FracEstate Phase 2: Game Economy & Financial Engine - Development Prompt

Project Context & Getting Started Tonight

You are developing **Phase 2** of FracEstate, a comprehensive real estate investment simulation game. Phase 1 (Property Generation & Market System) has been completed, and now you need to implement the core financial engine that powers the game's economy.

Repository: https://github.com/mghondo/BlockChain-RealEstateTransaction
Technology Stack: React 18 + TypeScript, Firebase Firestore, Vite build tool
Current Status: Ready for Phase 2 development
Tonight's Goal: Get the foundation systems running

PHASE 2 PRIORITY ORDER (Start Tonight)

🚀 TONIGHT: Core Time System (2-3 hours)

1. Time progression engine (1 hour = 2 months)
2. Basic background calculation framework
3. User session tracking

🌅 TOMORROW: Income & API Integration

4. Rental income system (monthly USDC payments)
5. Property appreciation (quarterly)
6. CoinGecko API integration for crypto prices

📊 DAY 3+: Advanced Features

7. Tax calculation system
8. Property sale mechanics
9. Offline progress catchup animations

---

1. CORE TIME PROGRESSION SYSTEM ⏰

Critical Concept: Game NEVER resets. User can be offline for days/weeks/months and game continues running. When they return, system calculates ALL missed progress in under 3 seconds.

A. Time Management Hook (START HERE)

```typescript
```

```
// src/hooks/useGameTime.ts
import { useState, useEffect, useRef } from 'react';
import { doc, getDoc, updateDoc, serverTimestamp } from 'firebase/firestore';
import { db } from '../config/firebase';

interface GameTimeState {
  currentGameTime: Date;
  lastRealTime: Date;
  gameStartTime: Date;
  isCalculatingOfflineProgress: boolean;
  offlineProgressCompleted: boolean;
}

export const useGameTime = (userId: string) => {
  const [gameTime, setGameTime] = useState<GameTimeState | null>(null);
  const intervalRef = useRef<NodeJS.Timeout>();

  // CRITICAL: 1 real hour = 2 game months
  // 1 real minute = 1 game hour
  // Multiplier: 60 * 24 * 60 = 86400 (1 real minute = 60 game hours)
  const TIME_MULTIPLIER = 1440; // 1 real hour = 60 game days = 2 game months

  const calculateGameTime = (realTime: Date, baseGameTime: Date, baseRealTime: Date): Date => {
    const realElapsedMs = realTime.getTime() - baseRealTime.getTime();
    const gameElapsedMs = realElapsedMs * TIME_MULTIPLIER;
    return new Date(baseGameTime.getTime() + gameElapsedMs);
  };

  const processOfflineProgress = async (lastSeen: Date): Promise<OfflineProgress> => {
    const now = new Date();
    const offlineRealTime = now.getTime() - lastSeen.getTime();
    const offlineGameTime = offlineRealTime * TIME_MULTIPLIER;
    const gameMonthsElapsed = Math.floor(offlineGameTime / (1000 * 60 * 60 * 24 * 30)); // Approx game months

    // TODO: Calculate missed rental income, appreciation, property changes
    return {
      realTimeOffline: offlineRealTime,
      gameTimeElapsed: offlineGameTime,
      gameMonthsElapsed,
      rentalIncome: 0, // Calculate in rental service
      appreciation: 0, // Calculate in appreciation service
      newProperties: [], // Calculate in property service
    };
```

```
};

// Initialize or resume game time
useEffect(() => {
  if (!userId) return;

  const initializeGameTime = async () => {
    const userTimeDoc = await getDoc(doc(db, 'gameTime', userId));
    const now = new Date();

    if (userTimeDoc.exists()) {
      // Returning user - calculate offline progress
      const data = userTimeDoc.data();
      const lastRealTime = data.lastRealTime.toDate();
      const lastGameTime = data.currentGameTime.toDate();

      // Check if user was offline for more than 5 minutes (significant time)
      const offlineMs = now.getTime() - lastRealTime.getTime();
      if (offlineMs > 5 * 60 * 1000) { // 5 minutes
        setGameTime(prev => ({ ...prev, isCalculatingOfflineProgress: true }));

        // Process offline progress
        const offlineProgress = await processOfflineProgress(lastRealTime);

        // Update game time to current
        const newGameTime = calculateGameTime(now, lastGameTime, lastRealTime);

        setGameTime({
          currentGameTime: newGameTime,
          lastRealTime: now,
          gameStartTime: data.gameStartTime.toDate(),
          isCalculatingOfflineProgress: false,
          offlineProgressCompleted: true,
        });

        // TODO: Show offline progress modal
      } else {
        // User just resumed - continue from where they left off
        const newGameTime = calculateGameTime(now, lastGameTime, lastRealTime);
        setGameTime({
          currentGameTime: newGameTime,
          lastRealTime: now,
          gameStartTime: data.gameStartTime.toDate(),
          isCalculatingOfflineProgress: false,
          offlineProgressCompleted: false,
        });
      }
    } else {
      // New user - initialize game time
      const gameStartTime = now;
      setGameTime({
        currentGameTime: gameStartTime,
        lastRealTime: now,
        gameStartTime,
        isCalculatingOfflineProgress: false,
        offlineProgressCompleted: false,
      });

      // Save to Firebase
      await updateDoc(doc(db, 'gameTime', userId), {
        currentGameTime: gameStartTime,
        lastRealTime: serverTimestamp(),
        gameStartTime: gameStartTime,
      });
    }
  };

  initializeGameTime();
}, [userId]);

// Update game time every minute
useEffect(() => {
  if (!gameTime) return;

  intervalRef.current = setInterval(() => {
    const now = new Date();
    const newGameTime = calculateGameTime(now, gameTime.currentGameTime, gameTime.lastRealTime);

    setGameTime(prev => ({
      ...prev!,
      currentGameTime: newGameTime,
      lastRealTime: now,
    }));

    // Update Firebase every 5 minutes
    if (now.getMinutes() % 5 === 0) {
      updateDoc(doc(db, 'gameTime', userId), {
        currentGameTime: newGameTime,
        lastRealTime: serverTimestamp(),
      });
    }
  }, 60000); // Update every minute

  return () => {
    if (intervalRef.current) {
      clearInterval(intervalRef.current);
    }
  };
}, [gameTime, userId]);

return {
  gameTime: gameTime?.currentGameTime,
  isCalculatingOfflineProgress: gameTime?.isCalculatingOfflineProgress || false,
  offlineProgressCompleted: gameTime?.offlineProgressCompleted || false,
  realTime: gameTime?.lastRealTime,
  gameStartTime: gameTime?.gameStartTime,
};
};

interface OfflineProgress {
  realTimeOffline: number;
  gameTimeElapsed: number;
  gameMonthsElapsed: number;
  rentalIncome: number;
  appreciation: number;
  newProperties: any[];
}
```

