:

The GUI was built using **Qt Designer**, allowing for a drag-and-drop creation of widgets like tables, buttons, and labels.

Each module (e.g., stock market, cryptocurrency, real estate) was implemented as separate sections in the main window.

Buttons like **BUY**, **SELL**, **Next Day**, and **Apply for Loan** trigger specific functions written in C++ using Qt’s signal and slot mechanism.

**Example**:

Stock and cryptocurrency tables were populated dynamically using **QTableWidget** with rows updated based on market changes.

**Stock Market and Cryptocurrency Mechanisms**:

**Implementation**:

Stock and cryptocurrency prices were represented using arrays or vectors in C++.

Price fluctuations were generated using random number generators, influenced by pre-defined risk factors to simulate market volatility.

**Example**:

Each company’s stock price was recalculated daily using a formula:  
new\_price = old\_price + (random\_factor \* risk\_factor)

The risk factor ensured some assets were more volatile than others, teaching players about varying levels of risk in investments.

**Real Estate System**:

**Implementation**:

A list of properties was implemented as a vector of structs, where each struct stored attributes like property name, market value, and rent.

The **BUY** and **SELL** buttons updated the player’s portfolio and adjusted their bank balance.

**Example**:

Buying a property reduced the player's bank balance, while owning a property added a daily rent income.

**News Headlines**:

**Implementation**:

Pre-written news headlines were stored in a text file or an array. Headlines were randomly selected and displayed in the **News** section.

Some headlines influenced stock or cryptocurrency prices, simulating real-world market reactions.

**Example**:

A headline like “Government launches a new tech subsidy” increased stock prices for tech companies by a fixed percentage.

**Loan and Reputation System**:

**Implementation**:

Loans were managed using a vector where each loan stored attributes like amount, days remaining, and status.

Daily loan repayment schedules decreased the outstanding amount, and failure to repay loans on time reduced the player’s reputation.

**Example**:

Players applying for a loan triggered a function to deduct funds from their **Bank Balance** over several days.

**Game Progression (Day-by-Day System)**:

**Implementation**:

The **Next Day** button updated all game states, including market prices, loan repayments, and reputation changes.

A day counter was incremented to track player progress.

**Example**:

Market values were recalculated, and loan repayment days were decremented whenever the player moved to the next day.

**Save and Load Features**:

**Implementation**:

Player data (e.g., bank balance, portfolio, loan status) was saved to and loaded from text files using file I/O in C++.

**Example**:

Saving data involved writing all key variables to a file in a structured format. Loading restored the variables and updated the game state.

#### ****Challenges Faced and Solutions****

**Challenge: Managing Dynamic Data in Tables**

**Problem**:

Populating and updating stock, crypto, and real estate tables dynamically while keeping them synced with the backend data structures was complex.

**Solution**:

Used **Qt’s Model-View-Controller (MVC)** framework to link tables directly to backend data. This allowed seamless updates whenever data changed.

**Challenge: Balancing Randomness with Realism**

**Problem**:

Generating random stock price changes often led to unrealistic fluctuations, which made the gameplay confusing.

**Solution**:

Introduced **risk factors** to moderate randomness and ensure prices followed logical trends based on predefined probabilities.

**Challenge: Ensuring Player Engagement**

**Problem**:

Early iterations of the game lacked sufficient interactivity, making it less engaging.

**Solution**:

Added features like **News Headlines**, **Reputation**, and **Diplomatic Relations** to provide a multi-faceted experience beyond basic buying and selling.

**Challenge: File I/O for Save and Load**

**Problem**:

Ensuring data consistency when saving and loading across sessions required careful handling of file formats.

**Solution**:

Used structured file storage (e.g., line-separated values) and implemented error handling to verify file integrity during load operations.

**Challenge: Performance Optimization**

**Problem**:

Frequent updates (e.g., recalculating all market prices) caused slight delays as the number of items in tables increased.

**Solution**:

Optimized data structures (e.g., used hash maps for quick lookups) and limited the scope of recalculations to only visible items.