B. Game Clock Display Component

```
typescript
```

```tsx
// src/components/GameTime/GameClock.tsx
import React from 'react';
import { useGameTime } from '../../hooks/useGameTime';
import { useAuth } from '../../hooks/useAuth';

interface GameClockProps {
  showDetailed?: boolean;
}

export const GameClock: React.FC<GameClockProps> = ({ showDetailed = false }) => {
  const { user } = useAuth();
  const { gameTime, realTime, gameStartTime } = useGameTime(user?.uid || '');

  if (!gameTime) {
    return (
      <div className="flex items-center space-x-2 text-sm text-gray-400">
        <div className="w-2 h-2 bg-gray-400 rounded-full animate-pulse"></div>
        <span>Syncing game time...</span>
      </div>
    );
  }

  const formatGameDate = (date: Date): string => {
    return date.toLocaleDateString('en-US', {
      year: 'numeric',
      month: 'long',
      day: 'numeric',
    });
  };

  const formatGameTime = (date: Date): string => {
    return date.toLocaleTimeString('en-US', {
      hour: '2-digit',
      minute: '2-digit',
      hour12: true,
    });
  };

  const calculateGameAge = (): string => {
    if (!gameStartTime) return '';
    const ageMs = gameTime.getTime() - gameStartTime.getTime();
    const gameYears = Math.floor(ageMs / (1000 * 60 * 60 * 24 * 365));
    const gameMonths = Math.floor((ageMs % (1000 * 60 * 60 * 24 * 365)) / (1000 * 60 * 60 * 24 * 30));
    if (gameYears > 0) {
      return `${gameYears}y ${gameMonths}m`;
    }
    return `${gameMonths} months`;
  };

  return (
    <div className="bg-gray-800 rounded-lg p-3 border border-gray-700">
      <div className="flex items-center justify-between">
        <div className="flex items-center space-x-2">
          <div className="w-2 h-2 bg-green-500 rounded-full animate-pulse"></div>
          <span className="text-xs text-gray-400">GAME TIME</span>
        </div>
        <span className="text-xs text-gray-400">1hr = 2 months</span>
      </div>

      <div className="mt-2">
        <div className="text-lg font-semibold text-white">
          {formatGameDate(gameTime)}
        </div>
        <div className="text-sm text-gray-300">
          {formatGameTime(gameTime)}
        </div>
      </div>

      {showDetailed && (
        <div className="mt-3 pt-3 border-t border-gray-700 space-y-1">
          <div className="flex justify-between text-xs">
            <span className="text-gray-400">Portfolio Age:</span>
            <span className="text-white">{calculateGameAge()}</span>
          </div>
          <div className="flex justify-between text-xs">
            <span className="text-gray-400">Real Time:</span>
            <span className="text-white">{realTime?.toLocaleTimeString()}</span>
          </div>
        </div>
      )}
    </div>
  );
};
```

C. User Session Tracking Service

typescript

```
// src/services/sessionService.ts
import { doc, updateDoc, getDoc, serverTimestamp } from 'firebase/firestore';
import { db } from '../config/firebase';

interface UserSession {
  userId: string;
  lastActiveTime: Date;
  gameTimeWhenLeft: Date;
  isOnline: boolean;
  deviceInfo: string;
  sessionCount: number;
}

export class SessionService {
  static async updateLastSeen(userId: string, gameTime: Date): Promise<void> {
    try {
      await updateDoc(doc(db, 'userSessions', userId), {
        lastActiveTime: serverTimestamp(),
        gameTimeWhenLeft: gameTime,
        isOnline: true,
        lastHeartbeat: serverTimestamp(),
      });
    } catch (error) {
      console.error('Error updating last seen:', error);
    }
  }

  static async getUserSession(userId: string): Promise<UserSession | null> {
    try {
      const sessionDoc = await getDoc(doc(db, 'userSessions', userId));
      if (sessionDoc.exists()) {
        const data = sessionDoc.data();
        return {
          userId,
          lastActiveTime: data.lastActiveTime.toDate(),
          gameTimeWhenLeft: data.gameTimeWhenLeft.toDate(),
          isOnline: data.isOnline || false,
          deviceInfo: data.deviceInfo || '',
          sessionCount: data.sessionCount || 0,
        };
      }
      return null;
    } catch (error) {
      console.error('Error getting user session:', error);
      return null;
    }
  }

  static async markUserOffline(userId: string): Promise<void> {
    try {
      await updateDoc(doc(db, 'userSessions', userId), {
        isOnline: false,
        lastSeenTime: serverTimestamp(),
      });
    } catch (error) {
      console.error('Error marking user offline:', error);
    }
  }

  static async initializeSession(userId: string, gameTime: Date): Promise<void> {
    try {
      const sessionDoc = await getDoc(doc(db, 'userSessions', userId));
      const sessionCount = sessionDoc.exists() ? (sessionDoc.data().sessionCount || 0) + 1 : 1;

      await updateDoc(doc(db, 'userSessions', userId), {
        lastActiveTime: serverTimestamp(),
        gameTimeWhenLeft: gameTime,
        isOnline: true,
        sessionCount,
        deviceInfo: navigator.userAgent,
        loginTime: serverTimestamp(),
      });
    } catch (error) {
      console.error('Error initializing session:', error);
    }
  }
}

// Hook to handle session lifecycle
export const useUserSession = (userId: string, gameTime: Date | null) => {
  useEffect(() => {
    if (!userId || !gameTime) return;

    // Initialize session on mount
    SessionService.initializeSession(userId, gameTime);

    // Update heartbeat every 30 seconds
    const heartbeatInterval = setInterval(() => {
      SessionService.updateLastSeen(userId, gameTime);
    }, 30000);

    // Mark offline on unmount/page close
    const handleBeforeUnload = () => {
      SessionService.markUserOffline(userId);
    };

    window.addEventListener('beforeunload', handleBeforeUnload);

    return () => {
      clearInterval(heartbeatInterval);
      window.removeEventListener('beforeunload', handleBeforeUnload);
      SessionService.markUserOffline(userId);
    };
  }, [userId, gameTime]);
};
```