* ***TESTING AND VALIDATION:***

#### ****1. Functional Testing****

Functional testing focused on verifying that the core features of Stakes worked as intended. Each module was tested individually and integrated into the full system to ensure smooth operation.

| **Feature Tested** | **Expected Outcome** | **Result** | **Remarks** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| **Stock Market Functionality** | Stock prices should update daily. Players should be able to buy/sell stocks, and their bank balance should adjust accordingly. | **Pass** | Prices updated correctly, and transactions adjusted balances accurately. Some edge cases tested, such as buying with insufficient funds. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cryptocurrency Module** | Players should be able to trade cryptocurrencies, with price volatility matching the risk factors. | **Pass** | Trades executed correctly. Risk factors worked as intended to create price fluctuations. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Real Estate Module** | Players should be able to buy/sell properties, and rents should add to their bank balance daily. | **Pass** | Rent income was correctly calculated. Transactions updated the portfolio and bank balance accurately. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Loan Management** | Players should be able to apply for and repay loans. Loan status should update daily, with reputation penalties for non-repayment. | **Pass** | Loan repayment schedules worked as intended. Reputation penalties applied for missed repayments. |

|  |  |  |  |
| --- | --- | --- | --- |
| **News Headlines** | News should display randomly selected headlines that influence market trends. | **Pass** | Headlines displayed correctly and correlated with minor price shifts in stock/crypto markets as per pre-set logic. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Save and Load Functionality** | Game state (e.g., portfolio, bank balance, loans) should save to a file and reload correctly. | **Pass** | Save and load tested successfully with different scenarios. Error handling worked for corrupt/missing files. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Next Day Progression** | Advancing the day should update all markets, loan statuses, and rent incomes. | **Pass** | Day progression worked flawlessly. All modules updated as expected. |

#### ****2. User Testing****

User testing involved a small group of participants from the target audience (young students) to evaluate the usability, engagement, and overall experience.

| **Aspect Evaluated** | **Feedback** | **Action Taken** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Ease of Use** | Players found the interface simple and intuitive. | No significant changes required. |

|  |  |  |
| --- | --- | --- |
| **Educational Value** | Users appreciated learning about financial management through gameplay. | Consider including a tutorial mode for first-time users to enhance educational impact. |

|  |  |  |
| --- | --- | --- |
| **Game Engagement** | Players found the gameplay engaging but suggested adding more events or random challenges. | Plan to introduce optional random events in future updates (e.g., economic crises, bonus opportunities). |

|  |  |  |
| --- | --- | --- |
| **News and Market Impact** | Some players felt the news headlines weren’t always clear about how they impacted the markets. | Improved the logic linking specific news headlines to visible stock/crypto price changes for better clarity. |

|  |  |  |
| --- | --- | --- |
| **Reputation Mechanism** | Users liked the reputation system but wanted more visible feedback when reputation changed. | Added on-screen indicators when reputation increased or decreased during the game. |

|  |  |  |
| --- | --- | --- |
| **Performance** | No lag or performance issues were reported. | No action required. |

|  |  |  |
| --- | --- | --- |
| **Save and Load** | Users appreciated the ability to save progress but wanted auto-save functionality. | Auto-save feature added for smoother gameplay experience. |

**KEY INSIGHTS AND OUTCOMES:**

**Functionality Success**:

All core functionalities performed as expected with no major bugs.

Testing confirmed the game logic, including financial calculations, price updates, and reputation penalties, worked as designed.

**User Engagement**:

The game’s intuitive design and click-based mechanics were well-received by users.

Feedback highlighted its educational and entertainment value, meeting the project’s primary goals.

**Areas for Improvement**:

Add a beginner tutorial or guided mode for new players.

Include more random events or scenarios to enhance replay-ability.