---

2. RENTAL INCOME SYSTEM 💰

A. Rental Income Service

```
typescript
```

```
// src/services/rentalIncomeService.ts
import { collection, query, where, getDocs, addDoc, updateDoc, doc } from 'firebase/firestore';
import { db } from '../config/firebase';

interface RentalPayment {
  id?: string;
  userId: string;
  propertyId: string;
  amount: number; // USDC
  sharesOwned: number;
  totalPropertyValue: number;
  rentalYield: number;
  paymentMonth: string; // "2025-01"
  gameDate: Date;
  realDate: Date;
  processed: boolean;
}

interface PropertyRentalInfo {
  propertyId: string;
  currentValue: number;
  rentalYield: number; // percentage (e.g., 8.5 for 8.5%)
  totalShares: number; // always 100
}

export class RentalIncomeService {

  // Calculate monthly rental income for a user's shares in a property
  static calculateMonthlyRental(
    propertyValue: number,
    rentalYield: number,
    userShares: number
  ): number {
    // Annual rental income = property value * (yield / 100)
    // Monthly rental income = annual / 12
    // User's portion = (monthly rental / 100 shares) * user shares

    const annualRental = propertyValue * (rentalYield / 100);
    const monthlyRental = annualRental / 12;
    const monthlyRentalPerShare = monthlyRental / 100;
    const userMonthlyRental = monthlyRentalPerShare * userShares;

    return Number(userMonthlyRental.toFixed(2));
```

```
}

// Process rental income for all users who own shares in properties
static async processMonthlyRentals(gameDate: Date): Promise<void> {
  try {
    console.log('Processing monthly rentals for game date:', gameDate);

    // Get all active properties
    const propertiesQuery = query(
      collection(db, 'properties'),
      where('status', '==', 'active')
    );
    const propertiesSnapshot = await getDocs(propertiesQuery);

    // Get all user investments
    const investmentsQuery = query(collection(db, 'investments'));
    const investmentsSnapshot = await getDocs(investmentsQuery);

    const paymentPromises: Promise<void>[] = [];

    // Process each investment
    investmentsSnapshot.docs.forEach(investmentDoc => {
      const investment = investmentDoc.data();
      const property = propertiesSnapshot.docs.find(p => p.id === investment.propertyId);

      if (property) {
        const propertyData = property.data();
        const rentalAmount = this.calculateMonthlyRental(
          propertyData.currentValue,
          propertyData.rentalYield,
          investment.sharesOwned
        );

        const paymentPromise = this.creditRentalIncome(
          investment.userId,
          investment.propertyId,
          rentalAmount,
          investment.sharesOwned,
          propertyData.currentValue,
          propertyData.rentalYield,
          gameDate
        );

        paymentPromises.push(paymentPromise);
      }
    });

    await Promise.all(paymentPromises);
    console.log(`Processed ${paymentPromises.length} rental payments`);

  } catch (error) {
    console.error('Error processing monthly rentals:', error);
    throw error;
  }
}

// Credit rental income to user's USDC balance
static async creditRentalIncome(
  userId: string,
  propertyId: string,
  amount: number,
  sharesOwned: number,
  propertyValue: number,
  rentalYield: number,
  gameDate: Date
): Promise<void> {
  try {
    const paymentMonth = `${gameDate.getFullYear()}-${String(gameDate.getMonth() + 1).padStart(2, '0')}`;

    // Create rental payment record
    const rentalPayment: Omit<RentalPayment, 'id'> = {
      userId,
      propertyId,
      amount,
      sharesOwned,
      totalPropertyValue: propertyValue,
      rentalYield,
      paymentMonth,
      gameDate,
      realDate: new Date(),
      processed: true,
    };

    // Add to rental income collection
    await addDoc(collection(db, 'rentalIncome'), rentalPayment);

    // Update user's USDC balance
    const userDoc = doc(db, 'users', userId);
    const userSnapshot = await getDocs(query(collection(db, 'users'), where('__name__', '==', userId)));

    if (!userSnapshot.empty) {
      const userData = userSnapshot.docs[0].data();
      const newUsdcBalance = (userData.usdcBalance || 0) + amount;

      await updateDoc(userDoc, {
        usdcBalance: newUsdcBalance,
        lastRentalPayment: gameDate,
      });
    }

  } catch (error) {
    console.error('Error crediting rental income:', error);
    throw error;
  }
}

// Get rental income history for a user
static async getRentalHistory(userId: string, limit: number = 50): Promise<RentalPayment[]> {
  try {
    const rentalQuery = query(
      collection(db, 'rentalIncome'),
      where('userId', '==', userId),
      // orderBy('gameDate', 'desc'),
      // limit(limit)
    );

    const rentalSnapshot = await getDocs(rentalQuery);
    return rentalSnapshot.docs.map(doc => ({
      id: doc.id,
      ...doc.data(),
      gameDate: doc.data().gameDate.toDate(),
      realDate: doc.data().realDate.toDate(),
    })) as RentalPayment[];

  } catch (error) {
    console.error('Error getting rental history:', error);
    return [];
  }
}

// Calculate total rental income for a user
static async getTotalRentalIncome(userId: string): Promise<number> {
    try {
      const history = await this.getRentalHistory(userId);
      return history.reduce((total, payment) => total + payment.amount, 0);
    } catch (error) {
      console.error('Error calculating total rental income:', error);
      return 0;
    }
  }
}
```

B. Rental Income Display Component

```typescript
```

```tsx
// src/components/Income/RentalIncomeTracker.tsx
import React, { useState, useEffect } from 'react';
import { RentalIncomeService } from '../../services/rentalIncomeService';
import { useAuth } from '../../hooks/useAuth';

interface RentalIncomeTrackerProps {
  className?: string;
}

export const RentalIncomeTracker: React.FC<RentalIncomeTrackerProps> = ({ className = '' }) => {
  const { user } = useAuth();
  const [totalIncome, setTotalIncome] = useState<number>(0);
  const [monthlyProjection, setMonthlyProjection] = useState<number>(0);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    if (!user?.uid) return;

    const loadRentalData = async () => {
      try {
        setLoading(true);

        // Get total rental income earned
        const total = await RentalIncomeService.getTotalRentalIncome(user.uid);
        setTotalIncome(total);

        // TODO: Calculate monthly projection based on current holdings
        // This would require getting user's current investments and calculating
        // expected monthly rental based on their portfolio

        setLoading(false);
      } catch (error) {
        console.error('Error loading rental data:', error);
        setLoading(false);
      }
    };

    loadRentalData();
  }, [user?.uid]);

  if (loading) {
    return (
      <div className={`bg-gray-800 rounded-lg p-4 border border-gray-700 ${className}`}>
      <div className="animate-pulse">
        <div className="h-4 bg-gray-700 rounded w-3/4 mb-2"></div>
        <div className="h-8 bg-gray-700 rounded w-1/2"></div>
      </div>
    </div>
  );
}

return (
  <div className={`bg-gray-800 rounded-lg p-4 border border-gray-700 ${className}`}>
    <div className="flex items-center justify-between mb-3">
      <h3 className="text-lg font-semibold text-white">Rental Income</h3>
      <div className="flex items-center text-green-500">
        <div className="w-2 h-2 bg-green-500 rounded-full mr-2 animate-pulse"></div>
        <span className="text-xs">EARNING</span>
      </div>
    </div>

    <div className="space-y-3">
      <div>
        <div className="text-xs text-gray-400 mb-1">Total Earned</div>
        <div className="text-2xl font-bold text-green-500">
          ${totalIncome.toLocaleString('en-US', { minimumFractionDigits: 2 })} USDC
        </div>
      </div>

      <div>
        <div className="text-xs text-gray-400 mb-1">Monthly Projection</div>
        <div className="text-lg font-semibold text-white">
          ${monthlyProjection.toLocaleString('en-US', { minimumFractionDigits: 2 })} USDC/month
        </div>
      </div>

      <div className="pt-3 border-t border-gray-700">
        <div className="text-xs text-gray-400">
          Next payment in: <span className="text-white font-medium">12 game days</span>
        </div>
      </div>
    </div>
  </div>
);
};
```