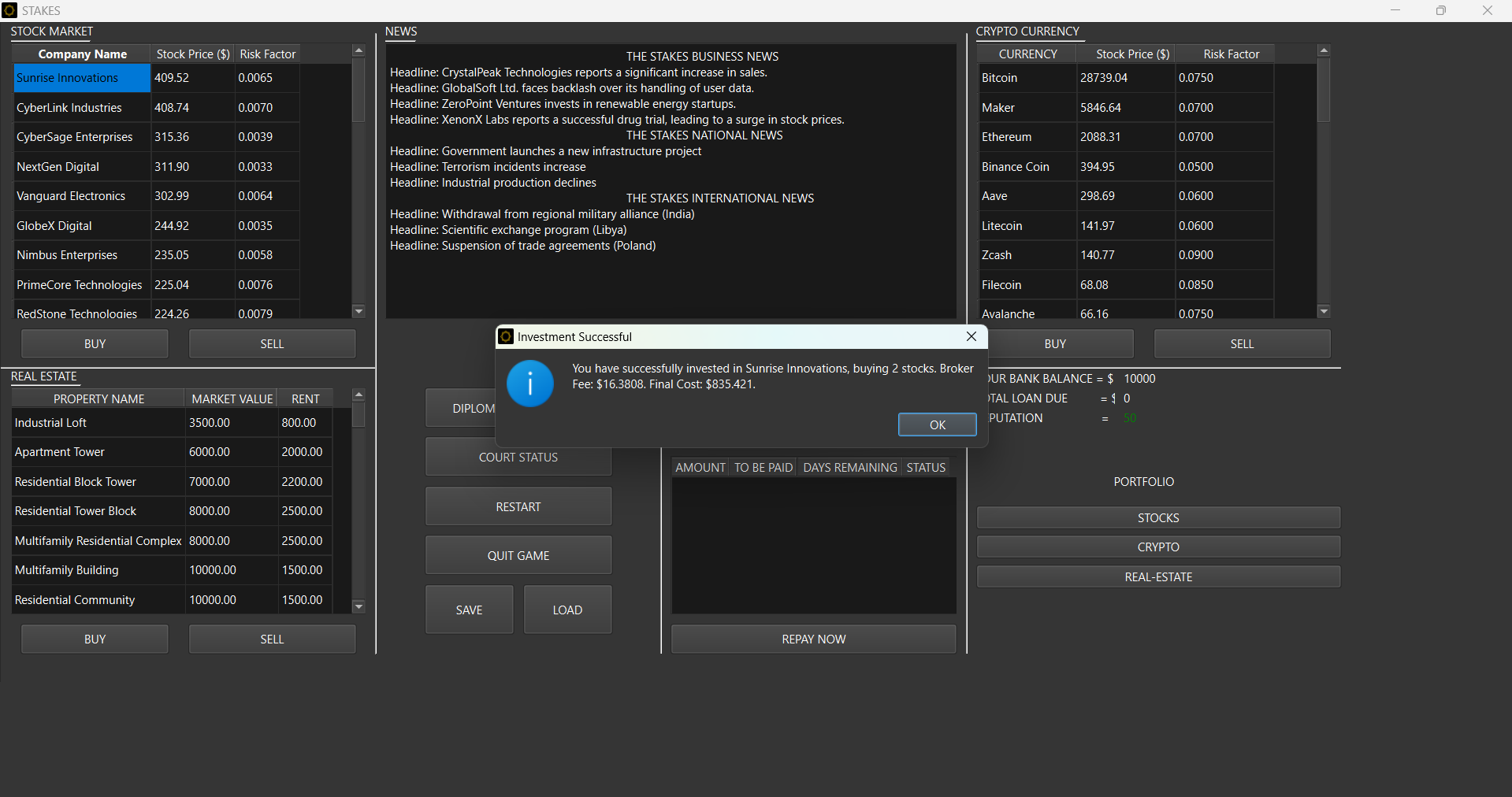
Make market changes from news events more visually evident.

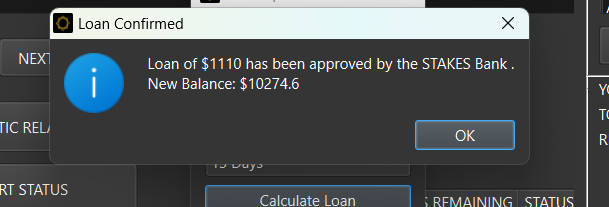
**Overall Testing Results**:

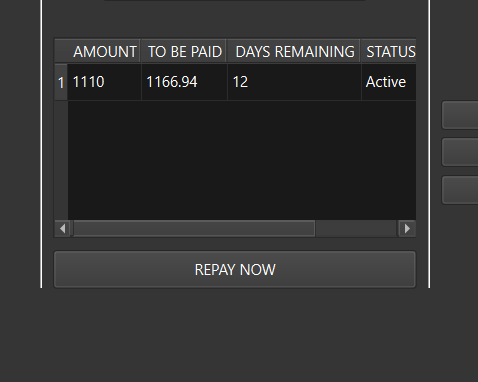
The system successfully passed both functional and user testing.

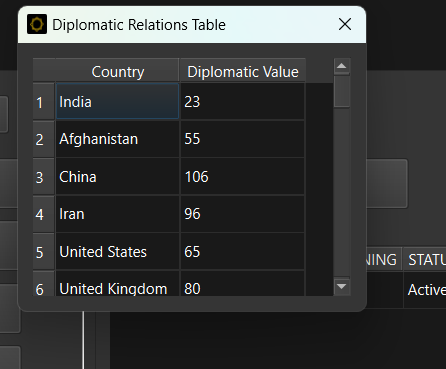
The game is ready for deployment with minor enhancements planned for future iterations.

**SOME SCREENSHOTS AS EVIDENCE OF CLAIMS:**









* ***CONCLUSION AND FUTURE WORK:***

The program has a lot of room for improvement and this is just the beginning. We can integrate real stock market and trends using APIs and databases for large storage. We can use trend analyzers for making much more realistic stock market changes. In other words nothing is ever really complete and this application is no different.

* ***REFERENCES:***

We didn’t have or use much references for this application.