3. PROPERTY APPRECIATION SYSTEM 📈

A. Appreciation Service

```typescript
```

```
// src/services/appreciationService.ts
import { collection, query, where, getDocs, updateDoc, doc, addDoc } from 'firebase/firestore';
import { db } from '../config/firebase';

interface AppreciationEvent {
 id?: string;
 propertyId: string;
 previousValue: number;
 newValue: number;
 appreciationRate: number; // percentage for this quarter
 quarter: string; // "2025-Q1"
 gameDate: Date;
 realDate: Date;
 affectedUsers: string[];
}

interface AppreciationConfig {
 [key: string]: {
  minAnnual: number;
  maxAnnual: number;
 };
}

const APPRECIATION_RATES: AppreciationConfig = {
 'A': { minAnnual: 0.04, maxAnnual: 0.10 }, // 4-10% annually
 'B': { minAnnual: 0.03, maxAnnual: 0.08 }, // 3-8% annually
 'C': { minAnnual: 0.02, maxAnnual: 0.06 }, // 2-6% annually
};

export class AppreciationService {

 // Calculate quarterly appreciation for a property
 static calculateQuarterlyAppreciation(
  propertyClass: 'A' | 'B' | 'C',
  currentValue: number
 ): { newValue: number; appreciationRate: number } {
  const config = APPRECIATION_RATES[propertyClass];

  // Random annual rate within class range
  const annualRate = config.minAnnual + Math.random() * (config.maxAnnual - config.minAnnual);

  // Convert to quarterly rate (with some randomness)
  const baseQuarterlyRate = annualRate / 4;
```

```
    const variance = baseQuarterlyRate * 0.3; // 30% variance
    const quarterlyRate = baseQuarterlyRate + (Math.random() - 0.5) * variance;

    // Ensure minimum positive appreciation
    const finalRate = Math.max(quarterlyRate, 0.001); // Minimum 0.1% per quarter

    const newValue = currentValue * (1 + finalRate);
    const appreciationPercentage = finalRate * 100;

    return {
      newValue: Number(newValue.toFixed(2)),
      appreciationRate: Number(appreciationPercentage.toFixed(3)),
    };
  }

  // Process appreciation for all active properties
  static async processQuarterlyAppreciation(gameDate: Date): Promise<void> {
    try {
      console.log('Processing quarterly appreciation for game date:', gameDate);

      // Get all active properties
      const propertiesQuery = query(
        collection(db, 'properties'),
        where('status', 'in', ['available', 'ending_soon', 'sold_out'])
      );
      const propertiesSnapshot = await getDocs(propertiesQuery);

      const appreciationPromises: Promise<void>[] = [];

      for (const propertyDoc of propertiesSnapshot.docs) {
        const property = propertyDoc.data();
        const propertyId = propertyDoc.id;

        const appreciationPromise = this.applyAppreciationToProperty(
          propertyId,
          property.class,
          property.currentValue,
          gameDate
        );

        appreciationPromises.push(appreciationPromise);
      }

      await Promise.all(appreciationPromises);
    console.log(`Processed appreciation for ${appreciationPromises.length} properties`);

  } catch (error) {
    console.error('Error processing quarterly appreciation:', error);
    throw error;
  }
}

// Apply appreciation to a specific property
static async applyAppreciationToProperty(
  propertyId: string,
  propertyClass: 'A' | 'B' | 'C',
  currentValue: number,
  gameDate: Date
): Promise<void> {
  try {
    const { newValue, appreciationRate } = this.calculateQuarterlyAppreciation(
      propertyClass,
      currentValue
    );

    // Update property value
    await updateDoc(doc(db, 'properties', propertyId), {
      currentValue: newValue,
      lastAppreciation: gameDate,
      lastAppreciationRate: appreciationRate,
    });

    // Get affected users (investors in this property)
    const investmentsQuery = query(
      collection(db, 'investments'),
      where('propertyId', '==', propertyId)
    );
    const investmentsSnapshot = await getDocs(investmentsQuery);
    const affectedUsers = investmentsSnapshot.docs.map(doc => doc.data().userId);

    // Create appreciation event record
    const quarter = `${gameDate.getFullYear()}-Q${Math.ceil((gameDate.getMonth() + 1) / 3)}`;

    const appreciationEvent: Omit<AppreciationEvent, 'id'> = {
      propertyId,
      previousValue: currentValue,
      newValue,
      appreciationRate,
    quarter,
    gameDate,
    realDate: new Date(),
    affectedUsers,
  };

  await addDoc(collection(db, 'propertyAppreciation'), appreciationEvent);

  // Update user investment values
  for (const investmentDoc of investmentsSnapshot.docs) {
    const investment = investmentDoc.data();
    const newInvestmentValue = (newValue / 100) * investment.sharesOwned;

    await updateDoc(doc(db, 'investments', investmentDoc.id), {
      currentValue: newInvestmentValue,
      lastUpdated: gameDate,
    });
  }

  // Check if property is now eligible for sale (40-50% appreciation)
  const totalAppreciation = ((newValue - investment.purchasePrice) / investment.purchasePrice)
```